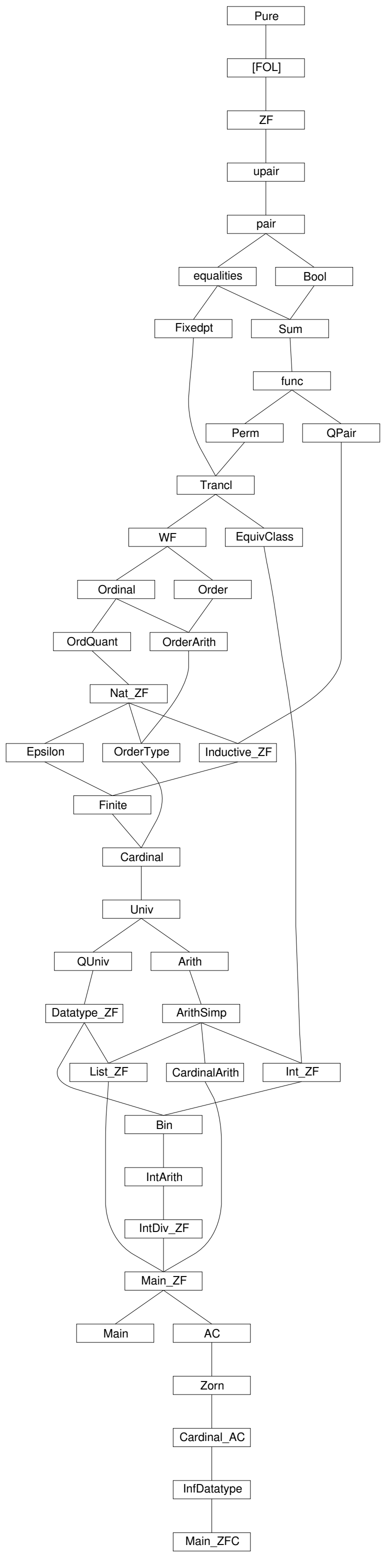
ZF

Lawrence C Paulson and others

December 3, 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 ⟪ *Unsynchronized.reset eta-contract* ⟫

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*] => *i*
fst :: *i* => *i*
snd :: *i* => *i*
split :: [[*i*, *i*] => '*a*, *i*] => '*a*::{} — for pattern-matching

Sigma and Pi Operators

consts
Sigma :: [*i*, *i* => *i*] => *i*
Pi :: [*i*, *i* => *i*] => *i*

Relations and Functions

consts
domain :: *i* => *i*
range :: *i* => *i*
field :: *i* => *i*
converse :: *i* => *i*
relation :: *i* => *o* — recognizes sets of pairs
function :: *i* => *o* — recognizes functions; can have non-pairs
Lambda :: [*i*, *i* => *i*] => *i*
restrict :: [*i*, *i*] => *i*

Infixes in order of decreasing precedence

consts

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

abbreviation
not-mem :: [*i*, *i*] => *o* (**infixl** ~: *50*) — negated membership relation
where *x* ~: *y* == ~ (*x* : *y*)

abbreviation
cart-prod :: [*i*, *i*] => *i* (**infixr** * *80*) — Cartesian product
where *A* * *B* == *Sigma*(*A*, %-. *B*)

abbreviation
function-space :: [*i*, *i*] => *i* (**infixr** -> *60*) — function space
where *A* -> *B* == *Pi*(*A*, %-. *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** → *60*) **and**
Subset (**infixl** ⊆ *50*) **and**
mem (**infixl** ∈ *50*) **and**
not-mem (**infixl** ∉ *50*) **and**
Union (⋃- [*90*] *90*) **and**
Inter (⋂- [*90*] *90*)

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

notation (*HTML* **output**)
cart-prod (**infixr** × *80*) **and**
Int (**infixl** ∩ *70*) **and**
Un (**infixl** ∪ *65*) **and**
Subset (**infixl** ⊆ *50*) **and**
mem (**infixl** ∈ *50*) **and**
not-mem (**infixl** ∉ *50*) **and**
Union (⋃- [*90*] *90*) **and**
Inter (⋂- [*90*] *90*)

syntax (*HTML* **output**)
@*Collect* :: [*pttrn*, *i*, *o*] => *i* ((*1*{- ∈ - ./ -}))
@*Replace* :: [*pttrn*, *pttrn*, *i*, *o*] => *i* ((*1*{- ./ - ∈ -, -}))
@*RepFun* :: [*i*, *pttrn*, *i*] => *i* ((*1*{- ./ - ∈ -}) [*51*,*0*,*51*])
@*UNION* :: [*pttrn*, *i*, *i*] => *i* ((*3*⋃-∈-./ -) *10*)
@*INTER* :: [*pttrn*, *i*, *i*] => *i* ((*3*⋂-∈-./ -) *10*)
@*PROD* :: [*pttrn*, *i*, *i*] => *i* ((*3*Π-∈-./ -) *10*)
@*SUM* :: [*pttrn*, *i*, *i*] => *i* ((*3*Σ-∈-./ -) *10*)
@*lam* :: [*pttrn*, *i*, *i*] => *i* ((*3*λ-∈-./ -) *10*)
@*Ball* :: [*pttrn*, *i*, *o*] => *o* ((*3*∀-∈-./ -) *10*)
@*Bex* :: [*pttrn*, *i*, *o*] => *o* ((*3*∃-∈-./ -) *10*)
@*Tuple* :: [*i*, *is*] => *i* (⟨(-,/ -)⟩)
@*pattern* :: *patterns* => *pttrn* (⟨-⟩)

finalconsts
0 Pow Inf Union PrimReplace mem

defs

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

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

local

axioms

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

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

$foundation$: $A{=}0 \mid (\exists x{\in}A. \forall y{\in}x. y\tilde{}{:}A)$

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

defs

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

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

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

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

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: $if(P,a,b) == THE\ z.\ P\ \&\ z{=}a\ |\ {\sim}P\ \&\ z{=}b$

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: $function(r) ==$
$\forall x\ y.\ <x,y>{:}r\ {-}{-}{>}\ (\forall y'.\ <x,y'>{:}r\ {-}{-}{>}\ y{=}y')$
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*: $[|\ b{\in}A;\ \ a{=}b\ |] ==> a{\in}A$
by (*erule ssubst*, *assumption*)

1.2 Bounded universal quantifier

lemma *ballI* [*intro*!]: [| !!x. x∈A ==> P(x) |] ==> ∀x∈A. P(x)
by (*simp add*: *Ball-def*)

lemmas *strip* = *impI allI ballI*

lemma *bspec* [*dest?*]: [| ∀x∈A. P(x); x: A |] ==> P(x)
by (*simp add*: *Ball-def*)

lemma *rev-ballE* [*elim*]:
  [| ∀x∈A. P(x); x~:A ==> Q; P(x) ==> Q |] ==> Q
by (*simp add*: *Ball-def*, *blast*)

lemma *ballE*: [| ∀x∈A. P(x); P(x) ==> Q; x~:A ==> Q |] ==> Q
by *blast*

lemma *rev-bspec*: [| x: A; ∀x∈A. P(x) |] ==> P(x)
by (*simp add*: *Ball-def*)

lemma *ball-triv* [*simp*]: (∀x∈A. P) <−> ((∃x. x∈A) −−> P)
by (*simp add*: *Ball-def*)

lemma *ball-cong* [*cong*]:
  [| A=A′; !!x. x∈A′ ==> P(x) <−> P′(x) |] ==> (∀x∈A. P(x)) <−> (∀x∈A′. P′(x))
by (*simp add*: *Ball-def*)

lemma *atomize-ball*:
  (!!x. x ∈ A ==> P(x)) == *Trueprop* (∀x∈A. P(x))
 by (*simp only*: *Ball-def atomize-all atomize-imp*)

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

lemma *rev-bexI*: [| x∈A; P(x) |] ==> ∃x∈A. P(x)
by *blast*

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

by *blast*

lemma *bexE* [*elim*!]: [| ∃ *x*∈*A*. *P*(*x*); !!*x*. [| *x*∈*A*; *P*(*x*) |] ==> *Q* |] ==> *Q*
by (*simp add*: *Bex-def*, *blast*)

lemma *bex-triv* [*simp*]: (∃ *x*∈*A*. *P*) <−> ((∃ *x*. *x*∈*A*) & *P*)
by (*simp add*: *Bex-def*)

lemma *bex-cong* [*cong*]:
  [| *A*=*A*′; !!*x*. *x*∈*A*′ ==> *P*(*x*) <−> *P*′(*x*) |]
  ==> (∃ *x*∈*A*. *P*(*x*)) <−> (∃ *x*∈*A*′. *P*′(*x*))
by (*simp add*: *Bex-def cong*: *conj-cong*)

1.4 Rules for subsets

lemma *subsetI* [*intro*!]:
  (!!*x*. *x*∈*A* ==> *x*∈*B*) ==> *A* <= *B*
by (*simp add*: *subset-def*)

lemma *subsetD* [*elim*]: [| *A* <= *B*; *c*∈*A* |] ==> *c*∈*B*
apply (*unfold subset-def*)
apply (*erule bspec*, *assumption*)
done

lemma *subsetCE* [*elim*]:
  [| *A* <= *B*; *c*~:*A* ==> *P*; *c*∈*B* ==> *P* |] ==> *P*
by (*simp add*: *subset-def*, *blast*)

lemma *rev-subsetD*: [| *c*∈*A*; *A*<=*B* |] ==> *c*∈*B*
by *blast*

lemma *contra-subsetD*: [| *A* <= *B*; *c* ~: *B* |] ==> *c* ~: *A*
by *blast*

lemma *rev-contra-subsetD*: [| *c* ~: *B*; *A* <= *B* |] ==> *c* ~: *A*
by *blast*

lemma *subset-refl* [*simp*]: *A* <= *A*
by *blast*

lemma *subset-trans*: [| *A*<=*B*; *B*<=*C* |] ==> *A*<=*C*
by *blast*

lemma *subset-iff*:

A<=*B* <-> (∀*x*. *x*∈*A* --> *x*∈*B*)
apply (*unfold subset-def Ball-def*)
apply (*rule iff-refl*)
done

1.5 Rules for equality

lemma *equalityI* [*intro*]: [| *A* <= *B*; *B* <= *A* |] ==> *A* = *B*
by (*rule extension* [*THEN iffD2*], *rule conjI*)

lemma *equality-iffI*: (!!*x*. *x*∈*A* <-> *x*∈*B*) ==> *A* = *B*
by (*rule equalityI*, *blast*+)

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*
by (*blast dest*: *equalityD1 equalityD2*)

lemma *equalityCE*:
[| *A* = *B*; [| *c*∈*A*; *c*∈*B* |] ==> *P*; [| *c*~:*A*; *c*~:*B* |] ==> *P* |] ==> *P*
by (*erule equalityE*, *blast*)

lemma *equality-iffD*:
A = *B* ==> (!!*x*. *x* : *A* <-> *x* : *B*)
by *auto*

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*))
apply (*unfold Replace-def*)
apply (*rule replacement* [*THEN iff-trans*], *blast*+)
done

lemma *ReplaceI* [*intro*]:
[| *P*(*x*,*b*); *x*: *A*; !!*y*. *P*(*x*,*y*) ==> *y*=*b* |] ==>
b : {*y*. *x*∈*A*, *P*(*x*,*y*)}
by (*rule Replace-iff* [*THEN iffD2*], *blast*)

lemma *ReplaceE*:
[| *b* : {*y*. *x*∈*A*, *P*(*x*,*y*)};
!!*x*. [| *x*: *A*; *P*(*x*,*b*); ∀*y*. *P*(*x*,*y*)-->*y*=*b* |] ==> *R*
|] ==> *R*
by (*rule Replace-iff* [*THEN iffD1*, *THEN bexE*], *simp*+)

```
lemma ReplaceE2 [elim!]:
    [| b : {y. x∈A, P(x,y)};
        !!x. [| x: A;  P(x,b) |] ==> R
     |] ==> R
by (erule ReplaceE, blast)

lemma Replace-cong [cong]:
    [| A=B;  !!x y. x∈B ==> P(x,y) <-> Q(x,y) |] ==>
     Replace(A,P) = Replace(B,Q)
apply (rule equality-iffI)
apply (simp add: Replace-iff)
done
```

1.7 Rules for RepFun

```
lemma RepFunI: a ∈ A ==> f(a) : {f(x). x∈A}
by (simp add: RepFun-def Replace-iff, blast)


lemma RepFun-eqI [intro]: [| b=f(a);  a ∈ A |] ==> b : {f(x). x∈A}
apply (erule ssubst)
apply (erule RepFunI)
done

lemma RepFunE [elim!]:
    [| b : {f(x). x∈A};
        !!x.[| x∈A;  b=f(x) |] ==> P |] ==>
     P
by (simp add: RepFun-def Replace-iff, blast)

lemma RepFun-cong [cong]:
    [| A=B;  !!x. x∈B ==> f(x)=g(x) |] ==> RepFun(A,f) = RepFun(B,g)
by (simp add: RepFun-def)

lemma RepFun-iff [simp]: b : {f(x). x∈A} <-> (∃x∈A. b=f(x))
by (unfold Bex-def, blast)

lemma triv-RepFun [simp]: {x. x∈A} = A
by blast
```

1.8 Rules for Collect – forming a subset by separation

```
lemma separation [simp]: a : {x∈A. P(x)} <-> a∈A & P(a)
by (unfold Collect-def, blast)

lemma CollectI [intro!]: [| a∈A;  P(a) |] ==> a : {x∈A. P(x)}
by simp

lemma CollectE [elim!]: [| a : {x∈A. P(x)};  [| a∈A; P(a) |] ==> R |] ==> R
by simp
```

lemma *CollectD1*: a : {x∈A. P(x)} ==> a∈A
by (*erule CollectE*, *assumption*)

lemma *CollectD2*: a : {x∈A. P(x)} ==> P(a)
by (*erule CollectE*, *assumption*)

lemma *Collect-cong* [*cong*]:
  [| A=B; !!x. x∈B ==> P(x) <-> Q(x) |]
  ==> Collect(A, %x. P(x)) = Collect(B, %x. Q(x))
by (*simp add*: *Collect-def*)

1.9 Rules for Unions

declare *Union-iff* [*simp*]

lemma *UnionI* [*intro*]: [| B: C; A: B |] ==> A: Union(C)
by (*simp*, *blast*)

lemma *UnionE* [*elim*!]: [| A ∈ Union(C); !!B.[| A: B; B: C |] ==> R |] ==> R
by (*simp*, *blast*)

1.10 Rules for Unions of families

lemma *UN-iff* [*simp*]: b : (⋃x∈A. B(x)) <-> (∃x∈A. b ∈ B(x))
by (*simp add*: *Bex-def*, *blast*)

lemma *UN-I*: [| a: A; b: B(a) |] ==> b: (⋃x∈A. B(x))
by (*simp*, *blast*)

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

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

1.11 Rules for the empty set

lemma *not-mem-empty* [*simp*]: a ~: *0*
apply (*cut-tac foundation*)
apply (*best dest*: *equalityD2*)
done

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

lemma *empty-subsetI* [*simp*]: *0* <= *A*
by *blast*

lemma *equals0I*: [| !!*y*. *y*∈*A* ==> *False* |] ==> *A*=*0*
by *blast*

lemma *equals0D* [*dest*]: *A*=*0* ==> *a* ~: *A*
by *blast*

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

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

lemma *not-emptyE*: [| *A* ~= *0*; !!*x*. *x*∈*A* ==> *R* |] ==> *R*
by *blast*

1.12 Rules for Inter

lemma *Inter-iff*: *A* ∈ *Inter*(*C*) <-> (∀*x*∈*C*. *A*: *x*) & *C*≠*0*
by (*simp add*: *Inter-def Ball-def*, *blast*)

lemma *InterI* [*intro*!]:
    [| !!*x*. *x*: *C* ==> *A*: *x*; *C*≠*0* |] ==> *A* ∈ *Inter*(*C*)
by (*simp add*: *Inter-iff*)

lemma *InterD* [*elim*]: [| *A* ∈ *Inter*(*C*); *B* ∈ *C* |] ==> *A* ∈ *B*
by (*unfold Inter-def*, *blast*)

lemma *InterE* [*elim*]:
    [| *A* ∈ *Inter*(*C*); *B*~:*C* ==> *R*; *A*∈*B* ==> *R* |] ==> *R*
by (*simp add*: *Inter-def*, *blast*)

1.13 Rules for Intersections of families

lemma *INT-iff*: *b* : (⋂*x*∈*A*. *B*(*x*)) <-> (∀*x*∈*A*. *b* ∈ *B*(*x*)) & *A*≠*0*
by (*force simp add*: *Inter-def*)

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

lemma *INT-E*: [| *b* : (⋂*x*∈*A*. *B*(*x*)); *a*: *A* |] ==> *b* ∈ *B*(*a*)
by *blast*

lemma *INT-cong*:

[| A=B; !!x. x∈B ==> C(x)=D(x) |] ==> (⋂x∈A. C(x)) = (⋂x∈B. D(x))
by *simp*

1.14 Rules for Powersets

lemma *PowI*: *A <= B ==> A ∈ Pow(B)*
by (*erule Pow-iff* [*THEN iffD2*])

lemma *PowD*: *A ∈ Pow(B) ==> A<=B*
by (*erule Pow-iff* [*THEN iffD1*])

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*: ∃ *S* ∈ *Pow*(*A*). ∀ *x*∈*A*. *b*(*x*) ~= *S*
by (*best elim*!: *equalityCE del*: *ReplaceI RepFun-eqI*)

ML
⟪
(∗*Converts A<=B to x∈A ==> x∈B*∗)
fun impOfSubs th = *th RSN* (*2*, @{*thm rev-subsetD*});

(∗*Takes assumptions* ∀ *x*∈*A*.*P*(*x*) *and a*∈*A*; *creates assumption P*(*a*)∗)
val ball-tac = *dtac* @{*thm bspec*} *THEN′ assume-tac*
⟫

end

2 upair: Unordered Pairs

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

setup *TypeCheck.setup*

lemma *atomize-ball* [*symmetric*, *rulify*]:
(!!*x*. *x*:*A* ==> *P*(*x*)) == *Trueprop* (*ALL x*:*A*. *P*(*x*))
by (*simp add*: *Ball-def atomize-all atomize-imp*)

2.1 Unordered Pairs: constant *Upair*

lemma *Upair-iff* [*simp*]: *c* : *Upair*(*a*,*b*) <−> (*c*=*a* | *c*=*b*)
by (*unfold Upair-def*, *blast*)

lemma *UpairI1*: *a* : *Upair*(*a*,*b*)
by *simp*

lemma *UpairI2*: *b* : *Upair*(*a*,*b*)
by *simp*

lemma *UpairE*: [| *a* : *Upair*(*b*,*c*); *a*=*b* ==> *P*; *a*=*c* ==> *P* |] ==> *P*
by (*simp*, *blast*)

2.2 Rules for Binary Union, Defined via *Upair*

lemma *Un-iff* [*simp*]: *c* : *A Un B* <−> (*c*:*A* | *c*:*B*)
apply (*simp add*: *Un-def*)
apply (*blast intro*: *UpairI1 UpairI2 elim*: *UpairE*)
done

lemma *UnI1*: *c* : *A* ==> *c* : *A Un B*
by *simp*

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

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

lemma *UnE* [*elim*!]: [| *c* : *A Un B*; *c*:*A* ==> *P*; *c*:*B* ==> *P* |] ==> *P*
by (*simp*, *blast*)

lemma *UnE'*: [| *c* : *A Un B*; *c*:*A* ==> *P*; [| *c*:*B*; *c*~:*A* |] ==> *P* |] ==> *P*
by (*simp*, *blast*)

lemma *UnCI* [*intro*!]: (*c* ~: *B* ==> *c* : *A*) ==> *c* : *A Un B*
by (*simp*, *blast*)

2.3 Rules for Binary Intersection, Defined via *Upair*

lemma *Int-iff* [*simp*]: *c* : *A Int B* <−> (*c*:*A* & *c*:*B*)
apply (*unfold Int-def*)
apply (*blast intro*: *UpairI1 UpairI2 elim*: *UpairE*)
done

lemma *IntI* [*intro*!]: [| *c* : *A*; *c* : *B* |] ==> *c* : *A Int B*
by *simp*

lemma *IntD1*: *c* : *A* *Int* *B* ==> *c* : *A*
by *simp*

lemma *IntD2*: *c* : *A* *Int* *B* ==> *c* : *B*
by *simp*

lemma *IntE* [*elim*!]: [| *c* : *A* *Int* *B*; [| *c*:*A*; *c*:*B* |] ==> *P* |] ==> *P*
by *simp*

2.4 Rules for Set Difference, Defined via *Upair*

lemma *Diff-iff* [*simp*]: *c* : *A*−*B* <−> (*c*:*A* & *c*~:*B*)
by (*unfold Diff-def*, *blast*)

lemma *DiffI* [*intro*!]: [| *c* : *A*; *c* ~: *B* |] ==> *c* : *A* − *B*
by *simp*

lemma *DiffD1*: *c* : *A* − *B* ==> *c* : *A*
by *simp*

lemma *DiffD2*: *c* : *A* − *B* ==> *c* ~: *B*
by *simp*

lemma *DiffE* [*elim*!]: [| *c* : *A* − *B*; [| *c*:*A*; *c*~:*B* |] ==> *P* |] ==> *P*
by *simp*

2.5 Rules for *cons*

lemma *cons-iff* [*simp*]: *a* : *cons*(*b*,*A*) <−> (*a*=*b* | *a*:*A*)
apply (*unfold cons-def*)
apply (*blast intro*: *UpairI1 UpairI2 elim*: *UpairE*)
done

lemma *consI1* [*simp*,*TC*]: *a* : *cons*(*a*,*B*)
by *simp*

lemma *consI2*: *a* : *B* ==> *a* : *cons*(*b*,*B*)
by *simp*

lemma *consE* [*elim*!]: [| *a* : *cons*(*b*,*A*); *a*=*b* ==> *P*; *a*:*A* ==> *P* |] ==> *P*
by (*simp*, *blast*)

lemma *consE'*:
[| *a* : *cons*(*b*,*A*); *a*=*b* ==> *P*; [| *a*:*A*; *a*~=*b* |] ==> *P* |] ==> *P*
by (*simp*, *blast*)

lemma *consCI* [*intro*!]: (*a*~:*B* ==> *a*=*b*) ==> *a*: *cons*(*b*,*B*)
by (*simp*, *blast*)

lemma *cons-not-0* [*simp*]: *cons*(*a*,*B*) ~= *0*
by (*blast elim*: *equalityE*)

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*
by *simp*

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

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

2.7 Descriptions

lemma *the-equality* [*intro*]:
    [| *P*(*a*); !!*x*. *P*(*x*) ==> *x*=*a* |] ==> (*THE x*. *P*(*x*)) = *a*
apply (*unfold the-def*)
apply (*fast dest*: *subst*)
done

lemma *the-equality2*: [| *EX*! *x*. *P*(*x*); *P*(*a*) |] ==> (*THE x*. *P*(*x*)) = *a*
by *blast*

lemma *theI*: *EX*! *x*. *P*(*x*) ==> *P*(*THE x*. *P*(*x*))
apply (*erule ex1E*)
apply (*subst the-equality*)
apply (*blast*+)
done

lemma *the-0*: ~ (*EX*! *x*. *P*(*x*)) ==> (*THE x*. *P*(*x*))=*0*
apply (*unfold the-def*)
apply (*blast elim*!: *ReplaceE*)
done

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

  shows *Q*(*THE x*. *P*(*x*))
apply (*rule classical*)
apply (*rule p2*)
apply (*rule theI*)
apply (*rule classical*)
apply (*rule p1*)
apply (*erule the-0* [*THEN subst*], *assumption*)
done

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

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

2.8 Conditional Terms: *if−then−else*

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

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

lemma *if-cong*:
  [| *P*<->*Q*; *Q* ==> *a*=*c*; ~*Q* ==> *b*=*d* |]
  ==> (*if P then a else b*) = (*if Q then c else d*)
by (*simp add*: *if-def cong add*: *conj-cong*)

lemma *if-weak-cong*: *P*<->*Q* ==> (*if P then x else y*) = (*if Q then x else y*)
by *simp*

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

lemma *if-not-P*: ~*P* ==> (*if P then a else b*) = *b*
by (*unfold if-def*, *blast*)

lemma *split-if* [*split*]:
  P(*if Q then x else y*) <-> ((*Q* --> *P*(*x*)) & (~*Q* --> *P*(*y*)))
by (*case-tac Q*, *simp-all*)

lemmas *split-if-eq1* = *split-if* [*of* %*x*. *x* = *b*, *standard*]
lemmas *split-if-eq2* = *split-if* [*of* %*x*. *a* = *x*, *standard*]

lemmas split-if-mem1 = split-if [of %x. x : b, standard]
lemmas split-if-mem2 = split-if [of %x. a : x, standard]

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

lemma if-iff: a: (if P then x else y) <-> P & a:x | ~P & a:y
by simp

lemma if-type [TC]:
  [| P ==> a: A; ~P ==> b: A |] ==> (if P then a else b): A
by simp

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

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

2.9 Consequences of Foundation

lemma mem-asym: [| a:b; ~P ==> b:a |] ==> P
apply (rule classical)
apply (rule-tac A1 = {a,b} **in** foundation [THEN disjE])
apply (blast elim!: equalityE)+
done

lemma mem-irrefl: a:a ==> P
by (blast intro: mem-asym)

lemma mem-not-refl: a ~: a
apply (rule notI)
apply (erule mem-irrefl)
done

lemma mem-imp-not-eq: a:A ==> a ~= A
by (blast elim!: mem-irrefl)

lemma eq-imp-not-mem: a=A ==> a ~: A
by (blast intro: elim: mem-irrefl)

2.10 Rules for Successor

lemma succ-iff: i : succ(j) <-> i=j | i:j

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

lemma *succI1* [*simp*]: *i* : *succ*(*i*)
by (*simp add*: *succ-iff*)

lemma *succI2*: *i* : *j* ==> *i* : *succ*(*j*)
by (*simp add*: *succ-iff*)

lemma *succE* [*elim*!]:
  [| *i* : *succ*(*j*); *i*=*j* ==> *P*; *i*:*j* ==> *P* |] ==> *P*
apply (*simp add*: *succ-iff*, *blast*)
done

lemma *succCI* [*intro*!]: (*i*~:*j* ==> *i*=*j*) ==> *i*: *succ*(*j*)
by (*simp add*: *succ-iff*, *blast*)

lemma *succ-not-0* [*simp*]: *succ*(*n*) ~= *0*
by (*blast elim*!: *equalityE*)

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*
by (*blast elim*: *mem-asym elim*!: *equalityE*)

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*) <−> (*ALL x*:*A*. *P*(*x*)) & (*A*=*0* | *Q*)
  (*ALL x*:*A*. *P*(*x*) | *Q*) <−> ((*ALL x*:*A*. *P*(*x*)) | *Q*)
  (*ALL x*:*A*. *P*(*x*) −−> *Q*) <−> ((*EX x*:*A*. *P*(*x*)) −−> *Q*)
  (~(*ALL x*:*A*. *P*(*x*))) <−> (*EX x*:*A*. ~*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*))

by *blast+*

lemma *ball-simps2*:
(ALL x:A. P & Q(x)) <-> (A=0 | P) & (ALL x:A. Q(x))
(ALL x:A. P | Q(x)) <-> (P | (ALL x:A. Q(x)))
(ALL x:A. P --> Q(x)) <-> (P --> (ALL x:A. Q(x)))
by *blast+*

lemma *ball-simps3*:
(ALL x:Collect(A,Q).P(x)) <-> (ALL x:A. Q(x) --> P(x))
by *blast+*

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)))
by *blast*

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) | Q) <-> (EX x:A. P(x)) | (A~=0 & Q)
(EX x:A. P(x) --> Q) <-> ((ALL x:A. P(x)) --> (A~=0 & Q))
(EX x:0.P(x)) <-> False
(EX x:succ(i).P(x)) <-> P(i) | (EX x:i. P(x))
(EX x:cons(a,B).P(x)) <-> P(a) | (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))
(~(EX x:A. P(x))) <-> (ALL x:A. ~P(x))
by *blast+*

lemma *bex-simps2*:
(EX x:A. P & Q(x)) <-> (P & (EX x:A. Q(x)))
(EX x:A. P | Q(x)) <-> (A~=0 & P) | (EX x:A. Q(x))
(EX x:A. P --> Q(x)) <-> ((A=0 | P) --> (EX x:A. Q(x)))
by *blast+*

lemma *bex-simps3*:
(EX x:Collect(A,Q).P(x)) <-> (EX x:A. Q(x) & P(x))
by *blast*

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

lemma *bex-disj-distrib*:
(EX x:A. P(x) | Q(x)) <-> ((EX x:A. P(x)) | (EX x:A. Q(x)))
by *blast*

lemma *bex-triv-one-point1* [*simp*]: (*EX x*:*A*. *x*=*a*) <-> (*a*:*A*)
by *blast*

lemma *bex-triv-one-point2* [*simp*]: (*EX x*:*A*. *a*=*x*) <-> (*a*:*A*)
by *blast*

lemma *bex-one-point1* [*simp*]: (*EX x*:*A*. *x*=*a* & *P*(*x*)) <-> (*a*:*A* & *P*(*a*))
by *blast*

lemma *bex-one-point2* [*simp*]: (*EX x*:*A*. *a*=*x* & *P*(*x*)) <-> (*a*:*A* & *P*(*a*))
by *blast*

lemma *ball-one-point1* [*simp*]: (*ALL x*:*A*. *x*=*a* --> *P*(*x*)) <-> (*a*:*A* --> *P*(*a*))
by *blast*

lemma *ball-one-point2* [*simp*]: (*ALL x*:*A*. *a*=*x* --> *P*(*x*)) <-> (*a*:*A* --> *P*(*a*))
by *blast*

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*))
by (*simp-all*, *blast*+)

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

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*)) = (*UN a*:*A*. *B*(*f*(*a*)))
by *blast*+

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

Opposite of miniscoping: pull the operator out

lemma *UN-extend-simps1*:
(*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*)
apply *simp-all*
apply *blast*+
done

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

lemma *UN-UN-extend*:
(*UN x*:*A*. *UN z*: *B*(*x*). *C*(*z*)) = (*UN z*: (*UN x*:*A*. *B*(*x*)). *C*(*z*))
by *blast*

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

2.15 Miniscoping of Intersections

lemma *INT-simps1*:
(*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*)
by (*simp-all add*: *Inter-def*, *blast*+)

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

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

Opposite of miniscoping: pull the operator out

lemma *INT-extend-simps1*:
  (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))
apply (*simp-all add*: *Inter-def*, *blast*+)
done

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

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

2.16 Other simprules

lemma *misc-simps* [*simp*]:
  0 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
by *blast*+

end

3 pair: Ordered Pairs

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

lemma *singleton-eq-iff* [*iff*]: *{a} = {b} <−> a=b*
by (*rule extension* [*THEN iff-trans*], *blast*)

lemma *doubleton-eq-iff*: *{a,b} = {c,d} <−> (a=c & b=d) | (a=d & b=c)*
by (*rule extension* [*THEN iff-trans*], *blast*)

lemma *Pair-iff* [*simp*]: <*a*,*b*> = <*c*,*d*> <−> *a*=*c* & *b*=*d*
by (*simp add*: *Pair-def doubleton-eq-iff*, *blast*)

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*> ~= *0*
apply (*unfold Pair-def*)
apply (*blast elim*: *equalityE*)
done

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* ==> *P*
apply (*unfold Pair-def*)
apply (*rule consI1* [*THEN mem-asym*, *THEN FalseE*])
apply (*erule subst*)
apply (*rule consI1*)
done

lemma *Pair-neq-snd*: <*a*,*b*>=*b* ==> *P*
apply (*unfold Pair-def*)
apply (*rule consI1* [*THEN consI2*, *THEN mem-asym*, *THEN FalseE*])
apply (*erule subst*)
apply (*rule consI1* [*THEN consI2*])
done

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

lemma *SigmaI* [*TC*,*intro*!]: [| *a*:*A*; *b*:*B*(*a*) |] ==> <*a*,*b*> : *Sigma*(*A*,*B*)
by *simp*

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*);
!!*x y*.[| *x*:*A*; *y*:*B*(*x*); *c*=<*x*,*y*> |] ==> *P*
|] ==> *P*
by (*unfold Sigma-def*, *blast*)

lemma *SigmaE2* [*elim*!]:
[| <*a*,*b*> : *Sigma*(*A*,*B*);
[| *a*:*A*; *b*:*B*(*a*) |] ==> *P*
|] ==> *P*
by (*unfold Sigma-def*, *blast*)

lemma *Sigma-cong*:
[| *A*=*A*'; !!*x*. *x*:*A*' ==> *B*(*x*)=*B*'(*x*) |] ==>
Sigma(*A*,*B*) = *Sigma*(*A*',*B*')
by (*simp add*: *Sigma-def*)

lemma *Sigma-empty1* [*simp*]: *Sigma*(*0*,*B*) = *0*
by *blast*

lemma *Sigma-empty2* [*simp*]: *A*∗*0* = *0*
by *blast*

lemma *Sigma-empty-iff*: *A*∗*B*=*0* <−> *A*=*0* | *B*=*0*
by *blast*

3.2 Projections *fst* and *snd*

lemma *fst-conv* [*simp*]: *fst*(<*a*,*b*>) = *a*
by (*simp add*: *fst-def*)

lemma *snd-conv* [*simp*]: *snd*(<*a*,*b*>) = *b*
by (*simp add*: *snd-def*)

lemma *fst-type* [*TC*]: *p*:*Sigma*(*A*,*B*) ==> *fst*(*p*) : *A*
by *auto*

lemma *snd-type* [*TC*]: *p*:*Sigma*(*A*,*B*) ==> *snd*(*p*) : *B*(*fst*(*p*))
by *auto*

lemma *Pair-fst-snd-eq*: *a*: *Sigma*(*A*,*B*) ==> <*fst*(*a*),*snd*(*a*)> = *a*
by *auto*

3.3 The Eliminator, *split*

lemma *split* [*simp*]: *split*(%*x y*. *c*(*x*,*y*), <*a*,*b*>) == *c*(*a*,*b*)
by (*simp add*: *split-def*)

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*)
apply (*erule SigmaE*, *auto*)

```
done

lemma expand-split:
  u: A∗B ==>
       R(split(c,u)) <−> (ALL x:A. ALL y:B. u = <x,y> −−> R(c(x,y)))
apply (simp add: split-def)
apply auto
done
```

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

```
lemma splitI: R(a,b) ==> split(R, <a,b>)
by (simp add: split-def)

lemma splitE:
    [| split(R,z);  z:Sigma(A,B);
        !!x y. [| z = <x,y>;  R(x,y) |] ==> P
     |] ==> P
apply (simp add: split-def)
apply (erule SigmaE, force)
done

lemma splitD: split(R,<a,b>) ==> R(a,b)
by (simp add: split-def)
```

Complex rules for Sigma.

```
lemma split-paired-Bex-Sigma [simp]:
     (∃ z ∈ Sigma(A,B). P(z)) <−> (∃ x ∈ A. ∃ y ∈ B(x). P(<x,y>))
by blast

lemma split-paired-Ball-Sigma [simp]:
     (∀ z ∈ Sigma(A,B). P(z)) <−> (∀ x ∈ A. ∀ y ∈ B(x). P(<x,y>))
by blast

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
by blast
```

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

4.1 Bounded Quantifiers

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

lemma *ball-Un*: (∀ *x* ∈ *A*∪*B*. *P*(*x*)) <−> (∀ *x* ∈ *A*. *P*(*x*)) & (∀ *x* ∈ *B*. *P*(*x*))
by *blast*

lemma *bex-Un*: (∃ *x* ∈ *A*∪*B*. *P*(*x*)) <−> (∃ *x* ∈ *A*. *P*(*x*)) | (∃ *x* ∈ *B*. *P*(*x*))
by *blast*

lemma *ball-UN*: (∀ *z* ∈ (⋃*x*∈*A*. *B*(*x*)). *P*(*z*)) <−> (∀ *x*∈*A*. ∀ *z* ∈ *B*(*x*). *P*(*z*))
by *blast*

lemma *bex-UN*: (∃ *z* ∈ (⋃*x*∈*A*. *B*(*x*)). *P*(*z*)) <−> (∃ *x*∈*A*. ∃ *z*∈*B*(*x*). *P*(*z*))
by *blast*

4.2 Converse of a Relation

lemma *converse-iff* [*simp*]: <*a*,*b*>∈ *converse*(*r*) <−> <*b*,*a*>∈*r*
by (*unfold converse-def*, *blast*)

lemma *converseI* [*intro*!]: <*a*,*b*>∈*r* ==> <*b*,*a*>∈*converse*(*r*)
by (*unfold converse-def*, *blast*)

lemma *converseD*: <*a*,*b*> ∈ *converse*(*r*) ==> <*b*,*a*> ∈ *r*
by (*unfold converse-def*, *blast*)

lemma *converseE* [*elim*!]:
[| *yx* ∈ *converse*(*r*);
!!*x y*. [| *yx*=<*y*,*x*>; <*x*,*y*>∈*r* |] ==> *P* |]
==> *P*
by (*unfold converse-def*, *blast*)

lemma *converse-converse*: *r*⊆*Sigma*(*A*,*B*) ==> *converse*(*converse*(*r*)) = *r*
by *blast*

lemma *converse-type*: *r*⊆*A*∗*B* ==> *converse*(*r*)⊆*B*∗*A*
by *blast*

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

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

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

4.3 Finite Set Constructions Using *cons*

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

lemma *subset-consI*: $B \subseteq cons(a,B)$
by *blast*

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

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

lemma *subset-empty-iff*: $A\subseteq 0 <-> A=0$
by *blast*

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

lemma *cons-eq*: $\{a\}$ *Un* $B = cons(a,B)$
by *blast*

lemma *cons-commute*: $cons(a, cons(b, C)) = cons(b, cons(a, C))$
by *blast*

lemma *cons-absorb*: $a: B ==> cons(a,B) = B$
by *blast*

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

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

lemma *equal-singleton* [*rule-format*]: $[| a: C; \forall y\in C. y=b |] ==> C = \{b\}$
by *blast*

lemma [*simp*]: $cons(a,cons(a,B)) = cons(a,B)$
by *blast*

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

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

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

lemma *succ-subsetI*: [| $i{\in}j$; $i{\subseteq}j$ |] ==> $succ(i){\subseteq}j$
by (*unfold succ-def*, *blast*)

lemma *succ-subsetE*:
  [| $succ(i) \subseteq j$; [| $i{\in}j$; $i{\subseteq}j$ |] ==> P |] ==> P
by (*unfold succ-def*, *blast*)

lemma *succ-subset-iff*: $succ(a) \subseteq B$ <-> ($a \subseteq B$ & $a \in B$)
by (*unfold succ-def*, *blast*)

4.4 Binary Intersection

lemma *Int-subset-iff*: $C \subseteq A$ *Int* B <-> $C \subseteq A$ & $C \subseteq B$
by *blast*

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

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

lemma *Int-greatest*: [| $C{\subseteq}A$; $C{\subseteq}B$ |] ==> $C \subseteq A$ *Int* B
by *blast*

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

lemma *Int-absorb* [*simp*]: A *Int* $A = A$
by *blast*

lemma *Int-left-absorb*: A *Int* (A *Int* B) = A *Int* B
by *blast*

lemma *Int-commute*: A *Int* B = B *Int* A
by *blast*

lemma *Int-left-commute*: A Int (B Int C) = B Int (A Int C)
by *blast*

lemma *Int-assoc*: (A Int B) Int C = A Int (B Int C)
by *blast*

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

lemma *Int-absorb1*: B ⊆ A ==> A ∩ B = B
 by *blast*

lemma *Int-absorb2*: A ⊆ B ==> A ∩ B = A
 by *blast*

lemma *Int-Un-distrib*: A Int (B Un C) = (A Int B) Un (A Int C)
by *blast*

lemma *Int-Un-distrib2*: (B Un C) Int A = (B Int A) Un (C Int A)
by *blast*

lemma *subset-Int-iff*: A⊆B <-> A Int B = A
by (*blast elim*!: *equalityE*)

lemma *subset-Int-iff2*: A⊆B <-> B Int A = A
by (*blast elim*!: *equalityE*)

lemma *Int-Diff-eq*: C⊆A ==> (A−B) Int C = C−B
by *blast*

lemma *Int-cons-left*:
 cons(a,A) Int B = (if a ∈ B then cons(a, A Int B) else A Int B)
by *auto*

lemma *Int-cons-right*:
 A Int cons(a, B) = (if a ∈ A then cons(a, A Int B) else A Int B)
by *auto*

lemma *cons-Int-distrib*: cons(x, A ∩ B) = cons(x, A) ∩ cons(x, B)
by *auto*

4.5 Binary Union

lemma *Un-subset-iff*: A Un B ⊆ C <-> A ⊆ C & B ⊆ C
by *blast*

lemma *Un-upper1*: A ⊆ A Un B
by *blast*

lemma *Un-upper2*: B ⊆ A *Un* B
by *blast*

lemma *Un-least*: [| A⊆C; B⊆C |] ==> A *Un* B ⊆ C
by *blast*

lemma *Un-cons*: *cons*(a,B) *Un* C = *cons*(a, B *Un* C)
by *blast*

lemma *Un-absorb* [*simp*]: A *Un* A = A
by *blast*

lemma *Un-left-absorb*: A *Un* (A *Un* B) = A *Un* B
by *blast*

lemma *Un-commute*: A *Un* B = B *Un* A
by *blast*

lemma *Un-left-commute*: A *Un* (B *Un* C) = B *Un* (A *Un* C)
by *blast*

lemma *Un-assoc*: (A *Un* B) *Un* C = A *Un* (B *Un* C)
by *blast*

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

lemma *Un-absorb1*: A ⊆ B ==> A ∪ B = B
by *blast*

lemma *Un-absorb2*: B ⊆ A ==> A ∪ B = A
by *blast*

lemma *Un-Int-distrib*: (A *Int* B) *Un* C = (A *Un* C) *Int* (B *Un* C)
by *blast*

lemma *subset-Un-iff*: A⊆B <−> A *Un* B = B
by (*blast elim*!: *equalityE*)

lemma *subset-Un-iff2*: A⊆B <−> B *Un* A = B
by (*blast elim*!: *equalityE*)

lemma *Un-empty* [*iff*]: (A *Un* B = *0*) <−> (A = *0* & B = *0*)
by *blast*

lemma *Un-eq-Union*: A *Un* B = *Union*({A, B})
by *blast*

4.6 Set Difference

lemma *Diff-subset*: A−B ⊆ A
by *blast*

lemma *Diff-contains*: [| C⊆A; C Int B = 0 |] ==> C ⊆ A−B
by *blast*

lemma *subset-Diff-cons-iff*: B ⊆ A − cons(c,C) <−> B⊆A−C & c ~: B
by *blast*

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

lemma *Diff-triv*: A Int B = 0 ==> A − B = A
by *blast*

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

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

lemma *Diff-eq-0-iff*: A − B = 0 <−> A ⊆ B
by (*blast elim*: *equalityE*)

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

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

lemma *Diff-disjoint*: A Int (B−A) = 0
by *blast*

lemma *Diff-partition*: A⊆B ==> A Un (B−A) = B
by *blast*

lemma *subset-Un-Diff*: A ⊆ B Un (A − B)
by *blast*

lemma *double-complement*: [| A⊆B; B⊆C |] ==> B−(C−A) = A
by *blast*

lemma *double-complement-Un*: (A Un B) − (B−A) = A
by *blast*

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*)
apply *blast*
done

lemma *Diff-Un*: *A* − (*B Un C*) = (*A*−*B*) *Int* (*A*−*C*)
by *blast*

lemma *Diff-Int*: *A* − (*B Int C*) = (*A*−*B*) *Un* (*A*−*C*)
by *blast*

lemma *Un-Diff*: (*A Un B*) − *C* = (*A* − *C*) *Un* (*B* − *C*)
by *blast*

lemma *Int-Diff*: (*A Int B*) − *C* = *A Int* (*B* − *C*)
by *blast*

lemma *Diff-Int-distrib*: *C Int* (*A*−*B*) = (*C Int A*) − (*C Int B*)
by *blast*

lemma *Diff-Int-distrib2*: (*A*−*B*) *Int C* = (*A Int C*) − (*B Int C*)
by *blast*

lemma *Un-Int-assoc-iff*: (*A Int B*) *Un C* = *A Int* (*B Un C*) <−> *C*⊆*A*
by (*blast elim*!: *equalityE*)

4.7 Big Union and Intersection

lemma *Union-subset-iff*: *Union*(*A*) ⊆ *C* <−> (∀ *x*∈*A*. *x* ⊆ *C*)
by *blast*

lemma *Union-upper*: *B*∈*A* ==> *B* ⊆ *Union*(*A*)
by *blast*

lemma *Union-least*: [| !!*x*. *x*∈*A* ==> *x*⊆*C* |] ==> *Union*(*A*) ⊆ *C*
by *blast*

lemma *Union-cons* [*simp*]: *Union*(*cons*(*a*,*B*)) = *a Un Union*(*B*)
by *blast*

lemma *Union-Un-distrib*: *Union*(*A Un B*) = *Union*(*A*) *Un Union*(*B*)
by *blast*

lemma *Union-Int-subset*: *Union*(*A Int B*) ⊆ *Union*(*A*) *Int Union*(*B*)
by *blast*

lemma *Union-disjoint*: *Union*(*C*) *Int A* = *0* <−> (∀ *B*∈*C*. *B Int A* = *0*)
by (*blast elim*!: *equalityE*)

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

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

lemma *Inter-subset-iff*: $A \neq 0 \ \ ==> \ \ C \subseteq Inter(A) <-> (\forall x \in A.\ C \subseteq x)$
by *blast*

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

lemma *Inter-greatest*: $[|\ A \neq 0;\ \ !!x.\ x \in A ==> C \subseteq x\ |] ==> C \subseteq Inter(A)$
by *blast*

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

lemma *INT-greatest*: $[|\ A \neq 0;\ \ !!x.\ x \in A ==> C \subseteq B(x)\ |] ==> C \subseteq (\bigcap x \in A.$
$B(x))$
by *force*

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

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

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

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

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

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

4.8 Unions and Intersections of Families

lemma *subset-UN-iff-eq*: $A \subseteq (\bigcup i{\in}I.\ B(i)) <-> A = (\bigcup i{\in}I.\ A\ Int\ B(i))$
by (*blast elim*!: *equalityE*)

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

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

lemma *UN-least*: $[|\ !!x.\ x{\in}A ==> B(x){\subseteq}C\ |] ==> (\bigcup x{\in}A.\ B(x)) \subseteq C$
by *blast*

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

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

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

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

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

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

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

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

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

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

lemma *Un-INT-distrib2*: $[|\ I\neq 0;\ \ J\neq 0\ |] ==>$
$(\bigcap i\in I.\ A(i))\ Un\ (\bigcap j\in J.\ B(j)) = (\bigcap i\in I.\ \bigcap j\in J.\ A(i)\ Un\ B(j))$
by *auto*

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

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

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

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

lemma *INT-Union-eq*:
$0\ \sim: A ==> (\bigcap x\in\ Union(A).\ B(x)) = (\bigcap y\in A.\ \bigcap x\in y.\ B(x))$
apply (*subgoal-tac* $\forall x\in A.\ x\sim=0$)
prefer *2* **apply** *blast*
apply (*force simp add*: *Inter-def ball-conj-distrib*)
done

lemma *INT-UN-eq*:
$(\forall x\in A.\ B(x)\ \sim= 0)$
$==> (\bigcap z\in\ (\bigcup x\in A.\ B(x)).\ C(z)) = (\bigcap x\in A.\ \bigcap z\in\ B(x).\ C(z))$
apply (*subst INT-Union-eq*, *blast*)
apply (*simp add*: *Inter-def*)
done

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

lemma *INT-Int-distrib*:
$I\neq 0 ==> (\bigcap i\in I.\ A(i)\ Int\ B(i)) = (\bigcap i\in I.\ A(i))\ Int\ (\bigcap i\in I.\ B(i))$
by (*blast elim*!: *not-emptyE*)

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

lemma *Diff-UN*: $I\neq 0 ==> B - (\bigcup i\in I.\ A(i)) = (\bigcap i\in I.\ B - A(i))$
by (*blast elim*!: *not-emptyE*)

lemma *Diff-INT*: $I \neq 0 ==> B - (\bigcap i \in I.\ A(i)) = (\bigcup i \in I.\ B - A(i))$
by (*blast elim*!: *not-emptyE*)

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

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

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

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

lemma *SUM-UN-distrib1*:
$(\Sigma\ x \in (\bigcup y \in A.\ C(y)).\ B(x)) = (\bigcup y \in A.\ \Sigma\ x \in C(y).\ B(x))$
by *blast*

lemma *SUM-UN-distrib2*:
$(\Sigma\ i \in I.\ \bigcup j \in J.\ C(i,j)) = (\bigcup j \in J.\ \Sigma\ i \in I.\ C(i,j))$
by *blast*

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

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

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

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

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

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

lemma *SUM-eq-UN*: (Σ i∈I. A(i)) = (⋃i∈I. {i} ∗ A(i))
by *blast*

lemma *times-subset-iff*:
  (A′∗B′ ⊆ A∗B) <−> (A′ = 0 | B′ = 0 | (A′⊆A) & (B′⊆B))
by *blast*

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))
by *blast*

lemma *domain-iff*: a: domain(r) <−> (EX y. <a,y>∈ r)
by (*unfold domain-def*, *blast*)

lemma *domainI* [*intro*]: <a,b>∈ r ==> a: domain(r)
by (*unfold domain-def*, *blast*)

lemma *domainE* [*elim*!]:
  [| a ∈ domain(r); !!y. <a,y>∈ r ==> P |] ==> P
by (*unfold domain-def*, *blast*)

lemma *domain-subset*: domain(Sigma(A,B)) ⊆ A
by *blast*

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

lemma *domain-0* [*simp*]: domain(0) = 0
by *blast*

lemma *domain-cons* [*simp*]: domain(cons(<a,b>,r)) = cons(a, domain(r))
by *blast*

lemma *domain-Un-eq* [*simp*]: domain(A Un B) = domain(A) Un domain(B)
by *blast*

lemma *domain-Int-subset*: domain(A Int B) ⊆ domain(A) Int domain(B)
by *blast*

lemma *domain-Diff-subset*: domain(A) − domain(B) ⊆ domain(A − B)
by *blast*

lemma *domain-UN*: *domain*(⋃*x*∈*A*. *B*(*x*)) = (⋃*x*∈*A*. *domain*(*B*(*x*)))
by *blast*

lemma *domain-Union*: *domain*(*Union*(*A*)) = (⋃*x*∈*A*. *domain*(*x*))
by *blast*

lemma *rangeI* [*intro*]: <*a*,*b*>∈ *r* ==> *b* ∈ *range*(*r*)
apply (*unfold range-def*)
apply (*erule converseI* [*THEN domainI*])
done

lemma *rangeE* [*elim*!]: [| *b* ∈ *range*(*r*); !!*x*. <*x*,*b*>∈ *r* ==> *P* |] ==> *P*
by (*unfold range-def*, *blast*)

lemma *range-subset*: *range*(*A*∗*B*) ⊆ *B*
apply (*unfold range-def*)
apply (*subst converse-prod*)
apply (*rule domain-subset*)
done

lemma *range-of-prod*: *a*∈*A* ==> *range*(*A*∗*B*) = *B*
by *blast*

lemma *range-0* [*simp*]: *range*(*0*) = *0*
by *blast*

lemma *range-cons* [*simp*]: *range*(*cons*(<*a*,*b*>,*r*)) = *cons*(*b*, *range*(*r*))
by *blast*

lemma *range-Un-eq* [*simp*]: *range*(*A Un B*) = *range*(*A*) *Un range*(*B*)
by *blast*

lemma *range-Int-subset*: *range*(*A Int B*) ⊆ *range*(*A*) *Int range*(*B*)
by *blast*

lemma *range-Diff-subset*: *range*(*A*) − *range*(*B*) ⊆ *range*(*A* − *B*)
by *blast*

lemma *domain-converse* [*simp*]: *domain*(*converse*(*r*)) = *range*(*r*)
by *blast*

lemma *range-converse* [*simp*]: *range*(*converse*(*r*)) = *domain*(*r*)
by *blast*

lemma *fieldI1*: <a,b>∈ r ==> a ∈ field(r)
by (unfold field-def, blast)

lemma *fieldI2*: <a,b>∈ r ==> b ∈ field(r)
by (unfold field-def, blast)

lemma *fieldCI* [intro]:
  (~ <c,a>∈r ==> <a,b>∈ r) ==> a ∈ field(r)
apply (unfold field-def, blast)
done

lemma *fieldE* [elim!]:
  [| a ∈ field(r);
    !!x. <a,x>∈ r ==> P;
    !!x. <x,a>∈ r ==> P |] ==> P
by (unfold field-def, blast)

lemma *field-subset*: field(A∗B) ⊆ A Un B
by blast

lemma *domain-subset-field*: domain(r) ⊆ field(r)
apply (unfold field-def)
apply (rule Un-upper1)
done

lemma *range-subset-field*: range(r) ⊆ field(r)
apply (unfold field-def)
apply (rule Un-upper2)
done

lemma *domain-times-range*: r ⊆ Sigma(A,B) ==> r ⊆ domain(r)∗range(r)
by blast

lemma *field-times-field*: r ⊆ Sigma(A,B) ==> r ⊆ field(r)∗field(r)
by blast

lemma *relation-field-times-field*: relation(r) ==> r ⊆ field(r)∗field(r)
by (simp add: relation-def, blast)

lemma *field-of-prod*: field(A∗A) = A
by blast

lemma *field-0* [simp]: field(0) = 0
by blast

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

lemma *field-Un-eq* [*simp*]: *field*(*A Un B*) = *field*(*A*) *Un field*(*B*)
by *blast*

lemma *field-Int-subset*: *field*(*A Int B*) ⊆ *field*(*A*) *Int field*(*B*)
by *blast*

lemma *field-Diff-subset*: *field*(*A*) − *field*(*B*) ⊆ *field*(*A* − *B*)
by *blast*

lemma *field-converse* [*simp*]: *field*(*converse*(*r*)) = *field*(*r*)
by *blast*

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

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

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

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

4.9 Image of a Set under a Function or Relation

lemma *image-iff*: *b* ∈ *r*``*A* <−> (∃ *x*∈*A*. <*x*,*b*>∈*r*)
by (*unfold image-def*, *blast*)

lemma *image-singleton-iff*: *b* ∈ *r*``{*a*} <−> <*a*,*b*>∈*r*
by (*rule image-iff* [*THEN iff-trans*], *blast*)

lemma *imageI* [*intro*]: [| <*a*,*b*>∈ *r*; *a*∈*A* |] ==> *b* ∈ *r*``*A*
by (*unfold image-def*, *blast*)

lemma *imageE* [*elim*!]:
[| *b*: *r*``*A*; !!*x*.[| <*x*,*b*>∈ *r*; *x*∈*A* |] ==> *P* |] ==> *P*
by (*unfold image-def*, *blast*)

lemma *image-subset*: *r* ⊆ *A*∗*B* ==> *r*``*C* ⊆ *B*
by *blast*

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

lemma *image-Un* [*simp*]: *r``(A Un B) = (r``A) Un (r``B)*
by *blast*

lemma *image-UN*: *r `` (⋃x∈A. B(x)) = (⋃x∈A. r `` B(x))*
by *blast*

lemma *Collect-image-eq*:
{z ∈ Sigma(A,B). P(z)} `` C = (⋃x ∈ A. {y ∈ B(x). x ∈ C & P(<x,y>)})
by *blast*

lemma *image-Int-subset*: *r``(A Int B) ⊆ (r``A) Int (r``B)*
by *blast*

lemma *image-Int-square-subset*: *(r Int A*A)``B ⊆ (r``B) Int A*
by *blast*

lemma *image-Int-square*: *B⊆A ==> (r Int A*A)``B = (r``B) Int A*
by *blast*

lemma *image-0-left* [*simp*]: *0``A = 0*
by *blast*

lemma *image-Un-left*: *(r Un s)``A = (r``A) Un (s``A)*
by *blast*

lemma *image-Int-subset-left*: *(r Int s)``A ⊆ (r``A) Int (s``A)*
by *blast*

4.10 Inverse Image of a Set under a Function or Relation

lemma *vimage-iff*:
a ∈ r−``B <−> (∃ y∈B. <a,y>∈r)
by (*unfold vimage-def image-def converse-def*, *blast*)

lemma *vimage-singleton-iff*: *a ∈ r−``{b} <−> <a,b>∈r*
by (*rule vimage-iff* [*THEN iff-trans*], *blast*)

lemma *vimageI* [*intro*]: *[| <a,b>∈ r; b∈B |] ==> a ∈ r−``B*
by (*unfold vimage-def*, *blast*)

lemma *vimageE* [*elim*!]:
[| a: r−``B; !!x.[| <a,x>∈ r; x∈B |] ==> P |] ==> P
apply (*unfold vimage-def*, *blast*)
done

lemma *vimage-subset*: *r ⊆ A*B ==> r−``C ⊆ A*
apply (*unfold vimage-def*)

apply (*erule converse-type* [*THEN image-subset*])
done

lemma *vimage-0* [*simp*]: $r-``0 = 0$
by *blast*

lemma *vimage-Un* [*simp*]: $r-``(A\ Un\ B) = (r-``A)\ Un\ (r-``B)$
by *blast*

lemma *vimage-Int-subset*: $r-``(A\ Int\ B) \subseteq (r-``A)\ Int\ (r-``B)$
by *blast*

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

lemma *function-vimage-Int*:
    $function(f) \Longrightarrow f-``(A\ Int\ B) = (f-``A)\ \ Int\ \ (f-``B)$
by (*unfold function-def*, *blast*)

lemma *function-vimage-Diff*: $function(f) \Longrightarrow f-``(A-B) = (f-``A) - (f-``B)$
by (*unfold function-def*, *blast*)

lemma *function-image-vimage*: $function(f) \Longrightarrow f\ ``\ (f-``\ A) \subseteq A$
by (*unfold function-def*, *blast*)

lemma *vimage-Int-square-subset*: $(r\ Int\ A{*}A)-``B \subseteq (r-``B)\ Int\ A$
by *blast*

lemma *vimage-Int-square*: $B{\subseteq}A \Longrightarrow (r\ Int\ A{*}A)-``B = (r-``B)\ Int\ A$
by *blast*

lemma *vimage-0-left* [*simp*]: $0-``A = 0$
by *blast*

lemma *vimage-Un-left*: $(r\ Un\ s)-``A = (r-``A)\ Un\ (s-``A)$
by *blast*

lemma *vimage-Int-subset-left*: $(r\ Int\ s)-``A \subseteq (r-``A)\ Int\ (s-``A)$
by *blast*

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

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

lemma *converse-Diff* [*simp*]: *converse*(*A* − *B*) = *converse*(*A*) − *converse*(*B*)
by *blast*

lemma *converse-UN* [*simp*]: *converse*(⋃*x*∈*A*. *B*(*x*)) = (⋃*x*∈*A*. *converse*(*B*(*x*)))
by *blast*

lemma *converse-INT* [*simp*]:
converse(⋂*x*∈*A*. *B*(*x*)) = (⋂*x*∈*A*. *converse*(*B*(*x*)))
apply (*unfold Inter-def*, *blast*)
done

4.11 Powerset Operator

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

lemma *Pow-insert*: *Pow* (*cons*(*a*,*A*)) = *Pow*(*A*) *Un* {*cons*(*a*,*X*) . *X*: *Pow*(*A*)}
apply (*rule equalityI*, *safe*)
apply (*erule swap*)
apply (*rule-tac a* = *x*−{*a*} **in** *RepFun-eqI*, *auto*)
done

lemma *Un-Pow-subset*: *Pow*(*A*) *Un Pow*(*B*) ⊆ *Pow*(*A Un B*)
by *blast*

lemma *UN-Pow-subset*: (⋃*x*∈*A*. *Pow*(*B*(*x*))) ⊆ *Pow*(⋃*x*∈*A*. *B*(*x*))
by *blast*

lemma *subset-Pow-Union*: *A* ⊆ *Pow*(*Union*(*A*))
by *blast*

lemma *Union-Pow-eq* [*simp*]: *Union*(*Pow*(*A*)) = *A*
by *blast*

lemma *Union-Pow-iff*: *Union*(*A*) ∈ *Pow*(*B*) <−> *A* ∈ *Pow*(*Pow*(*B*))
by *blast*

lemma *Pow-Int-eq* [*simp*]: *Pow*(*A Int B*) = *Pow*(*A*) *Int Pow*(*B*)
by *blast*

lemma *Pow-INT-eq*: *A*≠*0* ==> *Pow*(⋂*x*∈*A*. *B*(*x*)) = (⋂*x*∈*A*. *Pow*(*B*(*x*)))
by (*blast elim*!: *not-emptyE*)

4.12 RepFun

lemma *RepFun-subset*: [| !!x. x∈A ==> f(x) ∈ B |] ==> {f(x). x∈A} ⊆ B
by *blast*

lemma *RepFun-eq-0-iff* [*simp*]: {f(x).x∈A}=0 <-> A=0
by *blast*

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

4.13 Collect

lemma *Collect-subset*: *Collect*(A,P) ⊆ A
by *blast*

lemma *Collect-Un*: *Collect*(A *Un* B, P) = *Collect*(A,P) *Un* *Collect*(B,P)
by *blast*

lemma *Collect-Int*: *Collect*(A *Int* B, P) = *Collect*(A,P) *Int* *Collect*(B,P)
by *blast*

lemma *Collect-Diff*: *Collect*(A − B, P) = *Collect*(A,P) − *Collect*(B,P)
by *blast*

lemma *Collect-cons*: {x∈*cons*(a,B). P(x)} =
(*if* P(a) *then* *cons*(a, {x∈B. P(x)}) *else* {x∈B. P(x)})
by (*simp*, *blast*)

lemma *Int-Collect-self-eq*: A *Int* *Collect*(A,P) = *Collect*(A,P)
by *blast*

lemma *Collect-Collect-eq* [*simp*]:
Collect(*Collect*(A,P), Q) = *Collect*(A, %x. P(x) & Q(x))
by *blast*

lemma *Collect-Int-Collect-eq*:
Collect(A,P) *Int* *Collect*(A,Q) = *Collect*(A, %x. P(x) & Q(x))
by *blast*

lemma *Collect-Union-eq* [*simp*]:
Collect(⋃x∈A. B(x), P) = (⋃x∈A. *Collect*(B(x), P))
by *blast*

lemma *Collect-Int-left*: {x∈A. P(x)} *Int* B = {x ∈ A *Int* B. P(x)}
by *blast*

lemma *Collect-Int-right*: A *Int* {x∈B. P(x)} = {x ∈ A *Int* B. P(x)}
by *blast*

lemma *Collect-disj-eq*: {*x*∈*A*. *P*(*x*) | *Q*(*x*)} = *Collect*(*A*, *P*) *Un Collect*(*A*, *Q*)
by *blast*

lemma *Collect-conj-eq*: {*x*∈*A*. *P*(*x*) & *Q*(*x*)} = *Collect*(*A*, *P*) *Int Collect*(*A*, *Q*)
by *blast*

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

ML ⟪
```
val subset-cs = @{claset}
    delrules [@{thm subsetI}, @{thm subsetCE}]
    addSIs @{thms subset-SIs}
    addIs  [@{thm Union-upper}, @{thm Inter-lower}]
    addSEs [@{thm cons-subsetE}];
```
⟫

ML
⟪
```
val ZF-cs = @{claset} delrules [@{thm equalityI}];
```
⟫

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-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-mono(D,h)
by (*unfold bnd-mono-def*, *clarify*, *blast*)

lemma *bnd-monoD1*: bnd-mono(D,h) ==> h(D) <= D
apply (*unfold bnd-mono-def*)
apply (*erule conjunct1*)
done

lemma *bnd-monoD2*: [| bnd-mono(D,h); W<=X; X<=D |] ==> h(W) <= h(X)
by (*unfold bnd-mono-def*, *blast*)

lemma *bnd-mono-subset*:
[| bnd-mono(D,h); X<=D |] ==> h(X) <= D
by (*unfold bnd-mono-def*, *clarify*, *blast*)

lemma *bnd-mono-Un*:
[| bnd-mono(D,h); A <= D; B <= D |] ==> h(A) Un h(B) <= h(A Un B)
apply (*unfold bnd-mono-def*)
apply (*rule Un-least*, *blast+*)
done

lemma *bnd-mono-UN*:
[| bnd-mono(D,h); ∀ i∈I. A(i) <= D |]
==> (⋃ i∈I. h(A(i))) <= h((⋃ i∈I. A(i)))
apply (*unfold bnd-mono-def*)
apply (*rule UN-least*)
apply (*elim conjE*)
apply (*drule-tac x=A(i)* **in** *spec*)
apply (*drule-tac x=(⋃ i∈I. A(i))* **in** *spec*)
apply *blast*
done

lemma *bnd-mono-Int*:
[| bnd-mono(D,h); A <= D; B <= D |] ==> h(A Int B) <= h(A) Int h(B)
apply (*rule Int-greatest*)
apply (*erule bnd-monoD2*, *rule Int-lower1*, *assumption*)
apply (*erule bnd-monoD2*, *rule Int-lower2*, *assumption*)
done

5.2 Proof of Knaster-Tarski Theorem using *lfp*

lemma *lfp-lowerbound*:
  [| *h*(*A*) <= *A*; *A*<=*D* |] ==> *lfp*(*D*,*h*) <= *A*
by (*unfold lfp-def*, *blast*)

lemma *lfp-subset*: *lfp*(*D*,*h*) <= *D*
by (*unfold lfp-def Inter-def*, *blast*)

lemma *def-lfp-subset*: *A* == *lfp*(*D*,*h*) ==> *A* <= *D*
apply *simp*
apply (*rule lfp-subset*)
done

lemma *lfp-greatest*:
  [| *h*(*D*) <= *D*; !!*X*. [| *h*(*X*) <= *X*; *X*<=*D* |] ==> *A*<=*X* |] ==> *A* <= *lfp*(*D*,*h*)
by (*unfold lfp-def*, *blast*)

lemma *lfp-lemma1*:
  [| *bnd-mono*(*D*,*h*); *h*(*A*)<=*A*; *A*<=*D* |] ==> *h*(*lfp*(*D*,*h*)) <= *A*
apply (*erule bnd-monoD2* [*THEN subset-trans*])
apply (*rule lfp-lowerbound*, *assumption*+)
done

lemma *lfp-lemma2*: *bnd-mono*(*D*,*h*) ==> *h*(*lfp*(*D*,*h*)) <= *lfp*(*D*,*h*)
apply (*rule bnd-monoD1* [*THEN lfp-greatest*])
apply (*rule-tac* [*2*] *lfp-lemma1*)
apply (*assumption*+)
done

lemma *lfp-lemma3*:
  bnd-mono(*D*,*h*) ==> *lfp*(*D*,*h*) <= *h*(*lfp*(*D*,*h*))
apply (*rule lfp-lowerbound*)
apply (*rule bnd-monoD2*, *assumption*)
apply (*rule lfp-lemma2*, *assumption*)
apply (*erule-tac* [*2*] *bnd-mono-subset*)
apply (*rule lfp-subset*)+
done

lemma *lfp-unfold*: *bnd-mono*(*D*,*h*) ==> *lfp*(*D*,*h*) = *h*(*lfp*(*D*,*h*))
apply (*rule equalityI*)
apply (*erule lfp-lemma3*)
apply (*erule lfp-lemma2*)
done

lemma *def-lfp-unfold*:

```
  [| A==lfp(D,h);  bnd-mono(D,h) |] ==> A = h(A)
apply simp
apply (erule lfp-unfold)
done
```

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)
by (blast intro: lfp-lemma2 [THEN subsetD] bnd-monoD2 [THEN subsetD]
            lfp-subset [THEN subsetD])
```

```
lemma induct:
  [| bnd-mono(D,h);  a : lfp(D,h);
     !!x. x : h(Collect(lfp(D,h),P)) ==> P(x)
   |] ==> P(a)
apply (rule Collect-is-pre-fixedpt
          [THEN lfp-lowerbound, THEN subsetD, THEN CollectD2])
apply (rule-tac [3] lfp-subset [THEN Collect-subset [THEN subset-trans]],
     blast+)
done
```

```
lemma def-induct:
  [| A == lfp(D,h);  bnd-mono(D,h);  a:A;
     !!x. x : h(Collect(A,P)) ==> P(x)
   |] ==> P(a)
by (rule induct, blast+)
```

```
lemma lfp-Int-lowerbound:
  [| h(D Int A) <= A;  bnd-mono(D,h) |] ==> lfp(D,h) <= A
apply (rule lfp-lowerbound [THEN subset-trans])
apply (erule bnd-mono-subset [THEN Int-greatest], blast+)
done
```

```
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)
apply (rule bnd-monoD1 [THEN lfp-greatest])
apply (rule imono)
apply (rule hmono [THEN [2] lfp-Int-lowerbound])
apply (rule Int-lower1 [THEN subhi, THEN subset-trans])
apply (rule imono [THEN bnd-monoD2, THEN subset-trans], auto)
```

done

lemma *lfp-mono2*:
[| *i*(*D*) <= *D*; !!*X*. *X*<=*D* ==> *h*(*X*) <= *i*(*X*) |] ==> *lfp*(*D*,*h*) <= *lfp*(*D*,*i*)
apply (*rule lfp-greatest*, *assumption*)
apply (*rule lfp-lowerbound*, *blast*, *assumption*)
done

lemma *lfp-cong*:
[|*D*=*D*′; !!*X*. *X* <= *D*′ ==> *h*(*X*) = *h*′(*X*)|] ==> *lfp*(*D*,*h*) = *lfp*(*D*′,*h*′)
apply (*simp add*: *lfp-def*)
apply (*rule-tac t*=*Inter* **in** *subst-context*)
apply (*rule Collect-cong*, *simp-all*)
done

5.4 Proof of Knaster-Tarski Theorem using *gfp*

lemma *gfp-upperbound*: [| *A* <= *h*(*A*); *A*<=*D* |] ==> *A* <= *gfp*(*D*,*h*)
apply (*unfold gfp-def*)
apply (*rule PowI* [*THEN CollectI*, *THEN Union-upper*])
apply (*assumption*+)
done

lemma *gfp-subset*: *gfp*(*D*,*h*) <= *D*
by (*unfold gfp-def*, *blast*)

lemma *def-gfp-subset*: *A*==*gfp*(*D*,*h*) ==> *A* <= *D*
apply *simp*
apply (*rule gfp-subset*)
done

lemma *gfp-least*:
[| *bnd-mono*(*D*,*h*); !!*X*. [| *X* <= *h*(*X*); *X*<=*D* |] ==> *X*<=*A* |] ==>
gfp(*D*,*h*) <= *A*
apply (*unfold gfp-def*)
apply (*blast dest*: *bnd-monoD1*)
done

lemma *gfp-lemma1*:
[| *bnd-mono*(*D*,*h*); *A*<=*h*(*A*); *A*<=*D* |] ==> *A* <= *h*(*gfp*(*D*,*h*))
apply (*rule subset-trans*, *assumption*)
apply (*erule bnd-monoD2*)
apply (*rule-tac* [*2*] *gfp-subset*)
apply (*simp add*: *gfp-upperbound*)
done

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

apply (*rule gfp-least*)
apply (*rule-tac* [*2*] *gfp-lemma1*)
apply (*assumption*+)
done

lemma *gfp-lemma3*:
  bnd-mono(*D*,*h*) ==> *h*(*gfp*(*D*,*h*)) <= *gfp*(*D*,*h*)
apply (*rule gfp-upperbound*)
apply (*rule bnd-monoD2*, *assumption*)
apply (*rule gfp-lemma2*, *assumption*)
apply (*erule bnd-mono-subset*, *rule gfp-subset*)+
done

lemma *gfp-unfold*: *bnd-mono*(*D*,*h*) ==> *gfp*(*D*,*h*) = *h*(*gfp*(*D*,*h*))
apply (*rule equalityI*)
apply (*erule gfp-lemma2*)
apply (*erule gfp-lemma3*)
done

lemma *def-gfp-unfold*:
  [| *A*==*gfp*(*D*,*h*); *bnd-mono*(*D*,*h*) |] ==> *A* = *h*(*A*)
apply *simp*
apply (*erule gfp-unfold*)
done

5.5 Coinduction Rules for Greatest Fixed Points

lemma *weak-coinduct*: [| *a*: *X*; *X* <= *h*(*X*); *X* <= *D* |] ==> *a* : *gfp*(*D*,*h*)
by (*blast intro*: *gfp-upperbound* [*THEN subsetD*])

lemma *coinduct-lemma*:
  [| *X* <= *h*(*X Un gfp*(*D*,*h*)); *X* <= *D*; *bnd-mono*(*D*,*h*) |] ==>
   X Un gfp(*D*,*h*) <= *h*(*X Un gfp*(*D*,*h*))
apply (*erule Un-least*)
apply (*rule gfp-lemma2* [*THEN subset-trans*], *assumption*)
apply (*rule Un-upper2* [*THEN subset-trans*])
apply (*rule bnd-mono-Un*, *assumption*+)
apply (*rule gfp-subset*)
done

lemma *coinduct*:
   [| *bnd-mono*(*D*,*h*); *a*: *X*; *X* <= *h*(*X Un gfp*(*D*,*h*)); *X* <= *D* |]
    ==> *a* : *gfp*(*D*,*h*)
apply (*rule weak-coinduct*)
apply (*erule-tac* [*2*] *coinduct-lemma*)
apply (*simp-all add*: *gfp-subset Un-subset-iff*)
done

lemma *def-coinduct*:
[| *A* == *gfp*(*D*,*h*); *bnd-mono*(*D*,*h*); *a*: *X*; *X* <= *h*(*X Un A*); *X* <= *D* |]
==>
a : *A*
apply *simp*
apply (*rule coinduct*, *assumption*+)
done

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

a: *X*; *X* <= *D*; !!*z*. *z*: *X* ==> *P*(*X Un A*, *z*) |] ==>
a : *A*
apply (*rule def-coinduct*, *assumption*+, *blast*+)
done

lemma *gfp-mono*:
[| *bnd-mono*(*D*,*h*); *D* <= *E*;
!!*X*. *X*<=*D* ==> *h*(*X*) <= *i*(*X*) |] ==> *gfp*(*D*,*h*) <= *gfp*(*E*,*i*)
apply (*rule gfp-upperbound*)
apply (*rule gfp-lemma2* [*THEN subset-trans*], *assumption*)
apply (*blast del*: *subsetI intro*: *gfp-subset*)
apply (*blast del*: *subsetI intro*: *subset-trans gfp-subset*)
done

end

6 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*
by (*simp add*: *succ-def*)

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

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

lemma *one-not-0*: *1*~=*0*
by (*simp add*: *bool-defs*)

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

lemma *boolE*:
[| *c*: *bool*; *c*=*1* ==> *P*; *c*=*0* ==> *P* |] ==> *P*
by (*simp add*: *bool-defs*, *blast*)

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

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

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

by (*simp add*: *bool-defs*, *blast*)

lemma *cond-simple-type*: [| *b*: *bool*; *c*: *A*; *d*: *A* |] ==> *cond*(*b*,*c*,*d*): *A*
by (*simp add*: *bool-defs*)

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

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

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*
by (*simp add*: *not-def*)

lemma *and-type* [*TC*]: [| *a*:*bool*; *b*:*bool* |] ==> *a and b* : *bool*
by (*simp add*: *and-def*)

lemma *or-type* [*TC*]: [| *a*:*bool*; *b*:*bool* |] ==> *a or b* : *bool*
by (*simp add*: *or-def*)

lemma *xor-type* [*TC*]: [| *a*:*bool*; *b*:*bool* |] ==> *a xor b* : *bool*
by (*simp add*: *xor-def*)

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

6.1 Laws About 'not'

lemma *not-not* [*simp*]: *a*:*bool* ==> *not*(*not*(*a*)) = *a*
by (*elim boolE*, *auto*)

lemma *not-and* [*simp*]: *a*:*bool* ==> *not*(*a and b*) = *not*(*a*) *or not*(*b*)
by (*elim boolE*, *auto*)

lemma *not-or* [*simp*]: *a*:*bool* ==> *not*(*a or b*) = *not*(*a*) *and not*(*b*)
by (*elim boolE*, *auto*)

6.2 Laws About 'and'

lemma *and-absorb* [*simp*]: *a*: *bool* ==> *a and a* = *a*
by (*elim boolE*, *auto*)

lemma *and-commute*: [| *a*: *bool*; *b*:*bool* |] ==> *a and b* = *b and a*
by (*elim boolE*, *auto*)

lemma *and-assoc*: *a*: *bool* ==> (*a and b*) *and c* = *a and* (*b and c*)
by (*elim boolE*, *auto*)

lemma *and-or-distrib*: [| *a*: *bool*; *b*:*bool*; *c*:*bool* |] ==>
(*a or b*) *and c* = (*a and c*) *or* (*b and c*)
by (*elim boolE*, *auto*)

6.3 Laws About 'or'

lemma *or-absorb* [*simp*]: *a*: *bool* ==> *a or a* = *a*
by (*elim boolE*, *auto*)

lemma *or-commute*: [| *a*: *bool*; *b*:*bool* |] ==> *a or b* = *b or a*
by (*elim boolE*, *auto*)

lemma *or-assoc*: *a*: *bool* ==> (*a or b*) *or c* = *a or* (*b or c*)
by (*elim boolE*, *auto*)

lemma *or-and-distrib*: [| *a*: *bool*; *b*: *bool*; *c*: *bool* |] ==>
(*a and b*) *or c* = (*a or c*) *and* (*b or c*)
by (*elim boolE*, *auto*)

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

lemma [*simp*]: *bool-of-o*(*True*) = *1*
by (*simp add*: *bool-of-o-def*)

lemma [*simp*]: *bool-of-o*(*False*) = *0*
by (*simp add*: *bool-of-o-def*)

lemma [*simp*,*TC*]: *bool-of-o*(*P*) ∈ *bool*
by (*simp add*: *bool-of-o-def*)

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

lemma [*simp*]: (*bool-of-o*(*P*) = *0*) <-> ~*P*
by (*simp add*: *bool-of-o-def*)

ML
⟪
val bool-def = thm bool-def;

val bool-defs = thms bool-defs;
val singleton-0 = thm singleton-0;
val bool-1I = thm bool-1I;
val bool-0I = thm bool-0I;
val one-not-0 = thm one-not-0;
val one-neq-0 = thm one-neq-0;
val boolE = thm boolE;
val cond-1 = thm cond-1;
val cond-0 = thm cond-0;
val cond-type = thm cond-type;
val cond-simple-type = thm cond-simple-type;
val def-cond-1 = thm def-cond-1;
val def-cond-0 = thm def-cond-0;
val not-1 = thm not-1;
val not-0 = thm not-0;
val and-1 = thm and-1;
val and-0 = thm and-0;
val or-1 = thm or-1;
val or-0 = thm or-0;
val xor-1 = thm xor-1;
val xor-0 = thm xor-0;
val not-type = thm not-type;
val and-type = thm and-type;
val or-type = thm or-type;
val xor-type = thm xor-type;
val bool-typechecks = thms bool-typechecks;
val not-not = thm not-not;
val not-and = thm not-and;
val not-or = thm not-or;
val and-absorb = thm and-absorb;
val and-commute = thm and-commute;
val and-assoc = thm and-assoc;
val and-or-distrib = thm and-or-distrib;
val or-absorb = thm or-absorb;
val or-commute = thm or-commute;
val or-assoc = thm or-assoc;
val or-and-distrib = thm or-and-distrib;
⟫

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*))
apply (*unfold Part-def*)
apply (*rule separation*)
done

lemma *Part-eqI* [*intro*]:
  [| *a* : *A*; *a*=*h*(*b*) |] ==> *a* : *Part*(*A*,*h*)
by (*unfold Part-def*, *blast*)

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

lemma *PartE* [*elim*!]:
  [| *a* : *Part*(*A*,*h*); !!*z*. [| *a* : *A*; *a*=*h*(*z*) |] ==> *P*
   |] ==> *P*
apply (*unfold Part-def*, *blast*)
done

lemma *Part-subset*: *Part*(*A*,*h*) <= *A*
apply (*unfold Part-def*)
apply (*rule Collect-subset*)
done

7.2 Rules for Disjoint Sums

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

lemma *Sigma-bool*: *Sigma*(*bool*,*C*) = *C*(*0*) + *C*(*1*)
by (*unfold bool-def sum-def*, *blast*)

lemma *InlI* [*intro*!,*simp*,*TC*]: *a* : *A* ==> *Inl*(*a*) : *A*+*B*
by (*unfold sum-defs*, *blast*)

lemma *InrI* [*intro*!,*simp*,*TC*]: *b* : *B* ==> *Inr*(*b*) : *A*+*B*
by (*unfold sum-defs*, *blast*)

lemma *sumE* [*elim*!]:
  [| *u*: *A*+*B*;
    !!*x*. [| *x*:*A*; *u*=*Inl*(*x*) |] ==> *P*;
    !!*y*. [| *y*:*B*; *u*=*Inr*(*y*) |] ==> *P*
  |] ==> *P*
by (*unfold sum-defs*, *blast*)

lemma *Inl-iff* [*iff*]: *Inl*(*a*)=*Inl*(*b*) <-> *a*=*b*
by (*simp add*: *sum-defs*)

lemma *Inr-iff* [*iff*]: *Inr*(*a*)=*Inr*(*b*) <-> *a*=*b*
by (*simp add*: *sum-defs*)

lemma *Inl-Inr-iff* [*simp*]: *Inl*(*a*)=*Inr*(*b*) <-> *False*
by (*simp add*: *sum-defs*)

lemma *Inr-Inl-iff* [*simp*]: *Inr*(*b*)=*Inl*(*a*) <-> *False*
by (*simp add*: *sum-defs*)

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

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

by *blast*

lemma *InrD*: *Inr*(*b*): *A*+*B* ==> *b*: *B*
by *blast*

lemma *sum-iff*: *u*: *A*+*B* <−> (*EX x*. *x*:*A* & *u*=*Inl*(*x*)) | (*EX y*. *y*:*B* & *u*=*Inr*(*y*))
by *blast*

lemma *Inl-in-sum-iff* [*simp*]: (*Inl*(*x*) ∈ *A*+*B*) <−> (*x* ∈ *A*)
by *auto*

lemma *Inr-in-sum-iff* [*simp*]: (*Inr*(*y*) ∈ *A*+*B*) <−> (*y* ∈ *B*)
by *auto*

lemma *sum-subset-iff*: *A*+*B* <= *C*+*D* <−> *A*<=*C* & *B*<=*D*
by *blast*

lemma *sum-equal-iff*: *A*+*B* = *C*+*D* <−> *A*=*C* & *B*=*D*
by (*simp add*: *extension sum-subset-iff*, *blast*)

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

7.3 The Eliminator: *case*

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

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

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*)
by *auto*

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*))))
by *auto*

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*)

by *auto*

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*)
by *auto*

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

lemma *Part-mono*: *A*<=*B* ==> *Part*(*A*,*h*)<=*Part*(*B*,*h*)
by *blast*

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

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

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

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

lemma *PartD1*: *a* : *Part*(*A*,*h*) ==> *a* : *A*
by (*simp add*: *Part-def*)

lemma *Part-id*: *Part*(*A*,%*x*. *x*) = *A*
by *blast*

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

lemma *Part-sum-equality*: *C* <= *A*+*B* ==> *Part*(*C*,*Inl*) *Un* *Part*(*C*,*Inr*) = *C*
by *blast*

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

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

relation(*r*) ==> *converse*(*converse*(*r*)) = *r*
by (*simp add*: *relation-def*, *blast*)

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

lemma *Pi-iff*:
f: *Pi*(*A*,*B*) <-> *function*(*f*) & *f*<=*Sigma*(*A*,*B*) & *A*<=*domain*(*f*)
by (*unfold Pi-def*, *blast*)

lemma *Pi-iff-old*:
f: *Pi*(*A*,*B*) <-> *f*<=*Sigma*(*A*,*B*) & (*ALL x*:*A*. *EX*! *y*. <*x*,*y*>: *f*)
by (*unfold Pi-def function-def*, *blast*)

lemma *fun-is-function*: *f*: *Pi*(*A*,*B*) ==> *function*(*f*)
by (*simp only*: *Pi-iff*)

lemma *function-imp-Pi*:
[|*function*(*f*); *relation*(*f*)|] ==> *f* ∈ *domain*(*f*) -> *range*(*f*)
by (*simp add*: *Pi-iff relation-def*, *blast*)

lemma *functionI*:
[| !!*x y y'*. [| <*x*,*y*>:*r*; <*x*,*y'*>:*r* |] ==> *y*=*y'* |] ==> *function*(*r*)
by (*simp add*: *function-def*, *blast*)

lemma *fun-is-rel*: *f*: *Pi*(*A*,*B*) ==> *f* <= *Sigma*(*A*,*B*)
by (*unfold Pi-def*, *blast*)

lemma *Pi-cong*:
[| *A*=*A'*; !!*x*. *x*:*A'* ==> *B*(*x*)=*B'*(*x*) |] ==> *Pi*(*A*,*B*) = *Pi*(*A'*,*B'*)
by (*simp add*: *Pi-def cong add*: *Sigma-cong*)

lemma *fun-weaken-type*: [| *f*: *A*->*B*; *B*<=*D* |] ==> *f*: *A*->*D*
by (*unfold Pi-def*, *best*)

8.2 Function Application

lemma *apply-equality2*: [| <*a*,*b*>: *f*; <*a*,*c*>: *f*; *f*: *Pi*(*A*,*B*) |] ==> *b*=*c*
by (*unfold Pi-def function-def*, *blast*)

lemma *function-apply-equality*: [| <*a*,*b*>: *f*; *function*(*f*) |] ==> *f*‘*a* = *b*
by (*unfold apply-def function-def*, *blast*)

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

apply (*unfold Pi-def*)
apply (*blast intro*: *function-apply-equality*)
done

lemma *apply-0*: *a* ~: *domain*(*f*) ==> *f‘a* = *0*
by (*unfold apply-def*, *blast*)

lemma *Pi-memberD*: [| *f*: *Pi*(*A*,*B*); *c*: *f* |] ==> *EX x*:*A*. *c* = <*x*,*f‘x*>
apply (*frule fun-is-rel*)
apply (*blast dest*: *apply-equality*)
done

lemma *function-apply-Pair*: [| *function*(*f*); *a* : *domain*(*f*)|] ==> <*a*,*f‘a*>: *f*
apply (*simp add*: *function-def*, *clarify*)
apply (*subgoal-tac f‘a* = *y*, *blast*)
apply (*simp add*: *apply-def*, *blast*)
done

lemma *apply-Pair*: [| *f*: *Pi*(*A*,*B*); *a*:*A* |] ==> <*a*,*f‘a*>: *f*
apply (*simp add*: *Pi-iff*)
apply (*blast intro*: *function-apply-Pair*)
done

lemma *apply-type* [*TC*]: [| *f*: *Pi*(*A*,*B*); *a*:*A* |] ==> *f‘a* : *B*(*a*)
by (*blast intro*: *apply-Pair dest*: *fun-is-rel*)

lemma *apply-funtype*: [| *f*: *A*−>*B*; *a*:*A* |] ==> *f‘a* : *B*
by (*blast dest*: *apply-type*)

lemma *apply-iff*: *f*: *Pi*(*A*,*B*) ==> <*a*,*b*>: *f* <−> *a*:*A* & *f‘a* = *b*
apply (*frule fun-is-rel*)
apply (*blast intro*!: *apply-Pair apply-equality*)
done

lemma *Pi-type*: [| *f*: *Pi*(*A*,*C*); !!*x*. *x*:*A* ==> *f‘x* : *B*(*x*) |] ==> *f* : *Pi*(*A*,*B*)
apply (*simp only*: *Pi-iff*)
apply (*blast dest*: *function-apply-equality*)
done

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*))
by (*blast intro*: *Pi-type dest*: *apply-type*)

lemma *Pi-weaken-type*:
[| *f* : *Pi*(*A*,*B*); !!*x*. *x*:*A* ==> *B*(*x*)<=*C*(*x*) |] ==> *f* : *Pi*(*A*,*C*)
by (*blast intro*: *Pi-type dest*: *apply-type*)

lemma *domain-type*: [| <*a*,*b*> : *f*; *f*: *Pi*(*A*,*B*) |] ==> *a* : *A*
by (*blast dest*: *fun-is-rel*)

lemma *range-type*: [| <*a*,*b*> : *f*; *f*: *Pi*(*A*,*B*) |] ==> *b* : *B*(*a*)
by (*blast dest*: *fun-is-rel*)

lemma *Pair-mem-PiD*: [| <*a*,*b*>: *f*; *f*: *Pi*(*A*,*B*) |] ==> *a*:*A* & *b*:*B*(*a*) & *f*‘*a* = *b*
by (*blast intro*: *domain-type range-type apply-equality*)

8.3 Lambda Abstraction

lemma *lamI*: *a*:*A* ==> <*a*,*b*(*a*)> : (*lam x*:*A*. *b*(*x*))
apply (*unfold lam-def*)
apply (*erule RepFunI*)
done

lemma *lamE*:
[| *p*: (*lam x*:*A*. *b*(*x*)); !!*x*.[| *x*:*A*; *p*=<*x*,*b*(*x*)> |] ==> *P*
|] ==> *P*
by (*simp add*: *lam-def*, *blast*)

lemma *lamD*: [| <*a*,*c*>: (*lam x*:*A*. *b*(*x*)) |] ==> *c* = *b*(*a*)
by (*simp add*: *lam-def*)

lemma *lam-type* [*TC*]:
[| !!*x*. *x*:*A* ==> *b*(*x*): *B*(*x*) |] ==> (*lam x*:*A*. *b*(*x*)) : *Pi*(*A*,*B*)
by (*simp add*: *lam-def Pi-def function-def*, *blast*)

lemma *lam-funtype*: (*lam x*:*A*. *b*(*x*)) : *A* −> {*b*(*x*). *x*:*A*}
by (*blast intro*: *lam-type*)

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

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

lemma *beta-if* [*simp*]: (*lam x*:*A*. *b*(*x*)) ‘ *a* = (*if a* : *A then b*(*a*) *else 0*)
by (*simp add*: *apply-def lam-def*, *blast*)

lemma *beta*: *a* : *A* ==> (*lam x*:*A*. *b*(*x*)) ‘ *a* = *b*(*a*)
by (*simp add*: *apply-def lam-def*, *blast*)

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

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

lemma *lam-cong* [*cong*]:
  [| *A*=*A'*; !!*x*. *x*:*A'* ==> *b*(*x*)=*b'*(*x*) |] ==> *Lambda*(*A*,*b*) = *Lambda*(*A'*,*b'*)
by (*simp only*: *lam-def cong add*: *RepFun-cong*)

lemma *lam-theI*:
  (!!*x*. *x*:*A* ==> *EX*! *y*. *Q*(*x*,*y*)) ==> *EX f*. *ALL x*:*A*. *Q*(*x*, *f*‘*x*)
apply (*rule-tac x* = *lam x*: *A*. *THE y*. *Q* (*x*,*y*) **in** *exI*)
apply *simp*
apply (*blast intro*: *theI*)
done

lemma *lam-eqE*: [| (*lam x*:*A*. *f*(*x*)) = (*lam x*:*A*. *g*(*x*)); *a*:*A* |] ==> *f*(*a*)=*g*(*a*)
by (*fast intro*!: *lamI elim*: *equalityE lamE*)

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

lemma *singleton-fun* [*simp*]: {<*a*,*b*>} : {*a*} −> {*b*}
by (*unfold Pi-def function-def*, *blast*)

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

lemma *fun-space-empty-iff* [*iff*]: (*A*−>*X*)=*0* ⟷ *X*=*0* & (*A* ≠ *0*)
apply *auto*
apply (*fast intro*!: *equals0I intro*: *lam-type*)
done

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*
by (*force dest*: *Pi-memberD intro*: *apply-Pair*)

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

by (*blast del*: *subsetI intro*: *subset-refl sym fun-subset*)

lemma *eta* [*simp*]: *f* : *Pi*(*A*,*B*) ==> (*lam x*:*A*. *f*‘*x*) = *f*
apply (*rule fun-extension*)
apply (*auto simp add*: *lam-type apply-type beta*)
done

lemma *fun-extension-iff*:
  [| *f*:*Pi*(*A*,*B*); *g*:*Pi*(*A*,*C*) |] ==> (*ALL a*:*A*. *f*‘*a* = *g*‘*a*) <-> *f*=*g*
by (*blast intro*: *fun-extension*)

lemma *fun-subset-eq*: [| *f*:*Pi*(*A*,*B*); *g*:*Pi*(*A*,*C*) |] ==> *f* <= *g* <-> (*f* = *g*)
by (*blast dest*: *apply-Pair*
    intro: *fun-extension apply-equality* [*symmetric*])

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*
apply (*rule minor*)
apply (*rule-tac* [*2*] *eta* [*symmetric*])
apply (*blast intro*: *major apply-type*)+
done

8.5 Images of Functions

lemma *image-lam*: *C* <= *A* ==> (*lam x*:*A*. *b*(*x*)) ‘‘ *C* = {*b*(*x*). *x*:*C*}
by (*unfold lam-def*, *blast*)

lemma *Repfun-function-if*:
  function(*f*)
   ==> {*f*‘*x*. *x*:*C*} = (*if C* <= *domain*(*f*) *then f*‘‘*C else cons*(*0*,*f*‘‘*C*))
apply *simp*
apply (*intro conjI impI*)
 apply (*blast dest*: *function-apply-equality intro*: *function-apply-Pair*)
apply (*rule equalityI*)
 apply (*blast intro*!: *function-apply-Pair apply-0*)
apply (*blast dest*: *function-apply-equality intro*: *apply-0* [*symmetric*])
done

lemma *image-function*:
  [| *function*(*f*); *C* <= *domain*(*f*) |] ==> *f*‘‘*C* = {*f*‘*x*. *x*:*C*}
by (*simp add*: *Repfun-function-if*)

lemma *image-fun*: [| *f* : *Pi*(*A*,*B*); *C* <= *A* |] ==> *f*‘‘*C* = {*f*‘*x*. *x*:*C*}

apply (*simp add*: *Pi-iff*)
apply (*blast intro*: *image-function*)
done

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

lemma *Pi-image-cons*:
    [| *f*: *Pi*(*A*,*B*); *x*: *A* |] ==> *f* `` *cons*(*x*,*y*) = *cons*(*f*`*x*, *f*``*y*)
by (*blast dest*: *apply-equality apply-Pair*)

8.6 Properties of *restrict*(*f*, *A*)

lemma *restrict-subset*: *restrict*(*f*,*A*) <= *f*
by (*unfold restrict-def*, *blast*)

lemma *function-restrictI*:
   function(*f*) ==> *function*(*restrict*(*f*,*A*))
by (*unfold restrict-def function-def*, *blast*)

lemma *restrict-type2*: [| *f*: *Pi*(*C*,*B*); *A*<=*C* |] ==> *restrict*(*f*,*A*) : *Pi*(*A*,*B*)
by (*simp add*: *Pi-iff function-def restrict-def*, *blast*)

lemma *restrict*: *restrict*(*f*,*A*) ` *a* = (*if a* : *A then f*`*a else 0*)
by (*simp add*: *apply-def restrict-def*, *blast*)

lemma *restrict-empty* [*simp*]: *restrict*(*f*,*0*) = *0*
by (*unfold restrict-def*, *simp*)

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

lemma *restrict-restrict* [*simp*]:
    restrict(*restrict*(*r*,*A*),*B*) = *restrict*(*r*, *A Int B*)
by (*unfold restrict-def*, *blast*)

lemma *domain-restrict* [*simp*]: *domain*(*restrict*(*f*,*C*)) = *domain*(*f*) *Int C*
apply (*unfold restrict-def*)
apply (*auto simp add*: *domain-def*)
done

lemma *restrict-idem*: *f* <= *Sigma*(*A*,*B*) ==> *restrict*(*f*,*A*) = *f*
by (*simp add*: *restrict-def*, *blast*)

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

by (*simp add*: *restrict-def relation-def*, *blast*)

lemma *domain-restrict-lam* [*simp*]: *domain*(*restrict*(*Lambda*(*A*,*f*),*C*)) = *A Int C*
apply (*unfold restrict-def lam-def*)
apply (*rule equalityI*)
apply (*auto simp add*: *domain-iff*)
done

lemma *restrict-if* [*simp*]: *restrict*(*f*,*A*) ‘ *a* = (*if a* : *A then f*‘*a else 0*)
by (*simp add*: *restrict apply-0*)

lemma *restrict-lam-eq*:
  A<=*C* ==> *restrict*(*lam x*:*C*. *b*(*x*), *A*) = (*lam x*:*A*. *b*(*x*))
by (*unfold restrict-def lam-def*, *auto*)

lemma *fun-cons-restrict-eq*:
  f : *cons*(*a*, *b*) −> *B* ==> *f* = *cons*(<*a*, *f* ‘ *a*>, *restrict*(*f*, *b*))
apply (*rule equalityI*)
prefer *2* **apply** (*blast intro*: *apply-Pair restrict-subset* [*THEN subsetD*])
apply (*auto dest*!: *Pi-memberD simp add*: *restrict-def lam-def*)
done

8.7 Unions of Functions

lemma *function-Union*:
  [| *ALL x*:*S*. *function*(*x*);
    ALL x:*S*. *ALL y*:*S*. *x*<=*y* | *y*<=*x* |]
  ==> *function*(*Union*(*S*))
by (*unfold function-def*, *blast*)

lemma *fun-Union*:
  [| *ALL f*:*S*. *EX C D*. *f*:*C*−>*D*;
    ALL f:*S*. *ALL y*:*S*. *f*<=*y* | *y*<=*f* |] ==>
   Union(*S*) : *domain*(*Union*(*S*)) −> *range*(*Union*(*S*))
apply (*unfold Pi-def*)
apply (*blast intro*!: *rel-Union function-Union*)
done

lemma *gen-relation-Union* [*rule-format*]:
  $\forall f\in F$. *relation*(*f*) $\Longrightarrow$ *relation*(*Union*(*F*))
by (*simp add*: *relation-def*)

lemmas *Un-rls* = *Un-subset-iff SUM-Un-distrib1 prod-Un-distrib2*
    subset-trans [*OF - Un-upper1*]
    subset-trans [*OF - Un-upper2*]

lemma *fun-disjoint-Un*:
[| f: A->B; g: C->D; A Int C = 0 |]
==> (f Un g) : (A Un C) -> (B Un D)

apply (*simp add*: *Pi-iff extension Un-rls*)
apply (*unfold function-def*, *blast*)
done

lemma *fun-disjoint-apply1*: a ∉ domain(g) ==> (f Un g)‘a = f‘a
by (*simp add*: *apply-def*, *blast*)

lemma *fun-disjoint-apply2*: c ∉ domain(f) ==> (f Un g)‘c = g‘c
by (*simp add*: *apply-def*, *blast*)

8.8 Domain and Range of a Function or Relation

lemma *domain-of-fun*: f : Pi(A,B) ==> domain(f)=A
by (*unfold Pi-def*, *blast*)

lemma *apply-rangeI*: [| f : Pi(A,B); a: A |] ==> f‘a : range(f)
by (*erule apply-Pair* [*THEN rangeI*], *assumption*)

lemma *range-of-fun*: f : Pi(A,B) ==> f : A->range(f)
by (*blast intro*: *Pi-type apply-rangeI*)

8.9 Extensions of Functions

lemma *fun-extend*:
[| f: A->B; c~:A |] ==> cons(<c,b>,f) : cons(c,A) -> cons(b,B)
apply (*frule singleton-fun* [*THEN fun-disjoint-Un*], *blast*)
apply (*simp add*: *cons-eq*)
done

lemma *fun-extend3*:
[| f: A->B; c~:A; b: B |] ==> cons(<c,b>,f) : cons(c,A) -> B
by (*blast intro*: *fun-extend* [*THEN fun-weaken-type*])

lemma *extend-apply*:
c ~: domain(f) ==> cons(<c,b>,f)‘a = (if a=c then b else f‘a)
by (*auto simp add*: *apply-def*)

lemma *fun-extend-apply* [*simp*]:
[| f: A->B; c~:A |] ==> cons(<c,b>,f)‘a = (if a=c then b else f‘a)
apply (*rule extend-apply*)
apply (*simp add*: *Pi-def*, *blast*)
done

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

lemma *cons-fun-eq*:
  c ~: *A* ==> *cons*(*c*,*A*) −> *B* = (⋃*f* ∈ *A*−>*B*. ⋃*b*∈*B*. {*cons*(<*c*,*b*>, *f*)})
apply (*rule equalityI*)
apply (*safe elim*!: *fun-extend3*)

apply (*subgoal-tac restrict* (*x*, *A*) : *A* −> *B*)
 prefer *2* **apply** (*blast intro*: *restrict-type2*)
apply (*rule UN-I*, *assumption*)
apply (*rule apply-funtype* [*THEN UN-I*])
  apply *assumption*
 apply (*rule consI1*)
apply (*simp* (*no-asm*))
apply (*rule fun-extension*)
  apply *assumption*
 apply (*blast intro*: *fun-extend*)
apply (*erule consE*, *simp-all*)
done

lemma *succ-fun-eq*: *succ*(*n*) −> *B* = (⋃*f* ∈ *n*−>*B*. ⋃*b*∈*B*. {*cons*(<*n*,*b*>, *f*)})
by (*simp add*: *succ-def mem-not-refl cons-fun-eq*)

8.10 Function Updates

definition
  update :: [*i*,*i*,*i*] => *i* **where**
   update(*f*,*a*,*b*) == *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*)
apply (*simp add*: *update-def*)
apply (*case-tac z* ∈ *domain*(*f*))
apply (*simp-all add*: *apply-0*)
done

lemma *update-idem*: [| *f*‘*x* = *y*; *f*: *Pi*(*A*,*B*); *x*: *A* |] ==> *f*(*x*:=*y*) = *f*
apply (*unfold update-def*)
apply (*simp add*: *domain-of-fun cons-absorb*)
apply (*rule fun-extension*)
apply (*best intro*: *apply-type if-type lam-type*, *assumption*, *simp*)
done

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

lemma *domain-update* [*simp*]: *domain*(*f*(*x*:=*y*)) = *cons*(*x*, *domain*(*f*))
by (*unfold update-def*, *simp*)

lemma *update-type*: [| *f*:*Pi*(*A*,*B*); *x* : *A*; *y*: *B*(*x*) |] ==> *f*(*x*:=*y*) : *Pi*(*A*, *B*)
apply (*unfold update-def*)
apply (*simp add*: *domain-of-fun cons-absorb apply-funtype lam-type*)
done

8.11 Monotonicity Theorems

8.11.1 Replacement in its Various Forms

lemma *Replace-mono*: *A*<=*B* ==> *Replace*(*A*,*P*) <= *Replace*(*B*,*P*)
by (*blast elim*!: *ReplaceE*)

lemma *RepFun-mono*: *A*<=*B* ==> {*f*(*x*). *x*:*A*} <= {*f*(*x*). *x*:*B*}
by *blast*

lemma *Pow-mono*: *A*<=*B* ==> *Pow*(*A*) <= *Pow*(*B*)
by *blast*

lemma *Union-mono*: *A*<=*B* ==> *Union*(*A*) <= *Union*(*B*)
by *blast*

lemma *UN-mono*:
[| *A*<=*C*; !!*x*. *x*:*A* ==> *B*(*x*)<=*D*(*x*) |] ==> (⋃*x*∈*A*. *B*(*x*)) <= (⋃*x*∈*C*. *D*(*x*))
by *blast*

lemma *Inter-anti-mono*: [| *A*<=*B*; *A*≠*0* |] ==> *Inter*(*B*) <= *Inter*(*A*)
by *blast*

lemma *cons-mono*: *C*<=*D* ==> *cons*(*a*,*C*) <= *cons*(*a*,*D*)
by *blast*

lemma *Un-mono*: [| *A*<=*C*; *B*<=*D* |] ==> *A Un B* <= *C Un D*
by *blast*

lemma *Int-mono*: [| A<=C; B<=D |] ==> A Int B <= C Int D
by *blast*

lemma *Diff-mono*: [| A<=C; D<=B |] ==> A−B <= C−D
by *blast*

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)
by *blast*

lemma *sum-mono*: [| A<=C; B<=D |] ==> A+B <= C+D
by (*unfold sum-def*, *blast*)

lemma *Pi-mono*: B<=C ==> A−>B <= A−>C
by (*blast intro*: *lam-type elim*: *Pi-lamE*)

lemma *lam-mono*: A<=B ==> Lambda(A,c) <= Lambda(B,c)
apply (*unfold lam-def*)
apply (*erule RepFun-mono*)
done

8.11.3 Converse, Domain, Range, Field

lemma *converse-mono*: r<=s ==> converse(r) <= converse(s)
by *blast*

lemma *domain-mono*: r<=s ==> domain(r)<=domain(s)
by *blast*

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

lemma *range-mono*: r<=s ==> range(r)<=range(s)
by *blast*

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

lemma *field-mono*: r<=s ==> field(r)<=field(s)
by *blast*

lemma *field-rel-subset*: r <= A∗A ==> field(r) <= A
by (*erule field-mono* [*THEN subset-trans*], *blast*)

8.11.4 Images

lemma *image-pair-mono*:
[| !! x y. <x,y>:r ==> <x,y>:s; A<=B |] ==> r‘‘A <= s‘‘B

by *blast*

lemma *vimage-pair-mono*:
[| !! *x y*. <*x*,*y*>:*r* ==> <*x*,*y*>:*s*; *A*<=*B* |] ==> *r*−‘‘*A* <= *s*−‘‘*B*
by *blast*

lemma *image-mono*: [| *r*<=*s*; *A*<=*B* |] ==> *r*‘‘*A* <= *s*‘‘*B*
by *blast*

lemma *vimage-mono*: [| *r*<=*s*; *A*<=*B* |] ==> *r*−‘‘*A* <= *s*−‘‘*B*
by *blast*

lemma *Collect-mono*:
[| *A*<=*B*; !!*x*. *x*:*A* ==> *P*(*x*) −−> *Q*(*x*) |] ==> *Collect*(*A*,*P*) <= *Collect*(*B*,*Q*)
by *blast*

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* ⊆ *X* |] ==> (*EX x* : *f*‘‘*A*. *P*(*x*)) <−> (*EX x*:*A*. *P*(*f*‘*x*))
apply *safe*
apply *rule*
prefer *2* **apply** *assumption*
apply (*simp add*: *apply-equality*)
apply (*blast intro*: *apply-Pair*)
done

lemma *ball-image-simp*:
[| *f* : *Pi*(*X*, *Y*); *A* ⊆ *X* |] ==> (*ALL x* : *f*‘‘*A*. *P*(*x*)) <−> (*ALL x*:*A*. *P*(*f*‘*x*))
apply *safe*
apply (*blast intro*: *apply-Pair*)
apply (*drule bspec*) **apply** *assumption*
apply (*simp add*: *apply-equality*)
done

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

lemma *QPair-iff* [*simp*]: <*a*;*b*> = <*c*;*d*> <−> *a*=*c* & *b*=*d*
apply (*simp add*: *QPair-def*)
apply (*rule sum-equal-iff*)
done

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

lemma *QPair-inject1*: <*a*;*b*> = <*c*;*d*> ==> *a*=*c*
by *blast*

lemma *QPair-inject2*: <*a*;*b*> = <*c*;*d*> ==> *b*=*d*
by *blast*

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

lemma *QSigmaE* [*elim*!]:
[| *c*: *QSigma*(*A*,*B*);
!!*x* *y*.[| *x*:*A*; *y*:*B*(*x*); *c*=<*x*;*y*> |] ==> *P*
|] ==> *P*
by (*simp add*: *QSigma-def*, *blast*)

lemma *QSigmaE2* [*elim*!]:
[| <*a*;*b*>: *QSigma*(*A*,*B*); [| *a*:*A*; *b*:*B*(*a*) |] ==> *P* |] ==> *P*
by (*simp add*: *QSigma-def*)

lemma *QSigmaD1*: <*a*;*b*> : *QSigma*(*A*,*B*) ==> *a* : *A*
by *blast*

lemma *QSigmaD2*: <*a*;*b*> : *QSigma*(*A*,*B*) ==> *b* : *B*(*a*)
by *blast*

lemma *QSigma-cong*:
[| *A*=*A*′; !!*x*. *x*:*A*′ ==> *B*(*x*)=*B*′(*x*) |] ==>
QSigma(*A*,*B*) = *QSigma*(*A*′,*B*′)

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

lemma *QSigma-empty1* [*simp*]: *QSigma*(*0*,*B*) = *0*
by *blast*

lemma *QSigma-empty2* [*simp*]: *A* <∗> *0* = *0*
by *blast*

9.1.2 Projections: qfst, qsnd

lemma *qfst-conv* [*simp*]: *qfst*(<*a*;*b*>) = *a*
by (*simp add*: *qfst-def*)

lemma *qsnd-conv* [*simp*]: *qsnd*(<*a*;*b*>) = *b*
by (*simp add*: *qsnd-def*)

lemma *qfst-type* [*TC*]: *p*:*QSigma*(*A*,*B*) ==> *qfst*(*p*) : *A*
by *auto*

lemma *qsnd-type* [*TC*]: *p*:*QSigma*(*A*,*B*) ==> *qsnd*(*p*) : *B*(*qfst*(*p*))
by *auto*

lemma *QPair-qfst-qsnd-eq*: *a*: *QSigma*(*A*,*B*) ==> <*qfst*(*a*); *qsnd*(*a*)> = *a*
by *auto*

9.1.3 Eliminator: qsplit

lemma *qsplit* [*simp*]: *qsplit*(%*x y*. *c*(*x*,*y*), <*a*;*b*>) == *c*(*a*,*b*)
by (*simp add*: *qsplit-def*)

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

lemma *expand-qsplit*:
 u: *A*<∗>*B* ==> *R*(*qsplit*(*c*,*u*)) <−> (*ALL x*:*A*. *ALL y*:*B*. *u* = <*x*;*y*> −−> *R*(*c*(*x*,*y*)))
apply (*simp add*: *qsplit-def*, *auto*)
done

9.1.4 qsplit for predicates: result type o

lemma *qsplitI*: *R*(*a*,*b*) ==> *qsplit*(*R*, <*a*;*b*>)
by (*simp add*: *qsplit-def*)

lemma *qsplitE*:

[| qsplit(R,z); z:QSigma(A,B);
!!x y. [| z = <x;y>; R(x,y) |] ==> P
|] ==> P
by (simp add: qsplit-def, auto)

lemma qsplitD: qsplit(R,<a;b>) ==> R(a,b)
by (simp add: qsplit-def)

9.1.5 qconverse

lemma qconverseI [intro!]: <a;b>:r ==> <b;a>:qconverse(r)
by (simp add: qconverse-def, blast)

lemma qconverseD [elim!]: <a;b> : qconverse(r) ==> <b;a> : r
by (simp add: qconverse-def, blast)

lemma qconverseE [elim!]:
[| yx : qconverse(r);
!!x y. [| yx=<y;x>; <x;y>:r |] ==> P
|] ==> P
by (simp add: qconverse-def, blast)

lemma qconverse-qconverse: r<=QSigma(A,B) ==> qconverse(qconverse(r)) = r
by blast

lemma qconverse-type: r <= A <*> B ==> qconverse(r) <= B <*> A
by blast

lemma qconverse-prod: qconverse(A <*> B) = B <*> A
by blast

lemma qconverse-empty: qconverse(0) = 0
by blast

9.2 The Quine-inspired notion of disjoint sum

lemmas qsum-defs = qsum-def QInl-def QInr-def qcase-def

lemma QInlI [intro!]: a : A ==> QInl(a) : A <+> B
by (simp add: qsum-defs, blast)

lemma QInrI [intro!]: b : B ==> QInr(b) : A <+> B
by (simp add: qsum-defs, blast)

lemma qsumE [elim!]:

[| *u*: *A* <+> *B*;
  !!*x*. [| *x*:*A*; *u*=*QInl*(*x*) |] ==> *P*;
  !!*y*. [| *y*:*B*; *u*=*QInr*(*y*) |] ==> *P*
 |] ==> *P*
by (*simp add*: *qsum-defs*, *blast*)

lemma *QInl-iff* [*iff*]: *QInl*(*a*)=*QInl*(*b*) <−> *a*=*b*
by (*simp add*: *qsum-defs*)

lemma *QInr-iff* [*iff*]: *QInr*(*a*)=*QInr*(*b*) <−> *a*=*b*
by (*simp add*: *qsum-defs*)

lemma *QInl-QInr-iff* [*simp*]: *QInl*(*a*)=*QInr*(*b*) <−> *False*
by (*simp add*: *qsum-defs*)

lemma *QInr-QInl-iff* [*simp*]: *QInr*(*b*)=*QInl*(*a*) <−> *False*
by (*simp add*: *qsum-defs*)

lemma *qsum-empty* [*simp*]: *0*<+>*0* = *0*
by (*simp add*: *qsum-defs*)

lemmas *QInl-inject* = *QInl-iff* [*THEN iffD1*, *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*
by *blast*

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

lemma *qsum-iff*:
  u: *A* <+> *B* <−> (*EX x*. *x*:*A* & *u*=*QInl*(*x*)) | (*EX y*. *y*:*B* & *u*=*QInr*(*y*))
by *blast*

lemma *qsum-subset-iff*: *A* <+> *B* <= *C* <+> *D* <−> *A*<=*C* & *B*<=*D*
by *blast*

lemma *qsum-equal-iff*: *A* <+> *B* = *C* <+> *D* <−> *A*=*C* & *B*=*D*
apply (*simp* (*no-asm*) *add*: *extension qsum-subset-iff*)
apply *blast*

done

9.2.1 Eliminator – qcase

lemma *qcase-QInl* [*simp*]: *qcase*(*c*, *d*, *QInl*(*a*)) = *c*(*a*)
by (*simp add*: *qsum-defs*)

lemma *qcase-QInr* [*simp*]: *qcase*(*c*, *d*, *QInr*(*b*)) = *d*(*b*)
by (*simp add*: *qsum-defs*)

lemma *qcase-type*:
  [| *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*)
by (*simp add*: *qsum-defs*, *auto*)

lemma *Part-QInl*: *Part*(*A* <+> *B*,*QInl*) = {*QInl*(*x*). *x*: *A*}
by *blast*

lemma *Part-QInr*: *Part*(*A* <+> *B*,*QInr*) = {*QInr*(*y*). *y*: *B*}
by *blast*

lemma *Part-QInr2*: *Part*(*A* <+> *B*, %*x*. *QInr*(*h*(*x*))) = {*QInr*(*y*). *y*: *Part*(*B*,*h*)}
by *blast*

lemma *Part-qsum-equality*: *C* <= *A* <+> *B* ==> *Part*(*C*,*QInl*) *Un Part*(*C*,*QInr*) = *C*
by *blast*

9.2.2 Monotonicity

lemma *QPair-mono*: [| *a*<=*c*; *b*<=*d* |] ==> <*a*;*b*> <= <*c*;*d*>
by (*simp add*: *QPair-def sum-mono*)

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

lemma *QInl-mono*: *a*<=*b* ==> *QInl*(*a*) <= *QInl*(*b*)
by (*simp add*: *QInl-def subset-refl* [*THEN QPair-mono*])

lemma *QInr-mono*: *a*<=*b* ==> *QInr*(*a*) <= *QInr*(*b*)
by (*simp add*: *QInr-def subset-refl* [*THEN QPair-mono*])

lemma *qsum-mono*: [| *A*<=*C*; *B*<=*D* |] ==> *A* <+> *B* <= *C* <+> *D*
by *blast*

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*
apply (*unfold surj-def*)
apply (*erule CollectD1*)
done

lemma *fun-is-surj*: *f* : *Pi*(*A*,*B*) ==> *f*: *surj*(*A*,*range*(*f*))
apply (*unfold surj-def*)
apply (*blast intro*: *apply-equality range-of-fun domain-type*)
done

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

apply (*unfold surj-def*)
apply (*best intro*: *apply-Pair elim*: *range-type*)
done

lemma *f-imp-surjective*:
  [| *f*: *A*−>*B*; !!*y*. *y*:*B* ==> *d*(*y*): *A*; !!*y*. *y*:*B* ==> *f*‘*d*(*y*) = *y* |]
   ==> *f*: *surj*(*A*,*B*)
apply (*simp add*: *surj-def*, *blast*)
done

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*)
apply (*rule-tac d* = *d* **in** *f-imp-surjective*)
apply (*simp-all add*: *lam-type*)
done

lemma *cantor-surj*: *f* ~: *surj*(*A*,*Pow*(*A*))
apply (*unfold surj-def*, *safe*)
apply (*cut-tac cantor*)
apply (*best del*: *subsetI*)
done

10.2 Injections

lemma *inj-is-fun*: *f*: *inj*(*A*,*B*) ==> *f*: *A*−>*B*
apply (*unfold inj-def*)
apply (*erule CollectD1*)
done

lemma *inj-equality*:
  [| <*a*,*b*>:*f*; <*c*,*b*>:*f*; *f*: *inj*(*A*,*B*) |] ==> *a*=*c*
apply (*unfold inj-def*)
apply (*blast dest*: *Pair-mem-PiD*)
done

lemma *inj-apply-equality*: [| *f*:*inj*(*A*,*B*); *f*‘*a*=*f*‘*b*; *a*:*A*; *b*:*A* |] ==> *a*=*b*
by (*unfold inj-def*, *blast*)

lemma *f-imp-injective*: [| *f*: *A*−>*B*; *ALL x*:*A*. *d*(*f*‘*x*)=*x* |] ==> *f*: *inj*(*A*,*B*)

apply (*simp* (*no-asm-simp*) *add*: *inj-def*)
apply (*blast intro*: *subst-context* [*THEN box-equals*])
done

lemma *lam-injective*:
[| !!*x*. *x*:*A* ==> *c*(*x*): *B*;
!!*x*. *x*:*A* ==> *d*(*c*(*x*)) = *x* |]
==> (*lam x*:*A*. *c*(*x*)) : *inj*(*A*,*B*)
apply (*rule-tac d* = *d* **in** *f-imp-injective*)
apply (*simp-all add*: *lam-type*)
done

10.3 Bijections

lemma *bij-is-inj*: *f*: *bij*(*A*,*B*) ==> *f*: *inj*(*A*,*B*)
apply (*unfold bij-def*)
apply (*erule IntD1*)
done

lemma *bij-is-surj*: *f*: *bij*(*A*,*B*) ==> *f*: *surj*(*A*,*B*)
apply (*unfold bij-def*)
apply (*erule IntD2*)
done

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*)
apply (*unfold bij-def*)
apply (*blast intro*!: *lam-injective lam-surjective*)
done

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*)
apply (*rule-tac d* = *f* **in** *lam-bijective*)
apply (*auto simp add*: *the-equality2*)
done

10.4 Identity Function

lemma *idI* [*intro*!]: *a*:*A* ==> <*a*,*a*> : *id*(*A*)
apply (*unfold id-def*)
apply (*erule lamI*)
done

lemma *idE* [*elim*!]: [| *p*: *id*(*A*); !!*x*.[| *x*:*A*; *p*=<*x*,*x*> |] ==> *P* |] ==> *P*
by (*simp add*: *id-def lam-def*, *blast*)

lemma *id-type*: *id*(*A*) : *A*−>*A*
apply (*unfold id-def*)
apply (*rule lam-type*, *assumption*)
done

lemma *id-conv* [*simp*]: *x*:*A* ==> *id*(*A*)‘*x* = *x*
apply (*unfold id-def*)
apply (*simp* (*no-asm-simp*))
done

lemma *id-mono*: *A*<=*B* ==> *id*(*A*) <= *id*(*B*)
apply (*unfold id-def*)
apply (*erule lam-mono*)
done

lemma *id-subset-inj*: *A*<=*B* ==> *id*(*A*): *inj*(*A*,*B*)
apply (*simp add*: *inj-def id-def*)
apply (*blast intro*: *lam-type*)
done

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

lemma *id-surj*: *id*(*A*): *surj*(*A*,*A*)
apply (*unfold id-def surj-def*)
apply (*simp* (*no-asm-simp*))
done

lemma *id-bij*: *id*(*A*): *bij*(*A*,*A*)
apply (*unfold bij-def*)
apply (*blast intro*: *id-inj id-surj*)
done

lemma *subset-iff-id*: *A* <= *B* <−> *id*(*A*) : *A*−>*B*
apply (*unfold id-def*)
apply (*force intro*!: *lam-type dest*: *apply-type*)
done

id as the identity relation

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

10.5 Converse of a Function

lemma *inj-converse-fun*: *f*: *inj*(*A*,*B*) ==> *converse*(*f*) : *range*(*f*)−>*A*
apply (*unfold inj-def*)
apply (*simp* (*no-asm-simp*) *add*: *Pi-iff function-def*)

apply (*erule CollectE*)
apply (*simp* (*no-asm-simp*) *add*: *apply-iff*)
apply (*blast dest*: *fun-is-rel*)
done

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*
by (*blast intro*: *apply-Pair apply-equality converseI*)

lemma *left-inverse* [*simp*]: [| *f*: *inj*(*A*,*B*); *a*: *A* |] ==> *converse*(*f*)‘(*f*‘*a*) = *a*
by (*blast intro*: *left-inverse-lemma inj-converse-fun inj-is-fun*)

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

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*
by (*rule apply-Pair* [*THEN converseD* [*THEN apply-equality*]], *auto*)

lemma *right-inverse* [*simp*]:
[| *f*: *inj*(*A*,*B*); *b*: *range*(*f*) |] ==> *f*‘(*converse*(*f*)‘*b*) = *b*
by (*blast intro*: *right-inverse-lemma inj-converse-fun inj-is-fun*)

lemma *right-inverse-bij*: [| *f*: *bij*(*A*,*B*); *b*: *B* |] ==> *f*‘(*converse*(*f*)‘*b*) = *b*
by (*force simp add*: *bij-def surj-range*)

10.6 Converses of Injections, Surjections, Bijections

lemma *inj-converse-inj*: *f*: *inj*(*A*,*B*) ==> *converse*(*f*): *inj*(*range*(*f*), *A*)
apply (*rule f-imp-injective*)
apply (*erule inj-converse-fun*, *clarify*)
apply (*rule right-inverse*)
apply *assumption*
apply *blast*
done

lemma *inj-converse-surj*: *f*: *inj*(*A*,*B*) ==> *converse*(*f*): *surj*(*range*(*f*), *A*)
by (*blast intro*: *f-imp-surjective inj-converse-fun left-inverse inj-is-fun*
range-of-fun [*THEN apply-type*])

lemma *bij-converse-bij* [*TC*]: *f*: *bij*(*A*,*B*) ==> *converse*(*f*): *bij*(*B*,*A*)
apply (*unfold bij-def*)
apply (*fast elim*: *surj-range* [*THEN subst*] *inj-converse-inj inj-converse-surj*)

done

10.7 Composition of Two Relations

lemma *compI* [*intro*]: [| <a,b>:s; <b,c>:r |] ==> <a,c> : r O s
by (*unfold comp-def*, *blast*)

lemma *compE* [*elim*!]:
  [| xz : r O s;
    !!x y z. [| xz=<x,z>; <x,y>:s; <y,z>:r |] ==> P |]
  ==> P
by (*unfold comp-def*, *blast*)

lemma *compEpair*:
  [| <a,c> : r O s;
    !!y. [| <a,y>:s; <y,c>:r |] ==> P |]
  ==> P
by (*erule compE*, *simp*)

lemma *converse-comp*: converse(R O S) = converse(S) O converse(R)
by *blast*

10.8 Domain and Range – see Suppes, Section 3.1

lemma *range-comp*: range(r O s) <= range(r)
by *blast*

lemma *range-comp-eq*: domain(r) <= range(s) ==> range(r O s) = range(r)
by (*rule range-comp* [*THEN equalityI*], *blast*)

lemma *domain-comp*: domain(r O s) <= domain(s)
by *blast*

lemma *domain-comp-eq*: range(s) <= domain(r) ==> domain(r O s) = domain(s)
by (*rule domain-comp* [*THEN equalityI*], *blast*)

lemma *image-comp*: (r O s)``A = r``(s``A)
by *blast*

10.9 Other Results

lemma *comp-mono*: [| r'<=r; s'<=s |] ==> (r' O s') <= (r O s)
by *blast*

lemma *comp-rel*: [| s<=A*B; r<=B*C |] ==> (r O s) <= A*C
by *blast*

lemma *comp-assoc*: (*r O s*) *O t* = *r O* (*s O t*)
by *blast*

lemma *left-comp-id*: *r*<=*A***B* ==> *id*(*B*) *O r* = *r*
by *blast*

lemma *right-comp-id*: *r*<=*A***B* ==> *r O id*(*A*) = *r*
by *blast*

10.10 Composition Preserves Functions, Injections, and Surjections

lemma *comp-function*: [| *function*(*g*); *function*(*f*) |] ==> *function*(*f O g*)
by (*unfold function-def*, *blast*)

lemma *comp-fun*: [| *g*: *A*−>*B*; *f*: *B*−>*C* |] ==> (*f O g*) : *A*−>*C*
apply (*auto simp add*: *Pi-def comp-function Pow-iff comp-rel*)
apply (*subst range-rel-subset* [*THEN domain-comp-eq*], *auto*)
done

lemma *comp-fun-apply* [*simp*]:
    [| *g*: *A*−>*B*; *a*:*A* |] ==> (*f O g*)‘*a* = *f*‘(*g*‘*a*)
apply (*frule apply-Pair*, *assumption*)
apply (*simp add*: *apply-def image-comp*)
apply (*blast dest*: *apply-equality*)
done

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*)))
apply (*subgoal-tac* (*lam x*:*A*. *b*(*x*)) : *A* −> *B*)
 apply (*rule fun-extension*)
   apply (*blast intro*: *comp-fun lam-funtype*)
  apply (*rule lam-funtype*)
 apply *simp*
apply (*simp add*: *lam-type*)
done

lemma *comp-inj*:
    [| *g*: *inj*(*A*,*B*); *f*: *inj*(*B*,*C*) |] ==> (*f O g*) : *inj*(*A*,*C*)
apply (*frule inj-is-fun* [*of g*])
apply (*frule inj-is-fun* [*of f*])
apply (*rule-tac d* = %*y*. *converse* (*g*) ‘ (*converse* (*f*) ‘ *y*) **in** *f-imp-injective*)
 apply (*blast intro*: *comp-fun*, *simp*)

done

lemma *comp-surj*:
  [| *g*: *surj*(*A*,*B*); *f*: *surj*(*B*,*C*) |] ==> (*f O g*) : *surj*(*A*,*C*)
apply (*unfold surj-def*)
apply (*blast intro*!: *comp-fun comp-fun-apply*)
done

lemma *comp-bij*:
  [| *g*: *bij*(*A*,*B*); *f*: *bij*(*B*,*C*) |] ==> (*f O g*) : *bij*(*A*,*C*)
apply (*unfold bij-def*)
apply (*blast intro*: *comp-inj comp-surj*)
done

10.11 Dual Properties of *inj* and *surj*

Useful for proofs from D Pastre. Automatic theorem proving in set theory. Artificial Intelligence, 10:1–27, 1978.

lemma *comp-mem-injD1*:
  [| (*f O g*): *inj*(*A*,*C*); *g*: *A*−>*B*; *f*: *B*−>*C* |] ==> *g*: *inj*(*A*,*B*)
by (*unfold inj-def*, *force*)

lemma *comp-mem-injD2*:
  [| (*f O g*): *inj*(*A*,*C*); *g*: *surj*(*A*,*B*); *f*: *B*−>*C* |] ==> *f*: *inj*(*B*,*C*)
apply (*unfold inj-def surj-def*, *safe*)
apply (*rule-tac x1* = *x* **in** *bspec* [*THEN bexE*])
apply (*erule-tac* [*3*] *x1* = *w* **in** *bspec* [*THEN bexE*], *assumption*+, *safe*)
apply (*rule-tac t* = *op* ‘ (*g*) **in** *subst-context*)
apply (*erule asm-rl bspec* [*THEN bspec*, *THEN mp*])+
apply (*simp* (*no-asm-simp*))
done

lemma *comp-mem-surjD1*:
  [| (*f O g*): *surj*(*A*,*C*); *g*: *A*−>*B*; *f*: *B*−>*C* |] ==> *f*: *surj*(*B*,*C*)
apply (*unfold surj-def*)
apply (*blast intro*!: *comp-fun-apply* [*symmetric*] *apply-funtype*)
done

lemma *comp-mem-surjD2*:
  [| (*f O g*): *surj*(*A*,*C*); *g*: *A*−>*B*; *f*: *inj*(*B*,*C*) |] ==> *g*: *surj*(*A*,*B*)
apply (*unfold inj-def surj-def*, *safe*)
apply (*drule-tac x* = *f*‘*y* **in** *bspec*, *auto*)
apply (*blast intro*: *apply-funtype*)
done

10.11.1 Inverses of Composition

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

```
apply (unfold inj-def, clarify)
apply (rule equalityI)
 apply (auto simp add: apply-iff, blast)
done
```

```
lemma right-comp-inverse:
    f: surj(A,B) ==> f O converse(f) = id(B)
apply (simp add: surj-def, clarify)
apply (rule equalityI)
apply (best elim: domain-type range-type dest: apply-equality2)
apply (blast intro: apply-Pair)
done
```

10.11.2 Proving that a Function is a Bijection

```
lemma comp-eq-id-iff:
    [| f: A->B; g: B->A |] ==> f O g = id(B) <-> (ALL y:B. f`(g`y)=y)
apply (unfold id-def, safe)
 apply (drule-tac t = %h. h`y in subst-context)
 apply simp
apply (rule fun-extension)
  apply (blast intro: comp-fun lam-type)
 apply auto
done
```

```
lemma fg-imp-bijective:
    [| f: A->B; g: B->A; f O g = id(B); g O f = id(A) |] ==> f : bij(A,B)
apply (unfold bij-def)
apply (simp add: comp-eq-id-iff)
apply (blast intro: f-imp-injective f-imp-surjective apply-funtype)
done
```

```
lemma nilpotent-imp-bijective: [| f: A->A; f O f = id(A) |] ==> f : bij(A,A)
by (blast intro: fg-imp-bijective)
```

```
lemma invertible-imp-bijective:
     [| converse(f): B->A; f: A->B |] ==> f : bij(A,B)
by (simp add: fg-imp-bijective comp-eq-id-iff
              left-inverse-lemma right-inverse-lemma)
```

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)
apply (rule-tac d = %z. if z:B then converse (f) `z else converse (g) `z
       in lam-injective)
```

apply (*auto simp add*: *inj-is-fun* [*THEN apply-type*])
done

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*)
apply (*simp add*: *surj-def fun-disjoint-Un*)
apply (*blast dest*!: *domain-of-fun*
        intro!: *fun-disjoint-apply1 fun-disjoint-apply2*)
done

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*)
apply (*rule invertible-imp-bijective*)
apply (*subst converse-Un*)
apply (*auto intro*: *fun-disjoint-Un bij-is-fun bij-converse-bij*)
done

10.11.4 Restrictions as Surjections and Bijections

lemma *surj-image*:
  f: *Pi*(*A*,*B*) ==> *f*: *surj*(*A*, *f*``*A*)
apply (*simp add*: *surj-def*)
apply (*blast intro*: *apply-equality apply-Pair Pi-type*)
done

lemma *restrict-image* [*simp*]: *restrict*(*f*,*A*) `` *B* = *f* `` (*A Int B*)
by (*auto simp add*: *restrict-def*)

lemma *restrict-inj*:
  [| *f*: *inj*(*A*,*B*); *C*<=*A* |] ==> *restrict*(*f*,*C*): *inj*(*C*,*B*)
apply (*unfold inj-def*)
apply (*safe elim*!: *restrict-type2*, *auto*)
done

lemma *restrict-surj*: [| *f*: *Pi*(*A*,*B*); *C*<=*A* |] ==> *restrict*(*f*,*C*): *surj*(*C*, *f*``*C*)
apply (*insert restrict-type2* [*THEN surj-image*])
apply (*simp add*: *restrict-image*)
done

lemma *restrict-bij*:
  [| *f*: *inj*(*A*,*B*); *C*<=*A* |] ==> *restrict*(*f*,*C*): *bij*(*C*, *f*``*C*)
apply (*simp add*: *inj-def bij-def*)
apply (*blast intro*: *restrict-surj surj-is-fun*)
done

10.11.5 Lemmas for Ramsey's Theorem

lemma *inj-weaken-type*: [| *f*: *inj*(*A*,*B*); *B*<=*D* |] ==> *f*: *inj*(*A*,*D*)
apply (*unfold inj-def*)
apply (*blast intro*: *fun-weaken-type*)
done

lemma *inj-succ-restrict*:
[| *f*: *inj*(*succ*(*m*), *A*) |] ==> *restrict*(*f*,*m*) : *inj*(*m*, *A*−{*f*‘*m*})
apply (*rule restrict-bij* [*THEN bij-is-inj*, *THEN inj-weaken-type*], *assumption*, *blast*)
apply (*unfold inj-def*)
apply (*fast elim*: *range-type mem-irrefl dest*: *apply-equality*)
done

lemma *inj-extend*:
[| *f*: *inj*(*A*,*B*); *a*~:*A*; *b*~:*B* |]
==> *cons*(<*a*,*b*>,*f*) : *inj*(*cons*(*a*,*A*), *cons*(*b*,*B*))
apply (*unfold inj-def*)
apply (*force intro*: *apply-type simp add*: *fun-extend*)
done

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*> ~: *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* −−> ~ <*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^+ == r O r^*

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> ~: r |] ==> irrefl(A,r)
by (simp add: irrefl-def)

lemma irreflE: [| irrefl(A,r); x:A |] ==> <x,x> ~: r
by (simp add: irrefl-def)

11.1.2 symmetry

lemma symI:
[| !!x y.<x,y>: r ==> <y,x>: r |] ==> sym(r)
by (unfold sym-def, blast)

lemma symE: [| sym(r); <x,y>: r |] ==> <y,x>: r
by (unfold sym-def, blast)

11.1.3 antisymmetry

lemma antisymI:
[| !!x y.[| <x,y>: r; <y,x>: r |] ==> x=y |] ==> antisym(r)
by (simp add: antisym-def, blast)

lemma antisymE: [| antisym(r); <x,y>: r; <y,x>: r |] ==> x=y

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

11.1.4 transitivity

lemma *transD*: [| *trans*(*r*); <*a*,*b*>:*r*; <*b*,*c*>:*r* |] ==> <*a*,*c*>:*r*
by (*unfold trans-def*, *blast*)

lemma *trans-onD*:
[| *trans*[*A*](*r*); <*a*,*b*>:*r*; <*b*,*c*>:*r*; *a*:*A*; *b*:*A*; *c*:*A* |] ==> <*a*,*c*>:*r*
by (*unfold trans-on-def*, *blast*)

lemma *trans-imp-trans-on*: *trans*(*r*) ==> *trans*[*A*](*r*)
by (*unfold trans-def trans-on-def*, *blast*)

lemma *trans-on-imp-trans*: [|*trans*[*A*](*r*); *r* <= *A***A*|] ==> *trans*(*r*)
by (*simp add*: *trans-on-def trans-def*, *blast*)

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*))
by (*rule bnd-monoI*, *blast*+)

lemma *rtrancl-mono*: *r*<=*s* ==> *r*^* <= *s*^*
apply (*unfold rtrancl-def*)
apply (*rule lfp-mono*)
apply (*rule rtrancl-bnd-mono*)+
apply *blast*
done

lemmas *rtrancl-unfold* =
rtrancl-bnd-mono [*THEN rtrancl-def* [*THEN def-lfp-unfold*], *standard*]

lemmas *rtrancl-type* = *rtrancl-def* [*THEN def-lfp-subset*, *standard*]

lemma *relation-rtrancl*: *relation*(*r*^*)
apply (*simp add*: *relation-def*)
apply (*blast dest*: *rtrancl-type* [*THEN subsetD*])
done

lemma *rtrancl-refl*: [| *a*: *field*(*r*) |] ==> <*a*,*a*> : *r*^*
apply (*rule rtrancl-unfold* [*THEN ssubst*])
apply (*erule idI* [*THEN UnI1*])
done

lemma *rtrancl-into-rtrancl*: [| <*a*,*b*> : *r*^∗; <*b*,*c*> : *r* |] ==> <*a*,*c*> : *r*^∗
apply (*rule rtrancl-unfold* [*THEN ssubst*])
apply (*rule compI* [*THEN UnI2*], *assumption*, *assumption*)
done

lemma *r-into-rtrancl*: <*a*,*b*> : *r* ==> <*a*,*b*> : *r*^∗
by (*rule rtrancl-refl* [*THEN rtrancl-into-rtrancl*], *blast*+)

lemma *r-subset-rtrancl*: *relation*(*r*) ==> *r* <= *r*^∗
by (*simp add*: *relation-def*, *blast intro*: *r-into-rtrancl*)

lemma *rtrancl-field*: *field*(*r*^∗) = *field*(*r*)
by (*blast intro*: *r-into-rtrancl dest*!: *rtrancl-type* [*THEN subsetD*])

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*>)
by (*erule def-induct* [*OF rtrancl-def rtrancl-bnd-mono*], *blast*)

lemma *rtrancl-induct* [*case-names initial step*, *induct set*: *rtrancl*]:
  [| <*a*,*b*> : *r*^∗;
      P(*a*);
      !!*y z*.[| <*a*,*y*> : *r*^∗; <*y*,*z*> : *r*; *P*(*y*) |] ==> *P*(*z*)
   |] ==> *P*(*b*)

apply (*subgoal-tac ALL y*. <*a*,*b*> = <*a*,*y*> −−> *P* (*y*))

apply (*erule spec* [*THEN mp*], *rule refl*)

apply (*erule rtrancl-full-induct*, *blast*+)
done

lemma *trans-rtrancl*: *trans*(*r*^∗)
apply (*unfold trans-def*)
apply (*intro allI impI*)
apply (*erule-tac b* = *z* **in** *rtrancl-induct*, *assumption*)
apply (*blast intro*: *rtrancl-into-rtrancl*)
done

lemmas rtrancl-trans = trans-rtrancl [THEN transD, standard]

lemma rtranclE:
[| <a,b> : r^*; (a=b) ==> P;
!!y.[| <a,y> : r^*; <y,b> : r |] ==> P |]
==> P
apply (subgoal-tac a = b | (EX y. <a,y> : r^* & <y,b> : r))

apply blast
apply (erule rtrancl-induct, blast+)
done

lemma trans-trancl: trans(r^+)
apply (unfold trans-def trancl-def)
apply (blast intro: rtrancl-into-rtrancl
trans-rtrancl [THEN transD, THEN compI])
done

lemmas trans-on-trancl = trans-trancl [THEN trans-imp-trans-on]

lemmas trancl-trans = trans-trancl [THEN transD, standard]

lemma trancl-into-rtrancl: <a,b> : r^+ ==> <a,b> : r^*
apply (unfold trancl-def)
apply (blast intro: rtrancl-into-rtrancl)
done

lemma r-into-trancl: <a,b> : r ==> <a,b> : r^+
apply (unfold trancl-def)
apply (blast intro!: rtrancl-refl)
done

lemma r-subset-trancl: relation(r) ==> r <= r^+
by (simp add: relation-def, blast intro: r-into-trancl)

lemma rtrancl-into-trancl1: [| <a,b> : r^*; <b,c> : r |] ==> <a,c> : r^+
by (unfold trancl-def, blast)

```
lemma rtrancl-into-trancl2:
    [| <a,b> : r;  <b,c> : r^* |]   ==>  <a,c> : r^+
apply (erule rtrancl-induct)
 apply (erule r-into-trancl)
apply (blast intro: r-into-trancl trancl-trans)
done


lemma trancl-induct [case-names initial step, induct set: trancl]:
  [| <a,b> : r^+;
      !!y.  [| <a,y> : r |] ==> P(y);
      !!y z.[| <a,y> : r^+;  <y,z> : r;  P(y) |] ==> P(z)
   |] ==> P(b)
apply (rule compEpair)
apply (unfold trancl-def, assumption)

apply (subgoal-tac ALL z. <y,z> : r --> P (z) )

 apply blast
apply (erule rtrancl-induct)
apply (blast intro: rtrancl-into-trancl1)+
done


lemma tranclE:
    [| <a,b> : r^+;
        <a,b> : r ==> P;
        !!y.[| <a,y> : r^+; <y,b> : r |] ==> P
     |] ==> P
apply (subgoal-tac <a,b> : r | (EX y. <a,y> : r^+ & <y,b> : r) )
apply blast
apply (rule compEpair)
apply (unfold trancl-def, assumption)
apply (erule rtranclE)
apply (blast intro: rtrancl-into-trancl1)+
done

lemma trancl-type: r^+ <= field(r)*field(r)
apply (unfold trancl-def)
apply (blast elim: rtrancl-type [THEN subsetD, THEN SigmaE2])
done

lemma relation-trancl: relation(r^+)
apply (simp add: relation-def)
apply (blast dest: trancl-type [THEN subsetD])
done

lemma trancl-subset-times: r ⊆ A * A ==> r^+ ⊆ A * A
```

by (*insert trancl-type* [*of r*], *blast*)

lemma *trancl-mono*: *r<=s ==> r^+ <= s^+*
by (*unfold trancl-def*, *intro comp-mono rtrancl-mono*)

lemma *trancl-eq-r*: [|*relation*(*r*); *trans*(*r*)|] ==> *r^+ = r*
apply (*rule equalityI*)
 prefer *2* **apply** (*erule r-subset-trancl*, *clarify*)
apply (*frule trancl-type* [*THEN subsetD*], *clarify*)
apply (*erule trancl-induct*, *assumption*)
apply (*blast dest*: *transD*)
done

lemma *rtrancl-idemp* [*simp*]: (*r^**)^* = *r^**
apply (*rule equalityI*, *auto*)
 prefer *2*
 apply (*frule rtrancl-type* [*THEN subsetD*])
 apply (*blast intro*: *r-into-rtrancl*)

converse direction

apply (*frule rtrancl-type* [*THEN subsetD*], *clarify*)
apply (*erule rtrancl-induct*)
apply (*simp add*: *rtrancl-refl rtrancl-field*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *rtrancl-subset*: [| *R* <= *S*; *S* <= *R^** |] ==> *S^** = *R^**
apply (*drule rtrancl-mono*)
apply (*drule rtrancl-mono*, *simp-all*, *blast*)
done

lemma *rtrancl-Un-rtrancl*:
    [| *relation*(*r*); *relation*(*s*) |] ==> (*r^* Un s^**)^* = (*r Un s*)^*
apply (*rule rtrancl-subset*)
apply (*blast dest*: *r-subset-rtrancl*)
apply (*blast intro*: *rtrancl-mono* [*THEN subsetD*])
done

lemma *rtrancl-converseD*: <*x*,*y*>:*converse*(*r*)^* ==> <*x*,*y*>:*converse*(*r^**)
apply (*rule converseI*)
apply (*frule rtrancl-type* [*THEN subsetD*])
apply (*erule rtrancl-induct*)

apply (*blast intro*: *rtrancl-refl*)
apply (*blast intro*: *r-into-rtrancl rtrancl-trans*)
done

lemma *rtrancl-converseI*: <*x*,*y*>:*converse*(*r*^*) ==> <*x*,*y*>:*converse*(*r*)^*
apply (*drule converseD*)
apply (*frule rtrancl-type* [*THEN subsetD*])
apply (*erule rtrancl-induct*)
apply (*blast intro*: *rtrancl-refl*)
apply (*blast intro*: *r-into-rtrancl rtrancl-trans*)
done

lemma *rtrancl-converse*: *converse*(*r*)^* = *converse*(*r*^*)
apply (*safe intro*!: *equalityI*)
apply (*frule rtrancl-type* [*THEN subsetD*])
apply (*safe dest*!: *rtrancl-converseD intro*!: *rtrancl-converseI*)
done

lemma *trancl-converseD*: <*a*, *b*>:*converse*(*r*)^+ ==> <*a*, *b*>:*converse*(*r*^+)
apply (*erule trancl-induct*)
apply (*auto intro*: *r-into-trancl trancl-trans*)
done

lemma *trancl-converseI*: <*x*,*y*>:*converse*(*r*^+) ==> <*x*,*y*>:*converse*(*r*)^+
apply (*drule converseD*)
apply (*erule trancl-induct*)
apply (*auto intro*: *r-into-trancl trancl-trans*)
done

lemma *trancl-converse*: *converse*(*r*)^+ = *converse*(*r*^+)
apply (*safe intro*!: *equalityI*)
apply (*frule trancl-type* [*THEN subsetD*])
apply (*safe dest*!: *trancl-converseD intro*!: *trancl-converseI*)
done

lemma *converse-trancl-induct* [*case-names initial step*, *consumes 1*]:
[| <*a*, *b*>:*r*^+; !!*y*. <*y*, *b*> :*r* ==> *P*(*y*);
!!*y z*. [| <*y*, *z*> : *r*; <*z*, *b*> : *r*^+; *P*(*z*) |] ==> *P*(*y*) |]
==> *P*(*a*)
apply (*drule converseI*)
apply (*simp* (*no-asm-use*) *add*: *trancl-converse* [*symmetric*])
apply (*erule trancl-induct*)
apply (*auto simp add*: *trancl-converse*)
done

end

12 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)*
by (*unfold wf-def wf-on-def*, *force*)

lemma *wf-on-imp-wf*: *[|wf[A](r); r <= A*A|] ==> wf(r)*
by (*simp add*: *wf-on-def subset-Int-iff*)

lemma *wf-on-field-imp-wf*: *wf[field(r)](r) ==> wf(r)*
by (*unfold wf-def wf-on-def*, *fast*)

lemma *wf-iff-wf-on-field*: *wf(r) <-> wf[field(r)](r)*
by (*blast intro*: *wf-imp-wf-on wf-on-field-imp-wf*)

lemma *wf-on-subset-A*: [| *wf*[*A*](*r*); *B*<=*A* |] ==> *wf*[*B*](*r*)
by (*unfold wf-on-def wf-def*, *fast*)

lemma *wf-on-subset-r*: [| *wf*[*A*](*r*); *s*<=*r* |] ==> *wf*[*A*](*s*)
by (*unfold wf-on-def wf-def*, *fast*)

lemma *wf-subset*: [|*wf*(*s*); *r*<=*s*|] ==> *wf*(*r*)
by (*simp add*: *wf-def*, *fast*)

12.1.2 Introduction Rules for *wf-on*

If every non-empty subset of A has an r-minimal element then we have *wf*[*A*](*r*).

lemma *wf-onI*:
 assumes *prem*: !!*Z u*. [| *Z*<=*A*; *u*:*Z*; *ALL x*:*Z*. *EX y*:*Z*. <*y*,*x*>:*r* |] ==> *False*
 shows *wf*[*A*](*r*)
apply (*unfold wf-on-def wf-def*)
apply (*rule equals0I* [*THEN disjCI*, *THEN allI*])
apply (*rule-tac Z* = *Z* **in** *prem*, *blast*+)
done

If r allows well-founded induction over A then we have *wf*[*A*](*r*). Premise is equivalent to $\bigwedge B.\ \forall x{\in}A.\ (\forall y.\ \langle y,\ x\rangle \in r \longrightarrow y \in B) \longrightarrow x \in B \Longrightarrow A \subseteq B$

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*)
apply (*rule wf-onI*)
apply (*rule-tac c*=*u* **in** *prem* [*THEN DiffE*])
 prefer *3* **apply** *blast*
 apply *fast*+
done

12.1.3 Well-founded Induction

Consider the least z in $domain(r)$ such that $P(z)$ does not hold...

lemma *wf-induct* [*induct set*: *wf*]:
 [| *wf*(*r*);
 !!*x*.[| *ALL y*. <*y*,*x*>: *r* --> *P*(*y*) |] ==> *P*(*x*) |]
 ==> *P*(*a*)
apply (*unfold wf-def*)
apply (*erule-tac x* = {*z* : *domain*(*r*). ~ *P*(*z*)} **in** *allE*)
apply *blast*
done

```
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)
apply (erule-tac P=a:A in rev-mp)
apply (erule-tac a=a in wf-induct, blast)
done

lemma field-Int-square: field(r Int A*A) <= A
by blast

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)
apply (unfold wf-on-def)
apply (erule wf-induct2, assumption)
apply (rule field-Int-square, blast)
done

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)
apply (rule wf-on-subset-A [THEN wf-on-field-imp-wf])
apply (rule wf-onI2)
 prefer 2 apply blast
apply blast
done
```

12.2 Basic Properties of Well-Founded Relations

```
lemma wf-not-refl: wf(r) ==> <a,a> ~: r
by (erule-tac a=a in wf-induct, blast)

lemma wf-not-sym [rule-format]: wf(r) ==> ALL x. <a,x>:r --> <x,a> ~: r
by (erule-tac a=a in wf-induct, blast)


lemmas wf-asym = wf-not-sym [THEN swap, standard]
```

lemma *wf-on-not-refl*: [| wf[A](r); a: A |] ==> <a,a> ~: r
by (erule-tac a=a **in** wf-on-induct, assumption, blast)

lemma *wf-on-not-sym* [rule-format]:
[| wf[A](r); a:A |] ==> ALL b:A. <a,b>:r --> <b,a>~:r
apply (erule-tac a=a **in** wf-on-induct, assumption, blast)
done

lemma *wf-on-asym*:
[| wf[A](r); ~Z ==> <a,b> : r;
<b,a> ~: r ==> Z; ~Z ==> a : A; ~Z ==> b : A |] ==> Z
by (blast dest: wf-on-not-sym)

lemma *wf-on-chain3*:
[| wf[A](r); <a,b>:r; <b,c>:r; <c,a>:r; a:A; b:A; c:A |] ==> P
apply (subgoal-tac ALL y:A. ALL z:A. <a,y>:r --> <y,z>:r --> <z,a>:r
--> P,
blast)
apply (erule-tac a=a **in** wf-on-induct, assumption, blast)
done

transitive closure of a WF relation is WF provided *A* is downward closed

lemma *wf-on-trancl*:
[| wf[A](r); r-``A <= A |] ==> wf[A](r^+)
apply (rule wf-onI2)
apply (frule bspec [THEN mp], assumption+)
apply (erule-tac a = y **in** wf-on-induct, assumption)
apply (blast elim: tranclE, blast)
done

lemma *wf-trancl*: wf(r) ==> wf(r^+)
apply (simp add: wf-iff-wf-on-field)
apply (rule wf-on-subset-A)
apply (erule wf-on-trancl)
apply blast
apply (rule trancl-type [THEN field-rel-subset])
done

r -`` {*a*} is the set of everything under *a* in *r*

lemmas *underI* = vimage-singleton-iff [THEN iffD2, 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)
apply (unfold is-recfun-def)
apply (erule ssubst)

apply (*rule lamI* [*THEN rangeI*, *THEN lam-type*], *assumption*)
done

lemmas *is-recfun-imp-function* = *is-recfun-type* [*THEN fun-is-function*]

lemma *apply-recfun*:
[| *is-recfun*(*r*,*a*,*H*,*f*); <*x*,*a*>:*r* |] ==> *f*‘*x* = *H*(*x*, *restrict*(*f*,*r*−‘‘{*x*}))
apply (*unfold is-recfun-def*)

replace f only on the left-hand side

apply (*erule-tac P* = %*x*.?*t*(*x*) = ?*u* **in** *ssubst*)
apply (*simp add*: *underI*)
done

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*
apply (*frule-tac f* = *f* **in** *is-recfun-type*)
apply (*frule-tac f* = *g* **in** *is-recfun-type*)
apply (*simp add*: *is-recfun-def*)
apply (*erule-tac a*=*x* **in** *wf-induct*)
apply (*intro impI*)
apply (*elim ssubst*)
apply (*simp* (*no-asm-simp*) *add*: *vimage-singleton-iff restrict-def*)
apply (*rule-tac t* = %*z*. *H* (?*x*,*z*) **in** *subst-context*)
apply (*subgoal-tac ALL y* : *r*−‘‘{*x*}. *ALL z*. <*y*,*z*>:*f* <−> <*y*,*z*>:*g*)
apply (*blast dest*: *transD*)
apply (*simp add*: *apply-iff*)
apply (*blast dest*: *transD intro*: *sym*)
done

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*
apply (*frule-tac f* = *f* **in** *is-recfun-type*)
apply (*rule fun-extension*)
apply (*blast dest*: *transD intro*: *restrict-type2*)
apply (*erule is-recfun-type*, *simp*)
apply (*blast dest*: *transD intro*: *is-recfun-equal*)
done

12.4 Recursion: Main Existence Lemma

lemma *is-recfun-functional*:
[| *wf*(*r*); *trans*(*r*); *is-recfun*(*r*,*a*,*H*,*f*); *is-recfun*(*r*,*a*,*H*,*g*) |] ==> *f*=*g*
by (*blast intro*: *fun-extension is-recfun-type is-recfun-equal*)

lemma *the-recfun-eq*:
[| *is-recfun*(*r*,*a*,*H*,*f*); *wf*(*r*); *trans*(*r*) |] ==> *the-recfun*(*r*,*a*,*H*) = *f*

apply (*unfold the-recfun-def*)
apply (*blast intro*: *is-recfun-functional*)
done

lemma *is-the-recfun*:
[| *is-recfun*(*r*,*a*,*H*,*f*); *wf*(*r*); *trans*(*r*) |]
==> *is-recfun*(*r*, *a*, *H*, *the-recfun*(*r*,*a*,*H*))
by (*simp add*: *the-recfun-eq*)

lemma *unfold-the-recfun*:
[| *wf*(*r*); *trans*(*r*) |] ==> *is-recfun*(*r*, *a*, *H*, *the-recfun*(*r*,*a*,*H*))
apply (*rule-tac a*=*a* **in** *wf-induct*, *assumption*)
apply (*rename-tac a1*)
apply (*rule-tac f* = *lam y*: *r*−‘‘{*a1*}. *wftrec* (*r*,*y*,*H*) **in** *is-the-recfun*)
apply *typecheck*
apply (*unfold is-recfun-def wftrec-def*)
— Applying the substitution: must keep the quantified assumption!
apply (*rule lam-cong* [*OF refl*])
apply (*drule underD*)
apply (*fold is-recfun-def*)
apply (*rule-tac t* = %*z*. *H*(*?x*,*z*) **in** *subst-context*)
apply (*rule fun-extension*)
apply (*blast intro*: *is-recfun-type*)
apply (*rule lam-type* [*THEN restrict-type2*])
apply *blast*
apply (*blast dest*: *transD*)
apply (*frule spec* [*THEN mp*], *assumption*)
apply (*subgoal-tac* <*xa*,*a1*> : *r*)
apply (*drule-tac x1* = *xa* **in** *spec* [*THEN mp*], *assumption*)
apply (*simp add*: *vimage-singleton-iff*
apply-recfun is-recfun-cut)
apply (*blast dest*: *transD*)
done

12.5 Unfolding *wftrec*(*r*, *a*, *H*)

lemma *the-recfun-cut*:
[| *wf*(*r*); *trans*(*r*); <*b*,*a*>:*r* |]
==> *restrict*(*the-recfun*(*r*,*a*,*H*), *r*−‘‘{*b*}) = *the-recfun*(*r*,*b*,*H*)
by (*blast intro*: *is-recfun-cut unfold-the-recfun*)

lemma *wftrec*:
[| *wf*(*r*); *trans*(*r*) |] ==>
wftrec(*r*,*a*,*H*) = *H*(*a*, *lam x*: *r*−‘‘{*a*}. *wftrec*(*r*,*x*,*H*))
apply (*unfold wftrec-def*)
apply (*subst unfold-the-recfun* [*unfolded is-recfun-def*])
apply (*simp-all add*: *vimage-singleton-iff* [*THEN iff-sym*] *the-recfun-cut*)

done

12.5.1 Removal of the Premise *trans*(*r*)

lemma *wfrec*:
wf(*r*) ==> *wfrec*(*r*,*a*,*H*) = *H*(*a*, *lam x*:*r*−‘‘{*a*}. *wfrec*(*r*,*x*,*H*))
apply (*unfold wfrec-def*)
apply (*erule wf-trancl* [*THEN wftrec*, *THEN ssubst*])
apply (*rule trans-trancl*)
apply (*rule vimage-pair-mono* [*THEN restrict-lam-eq*, *THEN subst-context*])
apply (*erule r-into-trancl*)
apply (*rule subset-refl*)
done

lemma *def-wfrec*:
[| !!*x*. *h*(*x*)==*wfrec*(*r*,*x*,*H*); *wf*(*r*) |] ==>
h(*a*) = *H*(*a*, *lam x*: *r*−‘‘{*a*}. *h*(*x*))
apply *simp*
apply (*elim wfrec*)
done

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*)
apply (*rule-tac a* = *a* **in** *wf-induct2*, *assumption*+)
apply (*subst wfrec*, *assumption*)
apply (*simp add*: *lam-type underD*)
done

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*))
apply (*unfold wf-on-def wfrec-on-def*)
apply (*erule wfrec* [*THEN trans*])
apply (*simp add*: *vimage-Int-square cons-subset-iff*)
done

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

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

lemma *Transset-iff-Union-succ*: *Transset(A) <-> Union(succ(A)) = A*
apply (*unfold Transset-def*)
apply (*blast elim!: equalityE*)
done

lemma *Transset-iff-Union-subset*: *Transset(A) <-> Union(A) <= A*
by (*unfold Transset-def*, *blast*)

13.1.2 Consequences of Downwards Closure

lemma *Transset-doubleton-D*:
  [| *Transset*(*C*); {*a*,*b*}: *C* |] ==> *a*:*C* & *b*: *C*
by (*unfold Transset-def*, *blast*)

lemma *Transset-Pair-D*:
  [| *Transset*(*C*); <*a*,*b*>: *C* |] ==> *a*:*C* & *b*: *C*
apply (*simp add*: *Pair-def*)
apply (*blast dest*: *Transset-doubleton-D*)
done

lemma *Transset-includes-domain*:
  [| *Transset*(*C*); *A***B* <= *C*; *b*: *B* |] ==> *A* <= *C*
by (*blast dest*: *Transset-Pair-D*)

lemma *Transset-includes-range*:
  [| *Transset*(*C*); *A***B* <= *C*; *a*: *A* |] ==> *B* <= *C*
by (*blast dest*: *Transset-Pair-D*)

13.1.3 Closure Properties

lemma *Transset-0*: *Transset*(*0*)
by (*unfold Transset-def*, *blast*)

lemma *Transset-Un*:
  [| *Transset*(*i*); *Transset*(*j*) |] ==> *Transset*(*i Un j*)
by (*unfold Transset-def*, *blast*)

lemma *Transset-Int*:
  [| *Transset*(*i*); *Transset*(*j*) |] ==> *Transset*(*i Int j*)
by (*unfold Transset-def*, *blast*)

lemma *Transset-succ*: *Transset*(*i*) ==> *Transset*(*succ*(*i*))
by (*unfold Transset-def*, *blast*)

lemma *Transset-Pow*: *Transset*(*i*) ==> *Transset*(*Pow*(*i*))
by (*unfold Transset-def*, *blast*)

lemma *Transset-Union*: *Transset*(*A*) ==> *Transset*(*Union*(*A*))
by (*unfold Transset-def*, *blast*)

lemma *Transset-Union-family*:
  [| !!*i*. *i*:*A* ==> *Transset*(*i*) |] ==> *Transset*(*Union*(*A*))
by (*unfold Transset-def*, *blast*)

lemma *Transset-Inter-family*:
  [| !!*i*. *i*:*A* ==> *Transset*(*i*) |] ==> *Transset*(*Inter*(*A*))
by (*unfold Inter-def Transset-def*, *blast*)

lemma *Transset-UN*:
  (!!x. x ∈ A ==> *Transset*(B(x))) ==> *Transset* (⋃x∈A. B(x))
by (*rule Transset-Union-family*, *auto*)

lemma *Transset-INT*:
  (!!x. x ∈ A ==> *Transset*(B(x))) ==> *Transset* (⋂x∈A. B(x))
by (*rule Transset-Inter-family*, *auto*)

13.2 Lemmas for Ordinals

lemma *OrdI*:
  [| *Transset*(i); !!x. x:i ==> *Transset*(x) |] ==> *Ord*(i)
by (*simp add*: *Ord-def*)

lemma *Ord-is-Transset*: *Ord*(i) ==> *Transset*(i)
by (*simp add*: *Ord-def*)

lemma *Ord-contains-Transset*:
  [| *Ord*(i); j:i |] ==> *Transset*(j)
by (*unfold Ord-def*, *blast*)

lemma *Ord-in-Ord*: [| *Ord*(i); j:i |] ==> *Ord*(j)
by (*unfold Ord-def Transset-def*, *blast*)

lemma *Ord-in-Ord'*: [| j:i; *Ord*(i) |] ==> *Ord*(j)
by (*blast intro*: *Ord-in-Ord*)

lemmas *Ord-succD* = *Ord-in-Ord* [*OF* - *succI1*]

lemma *Ord-subset-Ord*: [| *Ord*(i); *Transset*(j); j<=i |] ==> *Ord*(j)
by (*simp add*: *Ord-def Transset-def*, *blast*)

lemma *OrdmemD*: [| j:i; *Ord*(i) |] ==> j<=i
by (*unfold Ord-def Transset-def*, *blast*)

lemma *Ord-trans*: [| i:j; j:k; *Ord*(k) |] ==> i:k
by (*blast dest*: *OrdmemD*)

lemma *Ord-succ-subsetI*: [| i:j; *Ord*(j) |] ==> *succ*(i) <= j
by (*blast dest*: *OrdmemD*)

13.3 The Construction of Ordinals: 0, succ, Union

lemma *Ord-0* [*iff*,*TC*]: *Ord*(0)
by (*blast intro*: *OrdI Transset-0*)

lemma *Ord-succ* [*TC*]: *Ord*(i) ==> *Ord*(*succ*(i))

by (*blast intro*: *OrdI Transset-succ Ord-is-Transset Ord-contains-Transset*)

lemmas *Ord-1* = *Ord-0* [*THEN Ord-succ*]

lemma *Ord-succ-iff* [*iff*]: *Ord*(*succ*(*i*)) <-> *Ord*(*i*)
by (*blast intro*: *Ord-succ dest*!: *Ord-succD*)

lemma *Ord-Un* [*intro*,*simp*,*TC*]: [| *Ord*(*i*); *Ord*(*j*) |] ==> *Ord*(*i Un j*)
apply (*unfold Ord-def*)
apply (*blast intro*!: *Transset-Un*)
done

lemma *Ord-Int* [*TC*]: [| *Ord*(*i*); *Ord*(*j*) |] ==> *Ord*(*i Int j*)
apply (*unfold Ord-def*)
apply (*blast intro*!: *Transset-Int*)
done

lemma *ON-class*: ~ (*ALL i*. *i*:*X* <-> *Ord*(*i*))
apply (*rule notI*)
apply (*frule-tac x* = *X* **in** *spec*)
apply (*safe elim*!: *mem-irrefl*)
apply (*erule swap*, *rule OrdI* [*OF - Ord-is-Transset*])
apply (*simp add*: *Transset-def*)
apply (*blast intro*: *Ord-in-Ord*)+
done

13.4 ¡ is 'less Than' for Ordinals

lemma *ltI*: [| *i*:*j*; *Ord*(*j*) |] ==> *i*<*j*
by (*unfold lt-def*, *blast*)

lemma *ltE*:
[| *i*<*j*; [| *i*:*j*; *Ord*(*i*); *Ord*(*j*) |] ==> *P* |] ==> *P*
apply (*unfold lt-def*)
apply (*blast intro*: *Ord-in-Ord*)
done

lemma *ltD*: *i*<*j* ==> *i*:*j*
by (*erule ltE*, *assumption*)

lemma *not-lt0* [*simp*]: ~ *i*<*0*
by (*unfold lt-def*, *blast*)

lemma *lt-Ord*: *j*<*i* ==> *Ord*(*j*)
by (*erule ltE*, *assumption*)

lemma *lt-Ord2*: *j*<*i* ==> *Ord*(*i*)
by (*erule ltE*, *assumption*)

lemmas *le-Ord2* = *lt-Ord2* [*THEN Ord-succD*]

lemmas *lt0E* = *not-lt0* [*THEN notE*, *elim*!]

lemma *lt-trans*: [| *i*<*j*; *j*<*k* |] ==> *i*<*k*
by (*blast intro*!: *ltI elim*!: *ltE intro*: *Ord-trans*)

lemma *lt-not-sym*: *i*<*j* ==> ~ (*j*<*i*)
apply (*unfold lt-def*)
apply (*blast elim*: *mem-asym*)
done

lemmas *lt-asym* = *lt-not-sym* [*THEN swap*]

lemma *lt-irrefl* [*elim*!]: *i*<*i* ==> *P*
by (*blast intro*: *lt-asym*)

lemma *lt-not-refl*: ~ *i*<*i*
apply (*rule notI*)
apply (*erule lt-irrefl*)
done

lemma *le-iff*: *i le j* <-> *i*<*j* | (*i*=*j* & *Ord*(*j*))
by (*unfold lt-def*, *blast*)

lemma *leI*: *i*<*j* ==> *i le j*
by (*simp* (*no-asm-simp*) *add*: *le-iff*)

lemma *le-eqI*: [| *i*=*j*; *Ord*(*j*) |] ==> *i le j*
by (*simp* (*no-asm-simp*) *add*: *le-iff*)

lemmas *le-refl* = *refl* [*THEN le-eqI*]

lemma *le-refl-iff* [*iff*]: *i le i* <-> *Ord*(*i*)
by (*simp* (*no-asm-simp*) *add*: *lt-not-refl le-iff*)

lemma *leCI*: (~ (*i*=*j* & *Ord*(*j*)) ==> *i*<*j*) ==> *i le j*
by (*simp add*: *le-iff*, *blast*)

lemma *leE*:
  [| *i le j*; *i*<*j* ==> *P*; [| *i*=*j*; *Ord*(*j*) |] ==> *P* |] ==> *P*

by (*simp add*: *le-iff*, *blast*)

lemma *le-anti-sym*: [| *i le j*; *j le i* |] ==> *i=j*
apply (*simp add*: *le-iff*)
apply (*blast elim*: *lt-asym*)
done

lemma *le0-iff* [*simp*]: *i le 0* <-> *i=0*
by (*blast elim*!: *leE*)

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

lemma *MemrelI* [*intro*!]: [| *a*: *b*; *a*: *A*; *b*: *A* |] ==> <*a*,*b*> : *Memrel*(*A*)
by *auto*

lemma *MemrelE* [*elim*!]:
  [| <*a*,*b*> : *Memrel*(*A*);
    [| *a*: *A*; *b*: *A*; *a*:*b* |] ==> *P* |]
  ==> *P*
by *auto*

lemma *Memrel-type*: *Memrel*(*A*) <= *A***A*
by (*unfold Memrel-def*, *blast*)

lemma *Memrel-mono*: *A*<=*B* ==> *Memrel*(*A*) <= *Memrel*(*B*)
by (*unfold Memrel-def*, *blast*)

lemma *Memrel-0* [*simp*]: *Memrel*(*0*) = *0*
by (*unfold Memrel-def*, *blast*)

lemma *Memrel-1* [*simp*]: *Memrel*(*1*) = *0*
by (*unfold Memrel-def*, *blast*)

lemma *relation-Memrel*: *relation*(*Memrel*(*A*))
by (*simp add*: *relation-def Memrel-def*)

lemma *wf-Memrel*: *wf*(*Memrel*(*A*))
apply (*unfold wf-def*)
apply (*rule foundation* [*THEN disjE*, *THEN allI*], *erule disjI1*, *blast*)
done

The premise *Ord*(*i*) does not suffice.

lemma *trans-Memrel*:

Ord(*i*) ==> *trans*(*Memrel*(*i*))
by (*unfold Ord-def Transset-def trans-def*, *blast*)

However, the following premise is strong enough.

lemma *Transset-trans-Memrel*:
∀*j*∈*i*. *Transset*(*j*) ==> *trans*(*Memrel*(*i*))
by (*unfold Transset-def trans-def*, *blast*)

lemma *Transset-Memrel-iff*:
Transset(*A*) ==> <*a*,*b*> : *Memrel*(*A*) <−> *a*:*b* & *b*:*A*
by (*unfold Transset-def*, *blast*)

13.6 Transfinite Induction

lemma *Transset-induct*:
[| *i*: *k*; *Transset*(*k*);
!!*x*.[| *x*: *k*; *ALL y*:*x*. *P*(*y*) |] ==> *P*(*x*) |]
==> *P*(*i*)
apply (*simp add*: *Transset-def*)
apply (*erule wf-Memrel* [*THEN wf-induct2*], *blast*+)
done

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*)
apply (*rule Ord-succ* [*THEN succI1* [*THEN Ord-induct*]], *assumption*)
apply (*blast intro*: *Ord-succ* [*THEN Ord-in-Ord*])
done

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*)
apply (*erule trans-induct*)
apply (*rule impI* [*THEN allI*])
apply (*erule-tac i*=*j* **in** *trans-induct*)
apply (*blast dest*: *Ord-trans*)
done

lemma *Ord-linear-lt*:
[| *Ord*(*i*); *Ord*(*j*); *i*<*j* ==> *P*; *i*=*j* ==> *P*; *j*<*i* ==> *P* |] ==> *P*
apply (*simp add*: *lt-def*)
apply (*rule-tac i1*=*i* **and** *j1*=*j* **in** *Ord-linear* [*THEN disjE*], *blast*+)
done

lemma *Ord-linear2*:
[| *Ord*(*i*); *Ord*(*j*); *i*<*j* ==> *P*; *j le i* ==> *P* |] ==> *P*
apply (*rule-tac i* = *i* **and** *j* = *j* **in** *Ord-linear-lt*)
apply (*blast intro*: *leI le-eqI sym*) +
done

lemma *Ord-linear-le*:
[| *Ord*(*i*); *Ord*(*j*); *i le j* ==> *P*; *j le i* ==> *P* |] ==> *P*
apply (*rule-tac i* = *i* **and** *j* = *j* **in** *Ord-linear-lt*)
apply (*blast intro*: *leI le-eqI*) +
done

lemma *le-imp-not-lt*: *j le i* ==> ~ *i*<*j*
by (*blast elim*!: *leE elim*: *lt-asym*)

lemma *not-lt-imp-le*: [| ~ *i*<*j*; *Ord*(*i*); *Ord*(*j*) |] ==> *j le i*
by (*rule-tac i* = *i* **and** *j* = *j* **in** *Ord-linear2*, *auto*)

13.6.2 Some Rewrite Rules for ¡, le

lemma *Ord-mem-iff-lt*: *Ord*(*j*) ==> *i*:*j* <-> *i*<*j*
by (*unfold lt-def*, *blast*)

lemma *not-lt-iff-le*: [| *Ord*(*i*); *Ord*(*j*) |] ==> ~ *i*<*j* <-> *j le i*
by (*blast dest*: *le-imp-not-lt not-lt-imp-le*)

lemma *not-le-iff-lt*: [| *Ord*(*i*); *Ord*(*j*) |] ==> ~ *i le j* <-> *j*<*i*
by (*simp* (*no-asm-simp*) *add*: *not-lt-iff-le* [*THEN iff-sym*])

lemma *Ord-0-le*: *Ord*(*i*) ==> *0 le i*
by (*erule not-lt-iff-le* [*THEN iffD1*], *auto*)

lemma *Ord-0-lt*: [| *Ord*(*i*); *i*~=*0* |] ==> *0*<*i*
apply (*erule not-le-iff-lt* [*THEN iffD1*])
apply (*rule Ord-0*, *blast*)
done

lemma *Ord-0-lt-iff*: *Ord*(*i*) ==> *i*~=*0* <-> *0*<*i*
by (*blast intro*: *Ord-0-lt*)

13.7 Results about Less-Than or Equals

lemma *zero-le-succ-iff* [*iff*]: 0 le succ(x) <-> Ord(x)
by (*blast intro*: *Ord-0-le elim*: *ltE*)

lemma *subset-imp-le*: [| j<=i; Ord(i); Ord(j) |] ==> j le i
apply (*rule not-lt-iff-le* [*THEN iffD1*], *assumption*+)
apply (*blast elim*: *ltE mem-irrefl*)
done

lemma *le-imp-subset*: i le j ==> i<=j
by (*blast dest*: *OrdmemD elim*: *ltE leE*)

lemma *le-subset-iff*: j le i <-> j<=i & Ord(i) & Ord(j)
by (*blast dest*: *subset-imp-le le-imp-subset elim*: *ltE*)

lemma *le-succ-iff*: i le succ(j) <-> i le j | i=succ(j) & Ord(i)
apply (*simp* (*no-asm*) *add*: *le-iff*)
apply *blast*
done

lemma *all-lt-imp-le*: [| Ord(i); Ord(j); !!x. x<j ==> x<i |] ==> j le i
by (*blast intro*: *not-lt-imp-le dest*: *lt-irrefl*)

13.7.1 Transitivity Laws

lemma *lt-trans1*: [| i le j; j<k |] ==> i<k
by (*blast elim*!: *leE intro*: *lt-trans*)

lemma *lt-trans2*: [| i<j; j le k |] ==> i<k
by (*blast elim*!: *leE intro*: *lt-trans*)

lemma *le-trans*: [| i le j; j le k |] ==> i le k
by (*blast intro*: *lt-trans1*)

lemma *succ-leI*: i<j ==> succ(i) le j
apply (*rule not-lt-iff-le* [*THEN iffD1*])
apply (*blast elim*: *ltE leE lt-asym*)+
done

lemma *succ-leE*: succ(i) le j ==> i<j
apply (*rule not-le-iff-lt* [*THEN iffD1*])
apply (*blast elim*: *ltE leE lt-asym*)+
done

lemma *succ-le-iff* [*iff*]: succ(i) le j <-> i<j
by (*blast intro*: *succ-leI succ-leE*)

lemma *succ-le-imp-le*: *succ*(*i*) *le succ*(*j*) ==> *i le j*
by (*blast dest*!: *succ-leE*)

lemma *lt-subset-trans*: [| *i* <= *j*; *j*<*k*; *Ord*(*i*) |] ==> *i*<*k*
apply (*rule subset-imp-le* [*THEN lt-trans1*])
apply (*blast intro*: *elim*: *ltE*) +
done

lemma *lt-imp-0-lt*: *j*<*i* ==> *0*<*i*
by (*blast intro*: *lt-trans1 Ord-0-le* [*OF lt-Ord*])

lemma *succ-lt-iff*: *succ*(*i*) < *j* <-> *i*<*j* & *succ*(*i*) ≠ *j*
apply *auto*
apply (*blast intro*: *lt-trans le-refl dest*: *lt-Ord*)
apply (*frule lt-Ord*)
apply (*rule not-le-iff-lt* [*THEN iffD1*])
  apply (*blast intro*: *lt-Ord2*)
 apply *blast*
apply (*simp add*: *lt-Ord lt-Ord2 le-iff*)
apply (*blast dest*: *lt-asym*)
done

lemma *Ord-succ-mem-iff*: *Ord*(*j*) ==> *succ*(*i*) ∈ *succ*(*j*) <-> *i*∈*j*
apply (*insert succ-le-iff* [*of i j*])
apply (*simp add*: *lt-def*)
done

13.7.2 Union and Intersection

lemma *Un-upper1-le*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *i le i Un j*
by (*rule Un-upper1* [*THEN subset-imp-le*], *auto*)

lemma *Un-upper2-le*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *j le i Un j*
by (*rule Un-upper2* [*THEN subset-imp-le*], *auto*)

lemma *Un-least-lt*: [| *i*<*k*; *j*<*k* |] ==> *i Un j* < *k*
apply (*rule-tac i* = *i* **and** *j* = *j* **in** *Ord-linear-le*)
apply (*auto simp add*: *Un-commute le-subset-iff subset-Un-iff lt-Ord*)
done

lemma *Un-least-lt-iff*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *i Un j* < *k* <-> *i*<*k* & *j*<*k*
apply (*safe intro*!: *Un-least-lt*)
apply (*rule-tac* [*2*] *Un-upper2-le* [*THEN lt-trans1*])
apply (*rule Un-upper1-le* [*THEN lt-trans1*], *auto*)
done

lemma *Un-least-mem-iff*:
  [| *Ord*(*i*); *Ord*(*j*); *Ord*(*k*) |] ==> *i Un j* : *k* <-> *i*:*k* & *j*:*k*

apply (*insert Un-least-lt-iff* [*of i j k*])
apply (*simp add*: *lt-def*)
done

lemma *Int-greatest-lt*: [| i<k; j<k |] ==> i *Int* j < k
apply (*rule-tac i* = *i* **and** *j* = *j* **in** *Ord-linear-le*)
apply (*auto simp add*: *Int-commute le-subset-iff subset-Int-iff lt-Ord*)
done

lemma *Ord-Un-if*:
  [| *Ord*(i); *Ord*(j) |] ==> i ∪ j = (*if* j<i *then* i *else* j)
by (*simp add*: *not-lt-iff-le le-imp-subset leI*
    subset-Un-iff [*symmetric*] *subset-Un-iff2* [*symmetric*])

lemma *succ-Un-distrib*:
  [| *Ord*(i); *Ord*(j) |] ==> *succ*(i ∪ j) = *succ*(i) ∪ *succ*(j)
by (*simp add*: *Ord-Un-if lt-Ord le-Ord2*)

lemma *lt-Un-iff*:
  [| *Ord*(i); *Ord*(j) |] ==> k < i ∪ j <-> k < i | k < j
apply (*simp add*: *Ord-Un-if not-lt-iff-le*)
apply (*blast intro*: *leI lt-trans2*)+
done

lemma *le-Un-iff*:
  [| *Ord*(i); *Ord*(j) |] ==> k ≤ i ∪ j <-> k ≤ i | k ≤ j
by (*simp add*: *succ-Un-distrib lt-Un-iff* [*symmetric*])

lemma *Un-upper1-lt*: [|k < i; *Ord*(j)|] ==> k < i *Un* j
by (*simp add*: *lt-Un-iff lt-Ord2*)

lemma *Un-upper2-lt*: [|k < j; *Ord*(i)|] ==> k < i *Un* j
by (*simp add*: *lt-Un-iff lt-Ord2*)

lemma *Ord-Union-succ-eq*: *Ord*(i) ==> ⋃(*succ*(i)) = i
by (*blast intro*: *Ord-trans*)

13.8 Results about Limits

lemma *Ord-Union* [*intro*,*simp*,*TC*]: [| !!i. i:A ==> *Ord*(i) |] ==> *Ord*(*Union*(A))
apply (*rule Ord-is-Transset* [*THEN Transset-Union-family*, *THEN OrdI*])
apply (*blast intro*: *Ord-contains-Transset*)+
done

lemma *Ord-UN* [*intro*,*simp*,*TC*]:
  [| !!x. x:A ==> *Ord*(B(x)) |] ==> *Ord*(⋃x∈A. B(x))
by (*rule Ord-Union*, *blast*)

lemma *Ord-Inter* [*intro*,*simp*,*TC*]:
  [| !!*i*. *i*:*A* ==> *Ord*(*i*) |] ==> *Ord*(*Inter*(*A*))
apply (*rule Transset-Inter-family* [*THEN OrdI*])
apply (*blast intro*: *Ord-is-Transset*)
apply (*simp add*: *Inter-def*)
apply (*blast intro*: *Ord-contains-Transset*)
done

lemma *Ord-INT* [*intro*,*simp*,*TC*]:
  [| !!*x*. *x*:*A* ==> *Ord*(*B*(*x*)) |] ==> *Ord*($\bigcap$ *x*∈*A*. *B*(*x*))
by (*rule Ord-Inter*, *blast*)

lemma *UN-least-le*:
  [| *Ord*(*i*); !!*x*. *x*:*A* ==> *b*(*x*) *le i* |] ==> ($\bigcup$ *x*∈*A*. *b*(*x*)) *le i*
apply (*rule le-imp-subset* [*THEN UN-least*, *THEN subset-imp-le*])
apply (*blast intro*: *Ord-UN elim*: *ltE*)+
done

lemma *UN-succ-least-lt*:
  [| *j*<*i*; !!*x*. *x*:*A* ==> *b*(*x*)<*j* |] ==> ($\bigcup$ *x*∈*A*. *succ*(*b*(*x*))) < *i*
apply (*rule ltE*, *assumption*)
apply (*rule UN-least-le* [*THEN lt-trans2*])
apply (*blast intro*: *succ-leI*)+
done

lemma *UN-upper-lt*:
  [| *a*∈*A*; *i* < *b*(*a*); *Ord*($\bigcup$ *x*∈*A*. *b*(*x*)) |] ==> *i* < ($\bigcup$ *x*∈*A*. *b*(*x*))
by (*unfold lt-def*, *blast*)

lemma *UN-upper-le*:
  [| *a*: *A*; *i le b*(*a*); *Ord*($\bigcup$ *x*∈*A*. *b*(*x*)) |] ==> *i le* ($\bigcup$ *x*∈*A*. *b*(*x*))
apply (*frule ltD*)
apply (*rule le-imp-subset* [*THEN subset-trans*, *THEN subset-imp-le*])
apply (*blast intro*: *lt-Ord UN-upper*)+
done

lemma *lt-Union-iff*: ∀ *i*∈*A*. *Ord*(*i*) ==> (*j* < $\bigcup$(*A*)) <-> (∃ *i*∈*A*. *j*<*i*)
by (*auto simp*: *lt-def Ord-Union*)

lemma *Union-upper-le*:
  [| *j*: *J*; *i*≤*j*; *Ord*($\bigcup$(*J*)) |] ==> *i* ≤ $\bigcup$ *J*
apply (*subst Union-eq-UN*)
apply (*rule UN-upper-le*, *auto*)
done

lemma *le-implies-UN-le-UN*:

[| !!x. x:A ==> c(x) le d(x) |] ==> (⋃x∈A. c(x)) le (⋃x∈A. d(x))
apply (*rule UN-least-le*)
apply (*rule-tac* [*2*] *UN-upper-le*)
apply (*blast intro*: *Ord-UN le-Ord2*)+
done

lemma *Ord-equality*: *Ord*(*i*) ==> (⋃y∈i. *succ*(*y*)) = *i*
by (*blast intro*: *Ord-trans*)

lemma *Ord-Union-subset*: *Ord*(*i*) ==> *Union*(*i*) <= *i*
by (*blast intro*: *Ord-trans*)

13.9 Limit Ordinals – General Properties

lemma *Limit-Union-eq*: *Limit*(*i*) ==> *Union*(*i*) = *i*
apply (*unfold Limit-def*)
apply (*fast intro*!: *ltI elim*!: *ltE elim*: *Ord-trans*)
done

lemma *Limit-is-Ord*: *Limit*(*i*) ==> *Ord*(*i*)
apply (*unfold Limit-def*)
apply (*erule conjunct1*)
done

lemma *Limit-has-0*: *Limit*(*i*) ==> *0* < *i*
apply (*unfold Limit-def*)
apply (*erule conjunct2* [*THEN conjunct1*])
done

lemma *Limit-nonzero*: *Limit*(*i*) ==> *i* ~= *0*
by (*drule Limit-has-0*, *blast*)

lemma *Limit-has-succ*: [| *Limit*(*i*); *j*<*i* |] ==> *succ*(*j*) < *i*
by (*unfold Limit-def*, *blast*)

lemma *Limit-succ-lt-iff* [*simp*]: *Limit*(*i*) ==> *succ*(*j*) < *i* <-> (*j*<*i*)
apply (*safe intro*!: *Limit-has-succ*)
apply (*frule lt-Ord*)
apply (*blast intro*: *lt-trans*)
done

lemma *zero-not-Limit* [*iff*]: ~ *Limit*(*0*)
by (*simp add*: *Limit-def*)

lemma *Limit-has-1*: *Limit*(*i*) ==> *1* < *i*
by (*blast intro*: *Limit-has-0 Limit-has-succ*)

lemma *increasing-LimitI*: [| *0*<*l*; ∀x∈l. ∃y∈l. *x*<*y* |] ==> *Limit*(*l*)

apply (*unfold Limit-def*, *simp add*: *lt-Ord2*, *clarify*)
apply (*drule-tac i=y* **in** *ltD*)
apply (*blast intro*: *lt-trans1* [*OF - ltI*] *lt-Ord2*)
done

lemma *non-succ-LimitI*:
[| *0<i*; *ALL y*. *succ*(*y*) ~= *i* |] ==> *Limit*(*i*)
apply (*unfold Limit-def*)
apply (*safe del*: *subsetI*)
apply (*rule-tac* [*2*] *not-le-iff-lt* [*THEN iffD1*])
apply (*simp-all add*: *lt-Ord lt-Ord2*)
apply (*blast elim*: *leE lt-asym*)
done

lemma *succ-LimitE* [*elim*!]: *Limit*(*succ*(*i*)) ==> *P*
apply (*rule lt-irrefl*)
apply (*rule Limit-has-succ*, *assumption*)
apply (*erule Limit-is-Ord* [*THEN Ord-succD*, *THEN le-refl*])
done

lemma *not-succ-Limit* [*simp*]: ~ *Limit*(*succ*(*i*))
by *blast*

lemma *Limit-le-succD*: [| *Limit*(*i*); *i le succ*(*j*) |] ==> *i le j*
by (*blast elim*!: *leE*)

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*)
by (*blast intro*!: *non-succ-LimitI Ord-0-lt*)

lemma *Ord-cases*:
[| *Ord*(*i*);
i=0 ==> *P*;
!!*j*. [| *Ord*(*j*); *i=succ*(*j*) |] ==> *P*;
Limit(*i*) ==> *P*
|] ==> *P*
by (*drule Ord-cases-disj*, *blast*)

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*)
apply (*erule trans-induct*)
apply (*erule Ord-cases*, *blast*+)
done

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*:
  $\forall i{\in}I.\ Ord(i) ==> I{=}0 \lor \bigcup(I) \in I \lor (\bigcup(I) \notin I \land Limit(\bigcup(I)))$
apply (*clarify elim*!: *not-emptyE*)
apply (*cases* $\bigcup(I)$ *rule*: *Ord-cases*)
   apply (*blast intro*: *Ord-Union*)
  apply (*blast intro*: *subst-elem*)
 apply *auto*
apply (*clarify elim*!: *equalityE succ-subsetE*)
apply (*simp add*: *Union-subset-iff*)
apply (*subgoal-tac B* = *succ*(*j*), *blast*)
apply (*rule le-anti-sym*)
 apply (*simp add*: *le-subset-iff*)
apply (*simp add*: *ltI*)
done

If the union of a set of ordinals is a successor, then it is an element of that set.

lemma *Ord-Union-eq-succD*: $[|\forall x{\in}X.\ Ord(x);\ \bigcup X = succ(j)|] ==> succ(j) \in X$
by (*drule Ord-set-cases*, *auto*)

lemma *Limit-Union* [*rule-format*]: $[|\ I \neq 0;\ \forall i{\in}I.\ Limit(i)\ |] ==> Limit(\bigcup I)$
apply (*simp add*: *Limit-def lt-def*)
apply (*blast intro*!: *equalityI*)
done

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

lemma [*simp*]: ~(*EX x*<*0*. *P*(*x*))
by (*simp add*: *oex-def*)

lemma [*simp*]: (*ALL x*<*succ*(*i*). *P*(*x*)) <−> (*Ord*(*i*) −−> *P*(*i*) & (*ALL x*<*i*. *P*(*x*)))
apply (*simp add*: *oall-def le-iff*)
apply (*blast intro*: *lt-Ord2*)
done

lemma [*simp*]: (*EX x*<*succ*(*i*). *P*(*x*)) <−> (*Ord*(*i*) & (*P*(*i*) | (*EX x*<*i*. *P*(*x*))))
apply (*simp add*: *oex-def le-iff*)
apply (*blast intro*: *lt-Ord2*)
done

14.1.2 Union over ordinals

lemma *Ord-OUN* [*intro*,*simp*]:
[| !!*x*. *x*<*A* ==> *Ord*(*B*(*x*)) |] ==> *Ord*(⋃*x*<*A*. *B*(*x*))
by (*simp add*: *OUnion-def ltI Ord-UN*)

lemma *OUN-upper-lt*:
$[|\ a{<}A;\ \ i < b(a);\ \ Ord(\bigcup x{<}A.\ b(x))\ |] ==> i < (\bigcup x{<}A.\ b(x))$
by (*unfold OUnion-def lt-def*, *blast*)

lemma *OUN-upper-le*:
$[|\ a{<}A;\ \ i{\le}b(a);\ \ Ord(\bigcup x{<}A.\ b(x))\ |] ==> i \le (\bigcup x{<}A.\ b(x))$
apply (*unfold OUnion-def*, *auto*)
apply (*rule UN-upper-le*)
apply (*auto simp add*: *lt-def*)
done

lemma *Limit-OUN-eq*: $Limit(i) ==> (\bigcup x{<}i.\ x) = i$
by (*simp add*: *OUnion-def Limit-Union-eq Limit-is-Ord*)

lemma *OUN-least*:
$(!!x.\ x{<}A ==> B(x) \subseteq C) ==> (\bigcup x{<}A.\ B(x)) \subseteq C$
by (*simp add*: *OUnion-def UN-least ltI*)

lemma *OUN-least-le*:
$[|\ Ord(i);\ \ !!x.\ x{<}A ==> b(x) \le i\ |] ==> (\bigcup x{<}A.\ b(x)) \le i$
by (*simp add*: *OUnion-def UN-least-le ltI Ord-0-le*)

lemma *le-implies-OUN-le-OUN*:
$[|\ !!x.\ x{<}A ==> c(x) \le d(x)\ |] ==> (\bigcup x{<}A.\ c(x)) \le (\bigcup x{<}A.\ d(x))$
by (*blast intro*: *OUN-least-le OUN-upper-le le-Ord2 Ord-OUN*)

lemma *OUN-UN-eq*:
$(!!x.\ x{:}A ==> Ord(B(x)))$
$==> (\bigcup z < (\bigcup x{\in}A.\ B(x)).\ C(z)) = (\bigcup x{\in}A.\ \bigcup z < B(x).\ C(z))$
by (*simp add*: *OUnion-def*)

lemma *OUN-Union-eq*:
$(!!x.\ x{:}X ==> Ord(x))$
$==> (\bigcup z < Union(X).\ C(z)) = (\bigcup x{\in}X.\ \bigcup z < x.\ C(z))$
by (*simp add*: *OUnion-def*)

lemma *atomize-oall* [*symmetric*, *rulify*]:
$(!!x.\ x{<}A ==> P(x)) == Trueprop\ (ALL\ x{<}A.\ P(x))$
by (*simp add*: *oall-def atomize-all atomize-imp*)

14.1.3 universal quantifier for ordinals

lemma *oallI* [*intro*!]:
$[|\ !!x.\ x{<}A ==> P(x)\ |] ==> ALL\ x{<}A.\ P(x)$
by (*simp add*: *oall-def*)

```
lemma ospec: [| ALL x<A. P(x);  x<A |] ==> P(x)
by (simp add: oall-def)

lemma oallE:
    [| ALL x<A. P(x);  P(x) ==> Q;  ~x<A ==> Q |] ==> Q
by (simp add: oall-def, blast)

lemma rev-oallE [elim]:
    [| ALL x<A. P(x);  ~x<A ==> Q;  P(x) ==> Q |] ==> Q
by (simp add: oall-def, blast)


lemma oall-simp [simp]: (ALL x<a. True) <-> True
by blast


lemma oall-cong [cong]:
    [| a=a';  !!x. x<a' ==> P(x) <-> P'(x) |]
     ==> oall(a, %x. P(x)) <-> oall(a', %x. P'(x))
by (simp add: oall-def)
```

14.1.4 existential quantifier for ordinals

```
lemma oexI [intro]:
    [| P(x);  x<A |] ==> EX x<A. P(x)
apply (simp add: oex-def, blast)
done


lemma oexCI:
   [| ALL x<A. ~P(x) ==> P(a);  a<A |] ==> EX x<A. P(x)
apply (simp add: oex-def, blast)
done

lemma oexE [elim!]:
    [| EX x<A. P(x);  !!x. [| x<A; P(x) |] ==> Q |] ==> Q
apply (simp add: oex-def, blast)
done

lemma oex-cong [cong]:
    [| a=a';  !!x. x<a' ==> P(x) <-> P'(x) |]
     ==> oex(a, %x. P(x)) <-> oex(a', %x. P'(x))
apply (simp add: oex-def cong add: conj-cong)
done
```

14.1.5 Rules for Ordinal-Indexed Unions

```
lemma OUN-I [intro]: [| a<i;  b: B(a) |] ==> b: (⋃z<i. B(z))
by (unfold OUnion-def lt-def, blast)
```

```
lemma OUN-E [elim!]:
  [| b : (⋃z<i. B(z)); !!a.[| b: B(a); a<i |] ==> R |] ==> R
apply (unfold OUnion-def lt-def, blast)
done

lemma OUN-iff: b : (⋃x<i. B(x)) <-> (EX x<i. b : B(x))
by (unfold OUnion-def oex-def lt-def, blast)

lemma OUN-cong [cong]:
  [| i=j; !!x. x<j ==> C(x)=D(x) |] ==> (⋃x<i. C(x)) = (⋃x<j. D(x))
by (simp add: OUnion-def lt-def OUN-iff)

lemma lt-induct:
  [| i<k; !!x.[| x<k; ALL y<x. P(y) |] ==> P(x) |]  ==>  P(i)
apply (simp add: lt-def oall-def)
apply (erule conjE)
apply (erule Ord-induct, assumption, blast)
done
```

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)
by (simp add: rall-def)
```

lemma *rspec*: [| *ALL x*[*M*]. *P*(*x*); *M*(*x*) |] ==> *P*(*x*)
by (*simp add*: *rall-def*)

lemma *rev-rallE* [*elim*]:
[| *ALL x*[*M*]. *P*(*x*); ~ *M*(*x*) ==> *Q*; *P*(*x*) ==> *Q* |] ==> *Q*
by (*simp add*: *rall-def*, *blast*)

lemma *rallE*: [| *ALL x*[*M*]. *P*(*x*); *P*(*x*) ==> *Q*; ~ *M*(*x*) ==> *Q* |] ==> *Q*
by *blast*

lemma *rall-triv* [*simp*]: (*ALL x*[*M*]. *P*) <-> ((*EX x*. *M*(*x*)) --> *P*)
by (*simp add*: *rall-def*)

lemma *rall-cong* [*cong*]:
(!!*x*. *M*(*x*) ==> *P*(*x*) <-> *P'*(*x*)) ==> (*ALL x*[*M*]. *P*(*x*)) <-> (*ALL x*[*M*]. *P'*(*x*))
by (*simp add*: *rall-def*)

14.2.2 Relativized existential quantifier

lemma *rexI* [*intro*]: [| *P*(*x*); *M*(*x*) |] ==> *EX x*[*M*]. *P*(*x*)
by (*simp add*: *rex-def*, *blast*)

lemma *rev-rexI*: [| *M*(*x*); *P*(*x*) |] ==> *EX x*[*M*]. *P*(*x*)
by *blast*

lemma *rexCI*: [| *ALL x*[*M*]. ~*P*(*x*) ==> *P*(*a*); *M*(*a*) |] ==> *EX x*[*M*]. *P*(*x*)
by *blast*

lemma *rexE* [*elim*!]: [| *EX x*[*M*]. *P*(*x*); !!*x*. [| *M*(*x*); *P*(*x*) |] ==> *Q* |] ==> *Q*
by (*simp add*: *rex-def*, *blast*)

lemma *rex-triv* [*simp*]: (*EX x*[*M*]. *P*) <-> ((*EX x*. *M*(*x*)) & *P*)
by (*simp add*: *rex-def*)

lemma *rex-cong* [*cong*]:
(!!*x*. *M*(*x*) ==> *P*(*x*) <-> *P'*(*x*)) ==> (*EX x*[*M*]. *P*(*x*)) <-> (*EX x*[*M*]. *P'*(*x*))
by (*simp add*: *rex-def cong*: *conj-cong*)

lemma *rall-is-ball* [*simp*]: (∀ *x*[%*z*. *z*∈*A*]. *P*(*x*)) <-> (∀ *x*∈*A*. *P*(*x*))
by *blast*

lemma *rex-is-bex* [*simp*]: (∃ *x*[%*z*. *z*∈*A*]. *P*(*x*)) <−> (∃ *x*∈*A*. *P*(*x*))
by *blast*

lemma *atomize-rall*: (!!*x*. *M*(*x*) ==> *P*(*x*)) == *Trueprop* (*ALL x*[*M*]. *P*(*x*))
by (*simp add*: *rall-def atomize-all atomize-imp*)

declare *atomize-rall* [*symmetric*, *rulify*]

lemma *rall-simps1*:
  (*ALL x*[*M*]. *P*(*x*) & *Q*) <−> (*ALL x*[*M*]. *P*(*x*)) & ((*ALL x*[*M*]. *False*) | *Q*)
  (*ALL x*[*M*]. *P*(*x*) | *Q*) <−> ((*ALL x*[*M*]. *P*(*x*)) | *Q*)
  (*ALL x*[*M*]. *P*(*x*) −−> *Q*) <−> ((*EX x*[*M*]. *P*(*x*)) −−> *Q*)
  (~(*ALL x*[*M*]. *P*(*x*))) <−> (*EX x*[*M*]. ~*P*(*x*))
by *blast*+

lemma *rall-simps2*:
  (*ALL x*[*M*]. *P* & *Q*(*x*)) <−> ((*ALL x*[*M*]. *False*) | *P*) & (*ALL x*[*M*]. *Q*(*x*))
  (*ALL x*[*M*]. *P* | *Q*(*x*)) <−> (*P* | (*ALL x*[*M*]. *Q*(*x*)))
  (*ALL x*[*M*]. *P* −−> *Q*(*x*)) <−> (*P* −−> (*ALL x*[*M*]. *Q*(*x*)))
by *blast*+

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*)))
by *blast*

lemma *rex-simps1*:
  (*EX x*[*M*]. *P*(*x*) & *Q*) <−> ((*EX x*[*M*]. *P*(*x*)) & *Q*)
  (*EX x*[*M*]. *P*(*x*) | *Q*) <−> (*EX x*[*M*]. *P*(*x*)) | ((*EX x*[*M*]. *True*) & *Q*)
  (*EX x*[*M*]. *P*(*x*) −−> *Q*) <−> ((*ALL x*[*M*]. *P*(*x*)) −−> ((*EX x*[*M*]. *True*)
& *Q*))
  (~(*EX x*[*M*]. *P*(*x*))) <−> (*ALL x*[*M*]. ~*P*(*x*))
by *blast*+

lemma *rex-simps2*:
  (*EX x*[*M*]. *P* & *Q*(*x*)) <−> (*P* & (*EX x*[*M*]. *Q*(*x*)))
  (*EX x*[*M*]. *P* | *Q*(*x*)) <−> ((*EX x*[*M*]. *True*) & *P*) | (*EX x*[*M*]. *Q*(*x*))
  (*EX x*[*M*]. *P* −−> *Q*(*x*)) <−> (((*ALL x*[*M*]. *False*) | *P*) −−> (*EX x*[*M*].
Q(*x*)))
by *blast*+

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

lemma *rex-disj-distrib*:
  (*EX x*[*M*]. *P*(*x*) | *Q*(*x*)) <−> ((*EX x*[*M*]. *P*(*x*)) | (*EX x*[*M*]. *Q*(*x*)))
by *blast*

14.2.3 One-point rule for bounded quantifiers

lemma *rex-triv-one-point1* [*simp*]: (*EX x*[*M*]. *x*=*a*) <−> (*M*(*a*))
by *blast*

lemma *rex-triv-one-point2* [*simp*]: (*EX x*[*M*]. *a*=*x*) <−> (*M*(*a*))
by *blast*

lemma *rex-one-point1* [*simp*]: (*EX x*[*M*]. *x*=*a* & *P*(*x*)) <−> (*M*(*a*) & *P*(*a*))
by *blast*

lemma *rex-one-point2* [*simp*]: (*EX x*[*M*]. *a*=*x* & *P*(*x*)) <−> (*M*(*a*) & *P*(*a*))
by *blast*

lemma *rall-one-point1* [*simp*]: (*ALL x*[*M*]. *x*=*a* −−> *P*(*x*)) <−> (*M*(*a*) −−> *P*(*a*))
by *blast*

lemma *rall-one-point2* [*simp*]: (*ALL x*[*M*]. *a*=*x* −−> *P*(*x*)) <−> (*M*(*a*) −−> *P*(*a*))
by *blast*

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

lemma *rall-setclass-is-ball* [*simp*]: (∀ *x*[##*A*]. *P*(*x*)) <−> (∀ *x*∈*A*. *P*(*x*))
by *auto*

lemma *rex-setclass-is-bex* [*simp*]: (∃ *x*[##*A*]. *P*(*x*)) <−> (∃ *x*∈*A*. *P*(*x*))
by *auto*

ML
⟪
val Ord-atomize =
atomize ([(*OrdQuant.oall*, [@{*thm ospec*}]),(*OrdQuant.rall*, [@{*thm rspec*}])]@
ZF-conn-pairs,
ZF-mem-pairs);
⟫
declaration ⟪ *fn* - =>
Simplifier.map-ss (*fn ss* => *ss setmksimps* (*map mk-eq o Ord-atomize o gen-all*))
⟫

Setting up the one-point-rule simproc

ML ⟪
local

val unfold-rex-tac = unfold-tac [@{*thm rex-def*}];
fun prove-rex-tac ss = unfold-rex-tac ss THEN Quantifier1.prove-one-point-ex-tac;
val rearrange-bex = Quantifier1.rearrange-bex prove-rex-tac;

val unfold-rall-tac = unfold-tac [@{*thm rall-def*}];
fun prove-rall-tac ss = unfold-rall-tac ss THEN Quantifier1.prove-one-point-all-tac;
val rearrange-ball = Quantifier1.rearrange-ball prove-rall-tac;

in

val defREX-regroup = Simplifier.simproc @{*theory*}
  defined REX [*EX x*[*M*]. *P*(*x*) & *Q*(*x*)] *rearrange-bex*;
val defRALL-regroup = Simplifier.simproc @{*theory*}
  defined RALL [*ALL x*[*M*]. *P*(*x*) --> *Q*(*x*)] *rearrange-ball*;

end;

Addsimprocs [*defRALL-regroup*,*defREX-regroup*];
⟫

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*})
apply (*rule bnd-monoI*)
apply (*cut-tac infinity*, *blast*, *blast*)
done

lemmas *nat-unfold* = *nat-bnd-mono* [*THEN nat-def* [*THEN def-lfp-unfold*], *standard*]

lemma *nat-0I* [*iff*,*TC*]: *0* : *nat*
apply (*subst nat-unfold*)
apply (*rule singletonI* [*THEN UnI1*])
done

lemma *nat-succI* [*intro*!,*TC*]: *n* : *nat* ==> *succ*(*n*) : *nat*
apply (*subst nat-unfold*)
apply (*erule RepFunI* [*THEN UnI2*])
done

lemma *nat-1I* [*iff*,*TC*]: *1* : *nat*
by (*rule nat-0I* [*THEN nat-succI*])

lemma *nat-2I* [*iff*,*TC*]: *2* : *nat*

by (*rule nat-1I* [*THEN nat-succI*])

lemma *bool-subset-nat*: *bool* <= *nat*
by (*blast elim*!: *boolE*)

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*)
by (*erule def-induct* [*OF nat-def nat-bnd-mono*], *blast*)

lemma *natE*:
[| *n*: *nat*; *n*=*0* ==> *P*; !!*x*. [| *x*: *nat*; *n*=*succ*(*x*) |] ==> *P* |] ==> *P*
by (*erule nat-unfold* [*THEN equalityD1*, *THEN subsetD*, *THEN UnE*], *auto*)

lemma *nat-into-Ord* [*simp*]: *n*: *nat* ==> *Ord*(*n*)
by (*erule nat-induct*, *auto*)

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*)
apply (*rule OrdI*)
apply (*erule-tac* [*2*] *nat-into-Ord* [*THEN Ord-is-Transset*])
apply (*unfold Transset-def*)
apply (*rule ballI*)
apply (*erule nat-induct*, *auto*)
done

lemma *Limit-nat* [*iff*]: *Limit*(*nat*)
apply (*unfold Limit-def*)
apply (*safe intro*!: *ltI Ord-nat*)
apply (*erule ltD*)
done

lemma *naturals-not-limit*: *a* ∈ *nat* ==> ~ *Limit*(*a*)
by (*induct a rule*: *nat-induct*, *auto*)

lemma *succ-natD*: *succ*(*i*): *nat* ==> *i*: *nat*
by (*rule Ord-trans* [*OF succI1*], *auto*)

lemma *nat-succ-iff* [*iff*]: *succ*(*n*): *nat* <-> *n*: *nat*
by (*blast dest*!: *succ-natD*)

lemma *nat-le-Limit*: *Limit*(*i*) ==> *nat le i*
apply (*rule subset-imp-le*)
apply (*simp-all add*: *Limit-is-Ord*)
apply (*rule subsetI*)
apply (*erule nat-induct*)
apply (*erule Limit-has-0* [*THEN ltD*])
apply (*blast intro*: *Limit-has-succ* [*THEN ltD*] *ltI Limit-is-Ord*)
done

lemmas *succ-in-naturalD* = *Ord-trans* [*OF succI1 - nat-into-Ord*]

lemma *lt-nat-in-nat*: [| *m*<*n*; *n*: *nat* |] ==> *m*: *nat*
apply (*erule ltE*)
apply (*erule Ord-trans*, *assumption*, *simp*)
done

lemma *le-in-nat*: [| *m le n*; *n*:*nat* |] ==> *m*:*nat*
by (*blast dest*!: *lt-nat-in-nat*)

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*)
apply (*erule nat-induct*)
apply (*simp-all add*: *distrib-simps le0-iff le-succ-iff*)
done

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*)
apply (*blast intro*: *nat-induct-from-lemma*)
done

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*)
apply (*erule-tac x* = *m* **in** *rev-bspec*)
apply (*erule nat-induct*, *simp*)
apply (*rule ballI*)
apply (*rename-tac i j*)
apply (*erule-tac n*=*j* **in** *nat-induct*, *auto*)
done

lemma *succ-lt-induct-lemma* [*rule-format*]:
m: *nat* ==> *P*(*m*,*succ*(*m*)) --> (*ALL x*: *nat*. *P*(*m*,*x*) --> *P*(*m*,*succ*(*x*)))
-->
(*ALL n*:*nat*. *m*<*n* --> *P*(*m*,*n*))
apply (*erule nat-induct*)
apply (*intro impI*, *rule nat-induct* [*THEN ballI*])
prefer *4* **apply** (*intro impI*, *rule nat-induct* [*THEN ballI*])
apply (*auto simp add*: *le-iff*)
done

lemma *succ-lt-induct*:
[| *m*<*n*; *n*: *nat*;
P(*m*,*succ*(*m*));
!!*x*. [| *x*: *nat*; *P*(*m*,*x*) |] ==> *P*(*m*,*succ*(*x*)) |]
==> *P*(*m*,*n*)
by (*blast intro*: *succ-lt-induct-lemma lt-nat-in-nat*)

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

lemma [*iff*]: *quasinat*(*succ*(*x*))
by (*simp add*: *quasinat-def*)

lemma *nat-imp-quasinat*: *n* ∈ *nat* ==> *quasinat*(*n*)
by (*erule natE*, *simp-all*)

lemma *non-nat-case*: ~ *quasinat*(*x*) ==> *nat-case*(*a*,*b*,*x*) = *0*
by (*simp add*: *quasinat-def nat-case-def*)

lemma *nat-cases-disj*: *k*=*0* | (∃ *y*. *k* = *succ*(*y*)) | ~ *quasinat*(*k*)
apply (*case-tac k*=*0*, *simp*)
apply (*case-tac* ∃ *m*. *k* = *succ*(*m*))

apply (*simp-all add*: *quasinat-def*)
done

lemma *nat-cases*:
  [|*k*=*0* ==> *P*; !!*y*. *k* = *succ*(*y*) ==> *P*; ~ *quasinat*(*k*) ==> *P*|] ==> *P*
by (*insert nat-cases-disj* [*of k*], *blast*)

lemma *nat-case-0* [*simp*]: *nat-case*(*a*,*b*,*0*) = *a*
by (*simp add*: *nat-case-def*)

lemma *nat-case-succ* [*simp*]: *nat-case*(*a*,*b*,*succ*(*n*)) = *b*(*n*)
by (*simp add*: *nat-case-def*)

lemma *nat-case-type* [*TC*]:
  [| *n*: *nat*; *a*: *C*(*0*); !!*m*. *m*: *nat* ==> *b*(*m*): *C*(*succ*(*m*)) |]
  ==> *nat-case*(*a*,*b*,*n*) : *C*(*n*)
by (*erule nat-induct*, *auto*)

lemma *split-nat-case*:
  P(*nat-case*(*a*,*b*,*k*)) <->
  ((*k*=*0* --> *P*(*a*)) & (∀ *x*. *k*=*succ*(*x*) --> *P*(*b*(*x*))) & (~ *quasinat*(*k*) ⟶ *P*(*0*)))
apply (*rule nat-cases* [*of k*])
apply (*auto simp add*: *non-nat-case*)
done

15.4 Recursion on the Natural Numbers

lemma *nat-rec-0*: *nat-rec*(*0*,*a*,*b*) = *a*
apply (*rule nat-rec-def* [*THEN def-wfrec*, *THEN trans*])
apply (*rule wf-Memrel*)
apply (*rule nat-case-0*)
done

lemma *nat-rec-succ*: *m*: *nat* ==> *nat-rec*(*succ*(*m*),*a*,*b*) = *b*(*m*, *nat-rec*(*m*,*a*,*b*))
apply (*rule nat-rec-def* [*THEN def-wfrec*, *THEN trans*])
apply (*rule wf-Memrel*)
apply (*simp add*: *vimage-singleton-iff*)
done

lemma *Un-nat-type* [*TC*]: [| *i*: *nat*; *j*: *nat* |] ==> *i Un j*: *nat*
apply (*rule Un-least-lt* [*THEN ltD*])
apply (*simp-all add*: *lt-def*)
done

lemma *Int-nat-type* [*TC*]: [| *i*: *nat*; *j*: *nat* |] ==> *i Int j*: *nat*
apply (*rule Int-greatest-lt* [*THEN ltD*])
apply (*simp-all add*: *lt-def*)
done

lemma *nat-nonempty* [*simp*]: *nat* ~= *0*
by *blast*

A natural number is the set of its predecessors

lemma *nat-eq-Collect-lt*: *i* ∈ *nat* ==> {*j*∈*nat*. *j*<*i*} = *i*
apply (*rule equalityI*)
apply (*blast dest*: *ltD*)
apply (*auto simp add*: *Ord-mem-iff-lt*)
apply (*blast intro*: *lt-trans*)
done

lemma *Le-iff* [*iff*]: <*x*,*y*> : *Le* <-> *x le y* & *x* : *nat* & *y* : *nat*
by (*force simp add*: *Le-def*)

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*
by *blast*

lemma *def-trans*: *f* == *g* ==> *g*(*a*) = *b* ==> *f*(*a*) = *b*
by *simp*

lemma *refl-thin*: !!*P*. *a* = *a* ==> *P* ==> *P* **.**

use *ind-syntax.ML*
use *Tools/cartprod.ML*
use *Tools/ind-cases.ML*

use *Tools/inductive-package.ML*
use *Tools/induct-tacs.ML*
use *Tools/primrec-package.ML*

setup *IndCases.setup*
setup *DatatypeTactics.setup*

ML ⟪
```
structure Lfp =
  struct
  val oper       = @{const lfp}
  val bnd-mono   = @{const bnd-mono}
  val bnd-monoI  = @{thm bnd-monoI}
  val subs       = @{thm def-lfp-subset}
  val Tarski     = @{thm def-lfp-unfold}
  val induct     = @{thm def-induct}
  end;

structure Standard-Prod =
  struct
  val sigma      = @{const Sigma}
  val pair       = @{const Pair}
  val split-name = @{const-name split}
  val pair-iff   = @{thm Pair-iff}
  val split-eq   = @{thm split}
  val fsplitI    = @{thm splitI}
  val fsplitD    = @{thm splitD}
  val fsplitE    = @{thm splitE}
  end;

structure Standard-CP = CartProd-Fun (Standard-Prod);

structure Standard-Sum =
  struct
  val sum        = @{const sum}
  val inl        = @{const Inl}
  val inr        = @{const Inr}
  val elim       = @{const case}
  val case-inl   = @{thm case-Inl}
  val case-inr   = @{thm case-Inr}
  val inl-iff    = @{thm Inl-iff}
  val inr-iff    = @{thm Inr-iff}
  val distinct   = @{thm Inl-Inr-iff}
  val distinct'  = @{thm Inr-Inl-iff}
  val free-SEs   = Ind-Syntax.mk-free-SEs
            [distinct, distinct', inl-iff, inr-iff, Standard-Prod.pair-iff]
  end;
```

```
structure Ind-Package =
    Add-inductive-def-Fun
      (structure Fp=Lfp and Pr=Standard-Prod and CP=Standard-CP
       and Su=Standard-Sum val coind = false);


structure Gfp =
  struct
  val oper      = @{const gfp}
  val bnd-mono  = @{const bnd-mono}
  val bnd-monoI = @{thm bnd-monoI}
  val subs      = @{thm def-gfp-subset}
  val Tarski    = @{thm def-gfp-unfold}
  val induct    = @{thm def-Collect-coinduct}
  end;

structure Quine-Prod =
  struct
  val sigma      = @{const QSigma}
  val pair       = @{const QPair}
  val split-name = @{const-name qsplit}
  val pair-iff   = @{thm QPair-iff}
  val split-eq   = @{thm qsplit}
  val fsplitI    = @{thm qsplitI}
  val fsplitD    = @{thm qsplitD}
  val fsplitE    = @{thm qsplitE}
  end;

structure Quine-CP = CartProd-Fun (Quine-Prod);

structure Quine-Sum =
  struct
  val sum        = @{const qsum}
  val inl        = @{const QInl}
  val inr        = @{const QInr}
  val elim       = @{const qcase}
  val case-inl   = @{thm qcase-QInl}
  val case-inr   = @{thm qcase-QInr}
  val inl-iff    = @{thm QInl-iff}
  val inr-iff    = @{thm QInr-iff}
  val distinct   = @{thm QInl-QInr-iff}
  val distinct'  = @{thm QInr-QInl-iff}
  val free-SEs   = Ind-Syntax.mk-free-SEs
            [distinct, distinct', inl-iff, inr-iff, Quine-Prod.pair-iff]
  end;


structure CoInd-Package =
  Add-inductive-def-Fun(structure Fp=Gfp and Pr=Quine-Prod and CP=Quine-CP
```

and Su=Quine-Sum val coind = true);

⟫

end

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*)
apply (*unfold eclose-def*)
apply (*rule nat-rec-0* [*THEN equalityD2*, *THEN subset-trans*])
apply (*rule nat-0I* [*THEN UN-upper*])
done

lemmas *arg-into-eclose* = *arg-subset-eclose* [*THEN subsetD*, *standard*]

lemma *Transset-eclose*: *Transset*(*eclose*(*A*))
apply (*unfold eclose-def Transset-def*)
apply (*rule subsetI* [*THEN ballI*])
apply (*erule UN-E*)
apply (*rule nat-succI* [*THEN UN-I*], *assumption*)
apply (*erule nat-rec-succ* [*THEN ssubst*])
apply (*erule UnionI*, *assumption*)
done

lemmas *eclose-subset* =
Transset-eclose [*unfolded Transset-def*, *THEN bspec*, *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*)
by (*rule arg-in-eclose-sing* [*THEN eclose-induct*], *blast*)

17.2 Leastness of *eclose*

lemma *eclose-least-lemma*:
[| *Transset*(*X*); *A*<=*X*; *n*: *nat* |] ==> *nat-rec*(*n*, *A*, %*m r*. *Union*(*r*)) <= *X*
apply (*unfold Transset-def*)
apply (*erule nat-induct*)
apply (*simp add*: *nat-rec-0*)
apply (*simp add*: *nat-rec-succ*, *blast*)
done

lemma *eclose-least*:
[| *Transset*(*X*); *A*<=*X* |] ==> *eclose*(*A*) <= *X*
apply (*unfold eclose-def*)
apply (*rule eclose-least-lemma* [*THEN UN-least*], *assumption*+)
done

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)
apply (*rule eclose-least [THEN subsetD, THEN CollectD2, of eclose(b)]*)
prefer *3* **apply** *assumption*
apply (*unfold Transset-def*)
apply (*blast intro: ecloseD*)
apply (*blast intro: arg-subset-eclose [THEN subsetD]*)
done

lemma *Transset-eclose-eq-arg*: *Transset(X) ==> eclose(X) = X*
apply (*erule equalityI [OF eclose-least arg-subset-eclose]*)
apply (*rule subset-refl*)
done

A transitive set either is empty or contains the empty set.

lemma *Transset-0-lemma [rule-format]*: *Transset(A) ==> x∈A −−> 0∈A*
apply (*simp add: Transset-def*)
apply (*rule-tac a=x* **in** *eps-induct, clarify*)
apply (*drule bspec, assumption*)
apply (*case-tac x=0, auto*)
done

lemma *Transset-0-disj*: *Transset(A) ==> A=0 | 0∈A*
by (*blast dest: Transset-0-lemma*)

17.3 Epsilon Recursion

lemma *mem-eclose-trans*: [| *A*: *eclose(B)*; *B*: *eclose(C)* |] ==> *A*: *eclose(C)*
by (*rule eclose-least [OF Transset-eclose eclose-subset, THEN subsetD], assumption+*)

lemma *mem-eclose-sing-trans*:
[| *A*: *eclose({B})*; *B*: *eclose({C})* |] ==> *A*: *eclose({C})*
by (*rule eclose-least [OF Transset-eclose singleton-subsetI, THEN subsetD], assumption+*)

lemma *under-Memrel*: [| *Transset(i)*; *j:i* |] ==> *Memrel(i)−``{j} = j*
by (*unfold Transset-def, blast*)

lemma *lt-Memrel*: *j < i ==> Memrel(i) −`` {j} = j*
by (*simp add: lt-def Ord-def under-Memrel*)

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*)
apply (*erule eclose-induct*)
apply (*rule wfrec-ssubst*)
apply (*rule wfrec-ssubst*)
apply (*simp add*: *under-Memrel-eclose mem-eclose-sing-trans* [*of* - *j i*])
done

lemma *wfrec-eclose-eq2*:
k: *i* ==> *wfrec*(*Memrel*(*eclose*({*i*})),*k*,*H*) = *wfrec*(*Memrel*(*eclose*({*k*})),*k*,*H*)
apply (*rule arg-in-eclose-sing* [*THEN wfrec-eclose-eq*])
apply (*erule arg-into-eclose-sing*)
done

lemma *transrec*: *transrec*(*a*,*H*) = *H*(*a*, *lam x*:*a*. *transrec*(*x*,*H*))
apply (*unfold transrec-def*)
apply (*rule wfrec-ssubst*)
apply (*simp add*: *wfrec-eclose-eq2 arg-in-eclose-sing under-Memrel-eclose*)
done

lemma *def-transrec*:
[| !!*x*. *f*(*x*)==*transrec*(*x*,*H*) |] ==> *f*(*a*) = *H*(*a*, *lam x*:*a*. *f*(*x*))
apply *simp*
apply (*rule transrec*)
done

lemma *transrec-type*:
[| !!*x u*. [| *x*:*eclose*({*a*}); *u*: *Pi*(*x*,*B*) |] ==> *H*(*x*,*u*) : *B*(*x*) |]
==> *transrec*(*a*,*H*) : *B*(*a*)
apply (*rule-tac i* = *a* **in** *arg-in-eclose-sing* [*THEN eclose-induct*])
apply (*subst transrec*)
apply (*simp add*: *lam-type*)
done

lemma *eclose-sing-Ord*: *Ord*(*i*) ==> *eclose*({*i*}) <= *succ*(*i*)
apply (*erule Ord-is-Transset* [*THEN Transset-succ*, *THEN eclose-least*])
apply (*rule succI1* [*THEN singleton-subsetI*])
done

lemma *succ-subset-eclose-sing*: *succ*(*i*) <= *eclose*({*i*})
apply (*insert arg-subset-eclose* [*of* {*i*}], *simp*)
apply (*frule eclose-subset*, *blast*)
done

lemma *eclose-sing-Ord-eq*: *Ord*(*i*) ==> *eclose*({*i*}) = *succ*(*i*)
apply (*rule equalityI*)
apply (*erule eclose-sing-Ord*)

apply (*rule succ-subset-eclose-sing*)
done

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*)
apply (*rule transrec-type*)
apply (*insert jini ordi*)
apply (*blast intro*!: *minor*
    intro: *Ord-trans*
    dest: *Ord-in-Ord* [*THEN eclose-sing-Ord*, *THEN subsetD*])
done

17.4 Rank

lemma *rank*: *rank*(*a*) = ($\bigcup$ *y*∈*a*. *succ*(*rank*(*y*)))
by (*subst rank-def* [*THEN def-transrec*], *simp*)

lemma *Ord-rank* [*simp*]: *Ord*(*rank*(*a*))
apply (*rule-tac a*=*a* **in** *eps-induct*)
apply (*subst rank*)
apply (*rule Ord-succ* [*THEN Ord-UN*])
apply (*erule bspec*, *assumption*)
done

lemma *rank-of-Ord*: *Ord*(*i*) ==> *rank*(*i*) = *i*
apply (*erule trans-induct*)
apply (*subst rank*)
apply (*simp add*: *Ord-equality*)
done

lemma *rank-lt*: *a*:*b* ==> *rank*(*a*) < *rank*(*b*)
apply (*rule-tac a1* = *b* **in** *rank* [*THEN ssubst*])
apply (*erule UN-I* [*THEN ltI*])
apply (*rule-tac* [*2*] *Ord-UN*, *auto*)
done

lemma *eclose-rank-lt*: *a*: *eclose*(*b*) ==> *rank*(*a*) < *rank*(*b*)
apply (*erule eclose-induct-down*)
apply (*erule rank-lt*)
apply (*erule rank-lt* [*THEN lt-trans*], *assumption*)
done

lemma *rank-mono*: *a*<=*b* ==> *rank*(*a*) *le rank*(*b*)
apply (*rule subset-imp-le*)
apply (*auto simp add*: *rank* [*of a*] *rank* [*of b*])
done

lemma *rank-Pow*: *rank*(*Pow*(*a*)) = *succ*(*rank*(*a*))
apply (*rule rank* [*THEN trans*])
apply (*rule le-anti-sym*)
apply (*rule-tac* [*2*] *UN-upper-le*)
apply (*rule UN-least-le*)
apply (*auto intro*: *rank-mono simp add*: *Ord-UN*)
done

lemma *rank-0* [*simp*]: *rank*(*0*) = *0*
by (*rule rank* [*THEN trans*], *blast*)

lemma *rank-succ* [*simp*]: *rank*(*succ*(*x*)) = *succ*(*rank*(*x*))
apply (*rule rank* [*THEN trans*])
apply (*rule equalityI* [*OF UN-least succI1* [*THEN UN-upper*]])
apply (*erule succE*, *blast*)
apply (*erule rank-lt* [*THEN leI*, *THEN succ-leI*, *THEN le-imp-subset*])
done

lemma *rank-Union*: *rank*(*Union*(*A*)) = ($\bigcup$*x*∈*A*. *rank*(*x*))
apply (*rule equalityI*)
apply (*rule-tac* [*2*] *rank-mono* [*THEN le-imp-subset*, *THEN UN-least*])
apply (*erule-tac* [*2*] *Union-upper*)
apply (*subst rank*)
apply (*rule UN-least*)
apply (*erule UnionE*)
apply (*rule subset-trans*)
apply (*erule-tac* [*2*] *RepFunI* [*THEN Union-upper*])
apply (*erule rank-lt* [*THEN succ-leI*, *THEN le-imp-subset*])
done

lemma *rank-eclose*: *rank*(*eclose*(*a*)) = *rank*(*a*)
apply (*rule le-anti-sym*)
apply (*rule-tac* [*2*] *arg-subset-eclose* [*THEN rank-mono*])
apply (*rule-tac a1* = *eclose* (*a*) **in** *rank* [*THEN ssubst*])
apply (*rule Ord-rank* [*THEN UN-least-le*])
apply (*erule eclose-rank-lt* [*THEN succ-leI*])
done

lemma *rank-pair1*: *rank*(*a*) < *rank*(<*a*,*b*>)
apply (*unfold Pair-def*)
apply (*rule consI1* [*THEN rank-lt*, *THEN lt-trans*])
apply (*rule consI1* [*THEN consI2*, *THEN rank-lt*])
done

lemma *rank-pair2*: *rank*(*b*) < *rank*(<*a*,*b*>)
apply (*unfold Pair-def*)
apply (*rule consI1* [*THEN consI2*, *THEN rank-lt*, *THEN lt-trans*])
apply (*rule consI1* [*THEN consI2*, *THEN rank-lt*])

done

lemma *the-equality-if*:
  P(*a*) ==> (*THE x*. *P*(*x*)) = (*if* (*EX*!*x*. *P*(*x*)) *then a else 0*)
by (*simp add*: *the-0 the-equality2*)

lemma *rank-apply*: [|*i* : *domain*(*f*); *function*(*f*)|] ==> *rank*(*f*‘*i*) < *rank*(*f*)
apply *clarify*
apply (*simp add*: *function-apply-equality*)
apply (*blast intro*: *lt-trans rank-lt rank-pair2*)
done

17.5 Corollaries of Leastness

lemma *mem-eclose-subset*: *A*:*B* ==> *eclose*(*A*)<=*eclose*(*B*)
apply (*rule Transset-eclose* [*THEN eclose-least*])
apply (*erule arg-into-eclose* [*THEN eclose-subset*])
done

lemma *eclose-mono*: *A*<=*B* ==> *eclose*(*A*) <= *eclose*(*B*)
apply (*rule Transset-eclose* [*THEN eclose-least*])
apply (*erule subset-trans*)
apply (*rule arg-subset-eclose*)
done

lemma *eclose-idem*: *eclose*(*eclose*(*A*)) = *eclose*(*A*)
apply (*rule equalityI*)
apply (*rule eclose-least* [*OF Transset-eclose subset-refl*])
apply (*rule arg-subset-eclose*)
done

lemma *transrec2-0* [*simp*]: *transrec2*(*0*,*a*,*b*) = *a*
by (*rule transrec2-def* [*THEN def-transrec*, *THEN trans*], *simp*)

lemma *transrec2-succ* [*simp*]: *transrec2*(*succ*(*i*),*a*,*b*) = *b*(*i*, *transrec2*(*i*,*a*,*b*))
apply (*rule transrec2-def* [*THEN def-transrec*, *THEN trans*])
apply (*simp add*: *the-equality if-P*)
done

lemma *transrec2-Limit*:
  Limit(*i*) ==> *transrec2*(*i*,*a*,*b*) = (⋃*j*<*i*. *transrec2*(*j*,*a*,*b*))
apply (*rule transrec2-def* [*THEN def-transrec*, *THEN trans*])
apply (*auto simp add*: *OUnion-def*)

done

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*)))
by (*simp add*: *transrec2-Limit*)

lemmas *recursor-lemma* = *recursor-def* [*THEN def-transrec*, *THEN trans*]

lemma *recursor-0*: *recursor*(*a*,*b*,*0*) = *a*
by (*rule nat-case-0* [*THEN recursor-lemma*])

lemma *recursor-succ*: *recursor*(*a*,*b*,*succ*(*m*)) = *b*(*m*, *recursor*(*a*,*b*,*m*))
by (*rule recursor-lemma*, *simp*)

lemma *rec-0* [*simp*]: *rec*(*0*,*a*,*b*) = *a*
apply (*unfold rec-def*)
apply (*rule recursor-0*)
done

lemma *rec-succ* [*simp*]: *rec*(*succ*(*m*),*a*,*b*) = *b*(*m*, *rec*(*m*,*a*,*b*))
apply (*unfold rec-def*)
apply (*rule recursor-succ*)
done

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*)
by (*erule nat-induct*, *auto*)

ML
⟪
val arg-subset-eclose = *thm arg-subset-eclose*;
val arg-into-eclose = *thm arg-into-eclose*;
val Transset-eclose = *thm Transset-eclose*;
val eclose-subset = *thm eclose-subset*;
val ecloseD = *thm ecloseD*;
val arg-in-eclose-sing = *thm arg-in-eclose-sing*;

val arg-into-eclose-sing = thm arg-into-eclose-sing;
val eclose-induct = thm eclose-induct;
val eps-induct = thm eps-induct;
val eclose-least = thm eclose-least;
val eclose-induct-down = thm eclose-induct-down;
val Transset-eclose-eq-arg = thm Transset-eclose-eq-arg;
val mem-eclose-trans = thm mem-eclose-trans;
val mem-eclose-sing-trans = thm mem-eclose-sing-trans;
val under-Memrel = thm under-Memrel;
val under-Memrel-eclose = thm under-Memrel-eclose;
val wfrec-ssubst = thm wfrec-ssubst;
val wfrec-eclose-eq = thm wfrec-eclose-eq;
val wfrec-eclose-eq2 = thm wfrec-eclose-eq2;
val transrec = thm transrec;
val def-transrec = thm def-transrec;
val transrec-type = thm transrec-type;
val eclose-sing-Ord = thm eclose-sing-Ord;
val Ord-transrec-type = thm Ord-transrec-type;
val rank = thm rank;
val Ord-rank = thm Ord-rank;
val rank-of-Ord = thm rank-of-Ord;
val rank-lt = thm rank-lt;
val eclose-rank-lt = thm eclose-rank-lt;
val rank-mono = thm rank-mono;
val rank-Pow = thm rank-Pow;
val rank-0 = thm rank-0;
val rank-succ = thm rank-succ;
val rank-Union = thm rank-Union;
val rank-eclose = thm rank-eclose;
val rank-pair1 = thm rank-pair1;
val rank-pair2 = thm rank-pair2;
val the-equality-if = thm the-equality-if;
val rank-apply = thm rank-apply;
val mem-eclose-subset = thm mem-eclose-subset;
val eclose-mono = thm eclose-mono;
val eclose-idem = thm eclose-idem;
val transrec2-0 = thm transrec2-0;
val transrec2-succ = thm transrec2-succ;
val transrec2-Limit = thm transrec2-Limit;
val recursor-0 = thm recursor-0;
val recursor-succ = thm recursor-succ;
val rec-0 = thm rec-0;
val rec-succ = thm rec-succ;
val rec-type = thm rec-type;
⟩⟩

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* | *x*=*y* | <*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*) ≡ *refl*(*A*, *r*) ∧ *trans*[*A*](*r*)

definition
partial-order-on(*A*, *r*) ≡ *preorder-on*(*A*, *r*) ∧ *antisym*(*r*)

abbreviation
Preorder(*r*) ≡ *preorder-on*(*field*(*r*), *r*)

abbreviation
Partial-order(*r*) ≡ *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*) ==
⋃*x*∈*A*. ⋃*y*∈*B*. ⋃*f* ∈ *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*~=*u* −−> <*u*,*v*> : *R*)

notation (*xsymbols*)
ord-iso ((⟨-, -⟩ ≅/ ⟨-, -⟩) *51*)

18.1 Immediate Consequences of the Definitions

lemma *part-ord-Imp-asym*:
part-ord(*A*,*r*) ==> *asym*(*r Int A*∗*A*)
by (*unfold part-ord-def irrefl-def trans-on-def asym-def*, *blast*)

lemma *linearE*:
[| *linear*(*A*,*r*); *x*:*A*; *y*:*A*;
<*x*,*y*>:*r* ==> *P*; *x*=*y* ==> *P*; <*y*,*x*>:*r* ==> *P* |]
==> *P*
by (*simp add*: *linear-def*, *blast*)

lemma *well-ordI*:
[| *wf*[*A*](*r*); *linear*(*A*,*r*) |] ==> *well-ord*(*A*,*r*)
apply (*simp add*: *irrefl-def part-ord-def tot-ord-def*
trans-on-def well-ord-def wf-on-not-refl)
apply (*fast elim*: *linearE wf-on-asym wf-on-chain3*)
done

lemma *well-ord-is-wf*:
well-ord(*A*,*r*) ==> *wf*[*A*](*r*)
by (*unfold well-ord-def*, *safe*)

lemma *well-ord-is-trans-on*:
well-ord(*A*,*r*) ==> *trans*[*A*](*r*)
by (*unfold well-ord-def tot-ord-def part-ord-def*, *safe*)

lemma *well-ord-is-linear*: *well-ord*(*A*,*r*) ==> *linear*(*A*,*r*)
by (*unfold well-ord-def tot-ord-def*, *blast*)

lemma *pred-iff*: *y* : *pred*(*A*,*x*,*r*) <−> <*y*,*x*>:*r* & *y*:*A*
by (*unfold pred-def*, *blast*)

lemmas *predI* = *conjI* [*THEN pred-iff* [*THEN iffD2*]]

lemma *predE*: [| *y*: *pred*(*A*,*x*,*r*); [| *y*:*A*; <*y*,*x*>:*r* |] ==> *P* |] ==> *P*
by (*simp add*: *pred-def*)

lemma *pred-subset-under*: *pred*(*A*,*x*,*r*) <= *r* −‘‘ {*x*}
by (*simp add*: *pred-def*, *blast*)

lemma *pred-subset*: *pred*(*A*,*x*,*r*) <= *A*
by (*simp add*: *pred-def*, *blast*)

lemma *pred-pred-eq*:
    pred(*pred*(*A*,*x*,*r*), *y*, *r*) = *pred*(*A*,*x*,*r*) *Int pred*(*A*,*y*,*r*)
by (*simp add*: *pred-def*, *blast*)

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*)
by (*unfold trans-on-def pred-def*, *blast*)

18.2 Restricting an Ordering's Domain

lemma *part-ord-subset*:
    [| *part-ord*(*A*,*r*); *B*<=*A* |] ==> *part-ord*(*B*,*r*)
by (*unfold part-ord-def irrefl-def trans-on-def*, *blast*)

lemma *linear-subset*:
    [| *linear*(*A*,*r*); *B*<=*A* |] ==> *linear*(*B*,*r*)
by (*unfold linear-def*, *blast*)

lemma *tot-ord-subset*:
    [| *tot-ord*(*A*,*r*); *B*<=*A* |] ==> *tot-ord*(*B*,*r*)
apply (*unfold tot-ord-def*)
apply (*fast elim*!: *part-ord-subset linear-subset*)
done

lemma *well-ord-subset*:
    [| *well-ord*(*A*,*r*); *B*<=*A* |] ==> *well-ord*(*B*,*r*)
apply (*unfold well-ord-def*)
apply (*fast elim*!: *tot-ord-subset wf-on-subset-A*)
done

lemma *irrefl-Int-iff*: *irrefl*(*A*,*r Int A*∗*A*) <−> *irrefl*(*A*,*r*)

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

lemma *trans-on-Int-iff*: *trans*[*A*](*r Int A*∗*A*) <−> *trans*[*A*](*r*)
by (*unfold trans-on-def*, *blast*)

lemma *part-ord-Int-iff*: *part-ord*(*A*,*r Int A*∗*A*) <−> *part-ord*(*A*,*r*)
apply (*unfold part-ord-def*)
apply (*simp add*: *irrefl-Int-iff trans-on-Int-iff*)
done

lemma *linear-Int-iff*: *linear*(*A*,*r Int A*∗*A*) <−> *linear*(*A*,*r*)
by (*unfold linear-def*, *blast*)

lemma *tot-ord-Int-iff*: *tot-ord*(*A*,*r Int A*∗*A*) <−> *tot-ord*(*A*,*r*)
apply (*unfold tot-ord-def*)
apply (*simp add*: *part-ord-Int-iff linear-Int-iff*)
done

lemma *wf-on-Int-iff*: *wf*[*A*](*r Int A*∗*A*) <−> *wf*[*A*](*r*)
apply (*unfold wf-on-def wf-def*, *fast*)
done

lemma *well-ord-Int-iff*: *well-ord*(*A*,*r Int A*∗*A*) <−> *well-ord*(*A*,*r*)
apply (*unfold well-ord-def*)
apply (*simp add*: *tot-ord-Int-iff wf-on-Int-iff*)
done

18.3 Empty and Unit Domains

lemma *wf-on-any-0*: *wf*[*A*](*0*)
by (*simp add*: *wf-on-def wf-def*, *fast*)

18.3.1 Relations over the Empty Set

lemma *irrefl-0*: *irrefl*(*0*,*r*)
by (*unfold irrefl-def*, *blast*)

lemma *trans-on-0*: *trans*[*0*](*r*)
by (*unfold trans-on-def*, *blast*)

lemma *part-ord-0*: *part-ord*(*0*,*r*)
apply (*unfold part-ord-def*)
apply (*simp add*: *irrefl-0 trans-on-0*)
done

lemma *linear-0*: *linear*(*0*,*r*)
by (*unfold linear-def*, *blast*)

lemma *tot-ord-0*: *tot-ord*(*0*,*r*)
apply (*unfold tot-ord-def*)

apply (*simp add*: *part-ord-0 linear-0*)
done

lemma *wf-on-0*: *wf*[*0*](*r*)
by (*unfold wf-on-def wf-def*, *blast*)

lemma *well-ord-0*: *well-ord*(*0*,*r*)
apply (*unfold well-ord-def*)
apply (*simp add*: *tot-ord-0 wf-on-0*)
done

18.3.2 The Empty Relation Well-Orders the Unit Set

by Grabczewski

lemma *tot-ord-unit*: *tot-ord*({*a*},*0*)
by (*simp add*: *irrefl-def trans-on-def part-ord-def linear-def tot-ord-def*)

lemma *well-ord-unit*: *well-ord*({*a*},*0*)
apply (*unfold well-ord-def*)
apply (*simp add*: *tot-ord-unit wf-on-any-0*)
done

18.4 Order-Isomorphisms

Suppes calls them "similarities"

lemma *mono-map-is-fun*: *f*: *mono-map*(*A*,*r*,*B*,*s*) ==> *f*: *A*−>*B*
by (*simp add*: *mono-map-def*)

lemma *mono-map-is-inj*:
  [| *linear*(*A*,*r*); *wf*[*B*](*s*); *f*: *mono-map*(*A*,*r*,*B*,*s*) |] ==> *f*: *inj*(*A*,*B*)
apply (*unfold mono-map-def inj-def*, *clarify*)
apply (*erule-tac x*=*w* **and** *y*=*x* **in** *linearE*, *assumption*+)
apply (*force intro*: *apply-type dest*: *wf-on-not-refl*)+
done

lemma *ord-isoI*:
  [| *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*)
by (*simp add*: *ord-iso-def*)

lemma *ord-iso-is-mono-map*:
  f: *ord-iso*(*A*,*r*,*B*,*s*) ==> *f*: *mono-map*(*A*,*r*,*B*,*s*)
apply (*simp add*: *ord-iso-def mono-map-def*)
apply (*blast dest*!: *bij-is-fun*)
done

lemma *ord-iso-is-bij*:

f: *ord-iso*(*A*,*r*,*B*,*s*) ==> *f*: *bij*(*A*,*B*)
by (*simp add*: *ord-iso-def*)

lemma *ord-iso-apply*:
[| *f*: *ord-iso*(*A*,*r*,*B*,*s*); <*x*,*y*>: *r*; *x*:*A*; *y*:*A* |] ==> <*f*'*x*, *f*'*y*> : *s*
by (*simp add*: *ord-iso-def*)

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*
apply (*simp add*: *ord-iso-def*, *clarify*)
apply (*erule bspec* [*THEN bspec*, *THEN iffD2*])
apply (*erule asm-rl bij-converse-bij* [*THEN bij-is-fun*, *THEN apply-type*])+
apply (*auto simp add*: *right-inverse-bij*)
done

lemma *ord-iso-refl*: *id*(*A*): *ord-iso*(*A*,*r*,*A*,*r*)
by (*rule id-bij* [*THEN ord-isoI*], *simp*)

lemma *ord-iso-sym*: *f*: *ord-iso*(*A*,*r*,*B*,*s*) ==> *converse*(*f*): *ord-iso*(*B*,*s*,*A*,*r*)
apply (*simp add*: *ord-iso-def*)
apply (*auto simp add*: *right-inverse-bij bij-converse-bij*
bij-is-fun [*THEN apply-funtype*])
done

lemma *mono-map-trans*:
[| *g*: *mono-map*(*A*,*r*,*B*,*s*); *f*: *mono-map*(*B*,*s*,*C*,*t*) |]
==> (*f O g*): *mono-map*(*A*,*r*,*C*,*t*)
apply (*unfold mono-map-def*)
apply (*auto simp add*: *comp-fun*)
done

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*)
apply (*unfold ord-iso-def*, *clarify*)
apply (*frule bij-is-fun* [*of f*])
apply (*frule bij-is-fun* [*of g*])
apply (*auto simp add*: *comp-bij*)
done

lemma *mono-ord-isoI*:
[| *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*)
apply (*simp add*: *ord-iso-def mono-map-def*, *safe*)
apply (*intro fg-imp-bijective*, *auto*)
apply (*subgoal-tac* <*g*‘ (*f*‘*x*), *g*‘ (*f*‘*y*) > : *r*)
apply (*simp add*: *comp-eq-id-iff* [*THEN iffD1*])
apply (*blast intro*: *apply-funtype*)
done

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*)
apply (*intro mono-ord-isoI*, *auto*)
apply (*frule mono-map-is-fun* [*THEN fun-is-rel*])
apply (*erule converse-converse* [*THEN subst*], *rule left-comp-inverse*)
apply (*blast intro*: *left-comp-inverse mono-map-is-inj well-ord-is-linear*
well-ord-is-wf)+
done

lemma *part-ord-ord-iso*:
[| *part-ord*(*B*,*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *part-ord*(*A*,*r*)
apply (*simp add*: *part-ord-def irrefl-def trans-on-def ord-iso-def*)
apply (*fast intro*: *bij-is-fun* [*THEN apply-type*])
done

lemma *linear-ord-iso*:
[| *linear*(*B*,*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *linear*(*A*,*r*)
apply (*simp add*: *linear-def ord-iso-def*, *safe*)
apply (*drule-tac x1* = *f*‘*x* **and** *x* = *f*‘*y* **in** *bspec* [*THEN bspec*])
apply (*safe elim*!: *bij-is-fun* [*THEN apply-type*])
apply (*drule-tac t* = *op* ‘ (*converse* (*f*)) **in** *subst-context*)
apply (*simp add*: *left-inverse-bij*)
done

lemma *wf-on-ord-iso*:
[| *wf*[*B*](*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *wf*[*A*](*r*)
apply (*simp add*: *wf-on-def wf-def ord-iso-def*, *safe*)
apply (*drule-tac x* = {*f*‘*z*. *z*:*Z Int A*} **in** *spec*)
apply (*safe intro*!: *equalityI*)
apply (*blast dest*!: *equalityD1 intro*: *bij-is-fun* [*THEN apply-type*])+
done

lemma *well-ord-ord-iso*:
  [| *well-ord*(*B*,*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *well-ord*(*A*,*r*)
apply (*unfold well-ord-def tot-ord-def*)
apply (*fast elim*!: *part-ord-ord-iso linear-ord-iso wf-on-ord-iso*)
done

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*
apply (*simp add*: *well-ord-def ord-iso-def*)
apply (*elim conjE CollectE*)
apply (*rule-tac a*=*y* **in** *wf-on-induct*, *assumption*+)
apply (*blast dest*: *bij-is-fun* [*THEN apply-type*])
done

lemma *well-ord-iso-predE*:
  [| *well-ord*(*A*,*r*); *f* : *ord-iso*(*A*, *r*, *pred*(*A*,*x*,*r*), *r*); *x*:*A* |] ==> *P*
apply (*insert well-ord-iso-subset-lemma* [*of A r f pred*(*A*,*x*,*r*) *x*])
apply (*simp add*: *pred-subset*)

apply (*drule ord-iso-is-bij* [*THEN bij-is-fun*, *THEN apply-type*], *assumption*)

apply (*simp add*: *well-ord-def pred-def*)
done

lemma *well-ord-iso-pred-eq*:
  [| *well-ord*(*A*,*r*); *f* : *ord-iso*(*pred*(*A*,*a*,*r*), *r*, *pred*(*A*,*c*,*r*), *r*);
    a:*A*; *c*:*A* |] ==> *a*=*c*
apply (*frule well-ord-is-trans-on*)
apply (*frule well-ord-is-linear*)
apply (*erule-tac x*=*a* **and** *y*=*c* **in** *linearE*, *assumption*+)
apply (*drule ord-iso-sym*)

apply (*auto elim*!: *well-ord-subset* [*OF - pred-subset*, *THEN well-ord-iso-predE*]
    intro!: *predI*
    simp add: *trans-pred-pred-eq*)
done

lemma *ord-iso-image-pred*:
  [|*f* : *ord-iso*(*A*,*r*,*B*,*s*); *a*:*A*|] ==> *f* `` *pred*(*A*,*a*,*r*) = *pred*(*B*, *f*'*a*, *s*)
apply (*unfold ord-iso-def pred-def*)
apply (*erule CollectE*)
apply (*simp* (*no-asm-simp*) *add*: *image-fun* [*OF bij-is-fun Collect-subset*])
apply (*rule equalityI*)

apply (*safe elim*!: *bij-is-fun* [*THEN apply-type*])
apply (*rule RepFun-eqI*)
apply (*blast intro*!: *right-inverse-bij* [*symmetric*])
apply (*auto simp add*: *right-inverse-bij bij-is-fun* [*THEN apply-funtype*])
done

lemma *ord-iso-restrict-image*:
[| *f* : *ord-iso*(*A*,*r*,*B*,*s*); *C*<=*A* |]
==> *restrict*(*f*,*C*) : *ord-iso*(*C*, *r*, *f*‘‘*C*, *s*)
apply (*simp add*: *ord-iso-def*)
apply (*blast intro*: *bij-is-inj restrict-bij*)
done

lemma *ord-iso-restrict-pred*:
[| *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*)
apply (*simp add*: *ord-iso-image-pred* [*symmetric*])
apply (*blast intro*: *ord-iso-restrict-image elim*: *predE*)
done

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*
apply (*frule ord-iso-is-bij* [*THEN bij-is-fun*, *THEN apply-type*], (*erule asm-rl predI predE*)+)
apply (*subgoal-tac b* = *g*‘*a*)
apply (*simp* (*no-asm-simp*))
apply (*rule well-ord-iso-pred-eq*, *auto*)
apply (*frule ord-iso-restrict-pred*, (*erule asm-rl predI*)+)
apply (*simp add*: *well-ord-is-trans-on trans-pred-pred-eq*)
apply (*erule ord-iso-sym* [*THEN ord-iso-trans*], *assumption*)
done

lemma *well-ord-iso-unique-lemma*:
[| *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*
apply (*frule well-ord-iso-subset-lemma*)
apply (*rule-tac f* = *converse* (*f*) **and** *g* = *g* **in** *ord-iso-trans*)
apply *auto*
apply (*blast intro*: *ord-iso-sym*)
apply (*frule ord-iso-is-bij* [*of f*])
apply (*frule ord-iso-is-bij* [*of g*])
apply (*frule ord-iso-converse*)

apply (*blast intro*!: *bij-converse-bij*
intro: *bij-is-fun apply-funtype*)+
apply (*erule notE*)
apply (*simp add*: *left-inverse-bij bij-is-fun comp-fun-apply* [*of* - *A B*])
done

lemma *well-ord-iso-unique*: [| *well-ord*(*A*,*r*);
f: *ord-iso*(*A*,*r*, *B*,*s*); *g*: *ord-iso*(*A*,*r*, *B*,*s*) |] ==> *f* = *g*
apply (*rule fun-extension*)
apply (*erule ord-iso-is-bij* [*THEN bij-is-fun*])+
apply (*subgoal-tac f*‘*x* : *B* & *g*‘*x* : *B* & *linear*(*B*,*s*))
apply (*simp add*: *linear-def*)
apply (*blast dest*: *well-ord-iso-unique-lemma*)
apply (*blast intro*: *ord-iso-is-bij bij-is-fun apply-funtype*
well-ord-is-linear well-ord-ord-iso ord-iso-sym)
done

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*
by (*unfold ord-iso-map-def*, *blast*)

lemma *domain-ord-iso-map*: *domain*(*ord-iso-map*(*A*,*r*,*B*,*s*)) <= *A*
by (*unfold ord-iso-map-def*, *blast*)

lemma *range-ord-iso-map*: *range*(*ord-iso-map*(*A*,*r*,*B*,*s*)) <= *B*
by (*unfold ord-iso-map-def*, *blast*)

lemma *converse-ord-iso-map*:
converse(*ord-iso-map*(*A*,*r*,*B*,*s*)) = *ord-iso-map*(*B*,*s*,*A*,*r*)
apply (*unfold ord-iso-map-def*)
apply (*blast intro*: *ord-iso-sym*)
done

lemma *function-ord-iso-map*:
well-ord(*B*,*s*) ==> *function*(*ord-iso-map*(*A*,*r*,*B*,*s*))
apply (*unfold ord-iso-map-def function-def*)
apply (*blast intro*: *well-ord-iso-pred-eq ord-iso-sym ord-iso-trans*)
done

lemma *ord-iso-map-fun*: *well-ord*(*B*,*s*) ==> *ord-iso-map*(*A*,*r*,*B*,*s*)
: *domain*(*ord-iso-map*(*A*,*r*,*B*,*s*)) −> *range*(*ord-iso-map*(*A*,*r*,*B*,*s*))
by (*simp add*: *Pi-iff function-ord-iso-map*
ord-iso-map-subset [*THEN domain-times-range*])

lemma *ord-iso-map-mono-map*:

[| *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*)
apply (*unfold mono-map-def*)
apply (*simp* (*no-asm-simp*) *add*: *ord-iso-map-fun*)
apply *safe*
apply (*subgoal-tac x*:*A* & *ya*:*A* & *y*:*B* & *yb*:*B*)
apply (*simp add*: *apply-equality* [*OF* - *ord-iso-map-fun*])
apply (*unfold ord-iso-map-def*)
apply (*blast intro*: *well-ord-iso-preserving*, *blast*)
done

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*)
apply (*rule well-ord-mono-ord-isoI*)
prefer *4*
apply (*rule converse-ord-iso-map* [*THEN subst*])
apply (*simp add*: *ord-iso-map-mono-map*
ord-iso-map-subset [*THEN converse-converse*])
apply (*blast intro*!: *domain-ord-iso-map range-ord-iso-map*
intro: *well-ord-subset ord-iso-map-mono-map*)+
done

lemma *domain-ord-iso-map-subset*:
[| *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*)
apply (*unfold ord-iso-map-def*)
apply (*safe intro*!: *predI*)

apply (*simp* (*no-asm-simp*))
apply (*frule-tac A* = *A* **in** *well-ord-is-linear*)
apply (*rename-tac b y f*)
apply (*erule-tac x*=*b* **and** *y*=*a* **in** *linearE*, *assumption*+)

apply *clarify*
apply *blast*

apply (*frule ord-iso-is-bij* [*THEN bij-is-fun*, *THEN apply-type*],
(*erule asm-rl predI predE*)+)
apply (*frule ord-iso-restrict-pred*)
apply (*simp add*: *pred-iff*)
apply (*simp split*: *split-if-asm*
add: *well-ord-is-trans-on trans-pred-pred-eq domain-UN domain-Union*,

```
blast)
done


lemma domain-ord-iso-map-cases:
     [| 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))
apply (frule well-ord-is-wf)
apply (unfold wf-on-def wf-def)
apply (drule-tac x = A−domain (ord-iso-map (A,r,B,s)) in spec)
apply safe

apply (rule domain-ord-iso-map [THEN equalityI])
apply (erule Diff-eq-0-iff [THEN iffD1])

apply (blast del: domainI subsetI
             elim!: predE
             intro!: domain-ord-iso-map-subset
             intro: subsetI)+
done


lemma range-ord-iso-map-cases:
    [| 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))
apply (rule converse-ord-iso-map [THEN subst])
apply (simp add: domain-ord-iso-map-cases)
done
```

Kunen's Theorem 6.3: Fundamental Theorem for Well-Ordered Sets

```
theorem well-ord-trichotomy:
   [| 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))
apply (frule-tac B = B in domain-ord-iso-map-cases, assumption)
apply (frule-tac B = B in range-ord-iso-map-cases, assumption)
apply (drule ord-iso-map-ord-iso, assumption)
apply (elim disjE bexE)
   apply (simp-all add: bexI)
apply (rule wf-on-not-refl [THEN notE])
  apply (erule well-ord-is-wf)
 apply assumption
apply (subgoal-tac <x,y>: ord-iso-map (A,r,B,s) )
 apply (drule rangeI)
 apply (simp add: pred-def)
apply (unfold ord-iso-map-def, blast)
```

done

18.7 Miscellaneous Results by Krzysztof Grabczewski

lemma *irrefl-converse*: *irrefl*(*A*,*r*) ==> *irrefl*(*A*,*converse*(*r*))
by (*unfold irrefl-def*, *blast*)

lemma *trans-on-converse*: *trans*[*A*](*r*) ==> *trans*[*A*](*converse*(*r*))
by (*unfold trans-on-def*, *blast*)

lemma *part-ord-converse*: *part-ord*(*A*,*r*) ==> *part-ord*(*A*,*converse*(*r*))
apply (*unfold part-ord-def*)
apply (*blast intro*!: *irrefl-converse trans-on-converse*)
done

lemma *linear-converse*: *linear*(*A*,*r*) ==> *linear*(*A*,*converse*(*r*))
by (*unfold linear-def*, *blast*)

lemma *tot-ord-converse*: *tot-ord*(*A*,*r*) ==> *tot-ord*(*A*,*converse*(*r*))
apply (*unfold tot-ord-def*)
apply (*blast intro*!: *part-ord-converse linear-converse*)
done

lemma *first-is-elem*: *first*(*b*,*B*,*r*) ==> *b*:*B*
by (*unfold first-def*, *blast*)

lemma *well-ord-imp-ex1-first*:
[| *well-ord*(*A*,*r*); *B*<=*A*; *B*~=*0* |] ==> (*EX*! *b*. *first*(*b*,*B*,*r*))
apply (*unfold well-ord-def wf-on-def wf-def first-def*)
apply (*elim conjE allE disjE*, *blast*)
apply (*erule bexE*)
apply (*rule-tac a* = *x* **in** *ex1I*, *auto*)
apply (*unfold tot-ord-def linear-def*, *blast*)
done

lemma *the-first-in*:
[| *well-ord*(*A*,*r*); *B*<=*A*; *B*~=*0* |] ==> (*THE b*. *first*(*b*,*B*,*r*)) : *B*
apply (*drule well-ord-imp-ex1-first*, *assumption*+)
apply (*rule first-is-elem*)
apply (*erule theI*)
done

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*)

apply (*auto simp*: *subset-def preorder-on-def refl-def vimage-def image-def*)
apply *blast*
unfolding *trans-on-def*
apply (*erule-tac P* = (λx. $\forall$ *y*∈*field*(*?r*).
$\forall$ *z*∈*field*(*?r*). ⟨*x*, *y*⟩ ∈ *?r* ⟶ ⟨*y*, *z*⟩ ∈ *?r* ⟶ ⟨*x*, *z*⟩ ∈ *?r*) **in** *rev-ballE*)

apply *best*
apply *blast*
done

lemma *subset-vimage1-vimage1-iff*:
[| *Preorder*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r −‘‘ {*a*} ⊆ *r* −‘‘ {*b*} <−> <*a*, *b*> : *r*
by (*simp add*: *subset-vimage-vimage-iff*)

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*
apply *rule*
apply (*frule equality-iffD*)
apply (*drule equality-iffD*)
apply (*simp add*: *antisym-def refl-def*)
apply *best*
apply (*simp add*: *antisym-def refl-def*)
done

lemma *Partial-order-eq-Image1-Image1-iff*:
[| *Partial-order*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r ‘‘ {*a*} = *r* ‘‘ {*b*} <−> *a* = *b*
by (*simp add*: *partial-order-on-def preorder-on-def*
Refl-antisym-eq-Image1-Image1-iff)

lemma *Refl-antisym-eq-vimage1-vimage1-iff*:
[| *refl*(*field*(*r*), *r*); *antisym*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r −‘‘ {*a*} = *r* −‘‘ {*b*} <−> *a* = *b*
apply *rule*
apply (*frule equality-iffD*)
apply (*drule equality-iffD*)
apply (*simp add*: *antisym-def refl-def*)
apply *best*
apply (*simp add*: *antisym-def refl-def*)
done

lemma *Partial-order-eq-vimage1-vimage1-iff*:
[| *Partial-order*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r −‘‘ {*a*} = *r* −‘‘ {*b*} <−> *a* = *b*
by (*simp add*: *partial-order-on-def preorder-on-def*
Refl-antisym-eq-vimage1-vimage1-iff)

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

lemma *radd-Inl-iff* [*iff*]:
  <Inl(a'), Inl(a)> : radd(A,r,B,s) <–> a':A & a:A & <a',a>:r
by (*unfold radd-def*, *blast*)

lemma *radd-Inr-iff* [*iff*]:
  <Inr(b'), Inr(b)> : radd(A,r,B,s) <–> b':B & b:B & <b',b>:s
by (*unfold radd-def*, *blast*)

lemma *radd-Inr-Inl-iff* [*simp*]:
<*Inr*(*b*), *Inl*(*a*)> : *radd*(*A*,*r*,*B*,*s*) <−> *False*
by (*unfold radd-def*, *blast*)

declare *radd-Inr-Inl-iff* [*THEN iffD1*, *dest*!]

19.1.2 Elimination Rule

lemma *raddE*:
[| <*p'*,*p*> : *radd*(*A*,*r*,*B*,*s*);
!!*x y*. [| *p'*=*Inl*(*x*); *x*:*A*; *p*=*Inr*(*y*); *y*:*B* |] ==> *Q*;
!!*x' x*. [| *p'*=*Inl*(*x'*); *p*=*Inl*(*x*); <*x'*,*x*>: *r*; *x'*:*A*; *x*:*A* |] ==> *Q*;
!!*y' y*. [| *p'*=*Inr*(*y'*); *p*=*Inr*(*y*); <*y'*,*y*>: *s*; *y'*:*B*; *y*:*B* |] ==> *Q*
|] ==> *Q*
by (*unfold radd-def*, *blast*)

19.1.3 Type checking

lemma *radd-type*: *radd*(*A*,*r*,*B*,*s*) <= (*A*+*B*) ∗ (*A*+*B*)
apply (*unfold radd-def*)
apply (*rule Collect-subset*)
done

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

19.1.5 Well-foundedness

lemma *wf-on-radd*: [| *wf*[*A*](*r*); *wf*[*B*](*s*) |] ==> *wf*[*A*+*B*](*radd*(*A*,*r*,*B*,*s*))
apply (*rule wf-onI2*)
apply (*subgoal-tac ALL x*:*A*. *Inl* (*x*) : *Ba*)
— Proving the lemma, which is needed twice!
prefer *2*
apply (*erule-tac V* = *y* : *A* + *B* **in** *thin-rl*)
apply (*rule-tac ballI*)
apply (*erule-tac r* = *r* **and** *a* = *x* **in** *wf-on-induct*, *assumption*)
apply *blast*

Returning to main part of proof

apply *safe*
apply *blast*
apply (*erule-tac r* = *s* **and** *a* = *ya* **in** *wf-on-induct*, *assumption*, *blast*)
done

lemma *wf-radd*: [| *wf*(*r*); *wf*(*s*) |] ==> *wf*(*radd*(*field*(*r*),*r*,*field*(*s*),*s*))
apply (*simp add*: *wf-iff-wf-on-field*)
apply (*rule wf-on-subset-A* [*OF* - *field-radd*])
apply (*blast intro*: *wf-on-radd*)
done

lemma *well-ord-radd*:
[| *well-ord*(*A*,*r*); *well-ord*(*B*,*s*) |] ==> *well-ord*(*A*+*B*, *radd*(*A*,*r*,*B*,*s*))
apply (*rule well-ordI*)
apply (*simp add*: *well-ord-def wf-on-radd*)
apply (*simp add*: *well-ord-def tot-ord-def linear-radd*)
done

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*)
apply (*rule-tac d* = *case* (%*x*. *Inl* (*converse*(*f*)‘*x*), %*y*. *Inr*(*converse*(*g*)‘*y*))
in *lam-bijective*)
apply (*typecheck add*: *bij-is-inj inj-is-fun*)
apply (*auto simp add*: *left-inverse-bij right-inverse-bij*)
done

lemma *sum-ord-iso-cong*:
[| *f*: *ord-iso*(*A*,*r*,*A'*,*r'*); *g*: *ord-iso*(*B*,*s*,*B'*,*s'*) |] ==>
(*lam z*:*A*+*B*. *case*(%*x*. *Inl*(*f*‘*x*), %*y*. *Inr*(*g*‘*y*), *z*))
: *ord-iso*(*A*+*B*, *radd*(*A*,*r*,*B*,*s*), *A'*+*B'*, *radd*(*A'*,*r'*,*B'*,*s'*))
apply (*unfold ord-iso-def*)
apply (*safe intro*!: *sum-bij*)

apply (*auto cong add*: *conj-cong simp add*: *bij-is-fun* [*THEN apply-type*])
done

lemma *sum-disjoint-bij*: *A Int B* = *0* ==>
(*lam z*:*A*+*B*. *case*(%*x*. *x*, %*y*. *y*, *z*)) : *bij*(*A*+*B*, *A Un B*)
apply (*rule-tac d* = %*z*. *if z*:*A then Inl* (*z*) *else Inr* (*z*) **in** *lam-bijective*)
apply *auto*
done

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*))
apply (*rule-tac d* = *case* (%*x*. *Inl* (*Inl* (*x*)), *case* (%*x*. *Inl* (*Inr* (*x*)), *Inr*))
in *lam-bijective*)
apply *auto*
done

```
lemma sum-assoc-ord-iso:
    (lam z:(A+B)+C. case(case(Inl, %y. Inr(Inl(y))), %y. Inr(Inr(y)), z))
     : ord-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)))
by (rule sum-assoc-bij [THEN ord-isoI], auto)
```

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)

by (unfold rmult-def, blast)

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
by blast
```

19.2.2 Type checking

```
lemma rmult-type: rmult(A,r,B,s) <= (A*B) * (A*B)
by (unfold rmult-def, rule Collect-subset)

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))
by (simp add: linear-def, blast)
```

19.2.4 Well-foundedness

```
lemma wf-on-rmult: [| wf[A](r);  wf[B](s) |] ==> wf[A*B](rmult(A,r,B,s))
apply (rule wf-onI2)
apply (erule SigmaE)
apply (erule ssubst)
apply (subgoal-tac ALL b:B. <x,b>: Ba, blast)
apply (erule-tac a = x in wf-on-induct, assumption)
apply (rule ballI)
apply (erule-tac a = b in wf-on-induct, assumption)
apply (best elim!: rmultE bspec [THEN mp])
done
```

lemma *wf-rmult*: [| *wf*(*r*); *wf*(*s*) |] ==> *wf*(*rmult*(*field*(*r*),*r*,*field*(*s*),*s*))
apply (*simp add*: *wf-iff-wf-on-field*)
apply (*rule wf-on-subset-A* [*OF* - *field-rmult*])
apply (*blast intro*: *wf-on-rmult*)
done

lemma *well-ord-rmult*:
[| *well-ord*(*A*,*r*); *well-ord*(*B*,*s*) |] ==> *well-ord*(*A*∗*B*, *rmult*(*A*,*r*,*B*,*s*))
apply (*rule well-ordI*)
apply (*simp add*: *well-ord-def wf-on-rmult*)
apply (*simp add*: *well-ord-def tot-ord-def linear-rmult*)
done

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*)
apply (*rule-tac d* = %<*x*,*y*>. <*converse* (*f*) ‘*x*, *converse* (*g*) ‘*y*>
in *lam-bijective*)
apply (*typecheck add*: *bij-is-inj inj-is-fun*)
apply (*auto simp add*: *left-inverse-bij right-inverse-bij*)
done

lemma *prod-ord-iso-cong*:
[| *f*: *ord-iso*(*A*,*r*,*A*′,*r*′); *g*: *ord-iso*(*B*,*s*,*B*′,*s*′) |]
==> (*lam* <*x*,*y*>:*A*∗*B*. <*f*‘*x*, *g*‘*y*>)
: *ord-iso*(*A*∗*B*, *rmult*(*A*,*r*,*B*,*s*), *A*′∗*B*′, *rmult*(*A*′,*r*′,*B*′,*s*′))
apply (*unfold ord-iso-def*)
apply (*safe intro*!: *prod-bij*)
apply (*simp-all add*: *bij-is-fun* [*THEN apply-type*])
apply (*blast intro*: *bij-is-inj* [*THEN inj-apply-equality*])
done

lemma *singleton-prod-bij*: (*lam z*:*A*. <*x*,*z*>) : *bij*(*A*, {*x*}∗*A*)
by (*rule-tac d* = *snd* **in** *lam-bijective*, *auto*)

lemma *singleton-prod-ord-iso*:
well-ord({*x*},*xr*) ==>
(*lam z*:*A*. <*x*,*z*>) : *ord-iso*(*A*, *r*, {*x*}∗*A*, *rmult*({*x*}, *xr*, *A*, *r*))
apply (*rule singleton-prod-bij* [*THEN ord-isoI*])
apply (*simp* (*no-asm-simp*))
apply (*blast dest*: *well-ord-is-wf* [*THEN wf-on-not-refl*])
done

```
lemma prod-sum-singleton-bij:
    a~:C ==>
      (lam x:C*B + D. case(%x. x, %y.<a,y>, x))
      : bij(C*B + D, C*B Un {a}*D)
apply (rule subst-elem)
apply (rule id-bij [THEN sum-bij, THEN comp-bij])
apply (rule singleton-prod-bij)
apply (rule sum-disjoint-bij, blast)
apply (simp (no-asm-simp) cong add: case-cong)
apply (rule comp-lam [THEN trans, symmetric])
apply (fast elim!: case-type)
apply (simp (no-asm-simp) add: case-case)
done
```

```
lemma prod-sum-singleton-ord-iso:
 [| 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))
apply (rule prod-sum-singleton-bij [THEN ord-isoI])
apply (simp (no-asm-simp) add: pred-iff well-ord-is-wf [THEN wf-on-not-refl])
apply (auto elim!: well-ord-is-wf [THEN wf-on-asym] predE)
done
```

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))
by (rule-tac d = case (%<x,y>.<Inl (x),y>, %<x,y>.<Inr (x),y>)
    in lam-bijective, auto)
```

```
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)))
by (rule sum-prod-distrib-bij [THEN ord-isoI], auto)
```

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))
by (rule-tac d = %<x, <y,z>>. <<x,y>, z> in lam-bijective, auto)
```

```
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)))
by (rule prod-assoc-bij [THEN ord-isoI], auto)
```

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

19.3.2 Type checking

lemma *rvimage-type*: *rvimage*(*A*,*f*,*r*) <= *A*∗*A*
by (*unfold rvimage-def*, *rule Collect-subset*)

lemmas *field-rvimage* = *rvimage-type* [*THEN field-rel-subset*]

lemma *rvimage-converse*: *rvimage*(*A*,*f*, *converse*(*r*)) = *converse*(*rvimage*(*A*,*f*,*r*))
by (*unfold rvimage-def*, *blast*)

19.3.3 Partial Ordering Properties

lemma *irrefl-rvimage*:
  [| *f*: *inj*(*A*,*B*); *irrefl*(*B*,*r*) |] ==> *irrefl*(*A*, *rvimage*(*A*,*f*,*r*))
apply (*unfold irrefl-def rvimage-def*)
apply (*blast intro*: *inj-is-fun* [*THEN apply-type*])
done

lemma *trans-on-rvimage*:
  [| *f*: *inj*(*A*,*B*); *trans*[*B*](*r*) |] ==> *trans*[*A*](*rvimage*(*A*,*f*,*r*))
apply (*unfold trans-on-def rvimage-def*)
apply (*blast intro*: *inj-is-fun* [*THEN apply-type*])
done

lemma *part-ord-rvimage*:
  [| *f*: *inj*(*A*,*B*); *part-ord*(*B*,*r*) |] ==> *part-ord*(*A*, *rvimage*(*A*,*f*,*r*))
apply (*unfold part-ord-def*)
apply (*blast intro*!: *irrefl-rvimage trans-on-rvimage*)
done

19.3.4 Linearity

lemma *linear-rvimage*:
  [| *f*: *inj*(*A*,*B*); *linear*(*B*,*r*) |] ==> *linear*(*A*,*rvimage*(*A*,*f*,*r*))
apply (*simp add*: *inj-def linear-def rvimage-iff*)
apply (*blast intro*: *apply-funtype*)
done

lemma *tot-ord-rvimage*:
  [| *f*: *inj*(*A*,*B*); *tot-ord*(*B*,*r*) |] ==> *tot-ord*(*A*, *rvimage*(*A*,*f*,*r*))
apply (*unfold tot-ord-def*)
apply (*blast intro*!: *part-ord-rvimage linear-rvimage*)
done

19.3.5 Well-foundedness

lemma *wf-rvimage* [*intro*!]: *wf*(*r*) ==> *wf*(*rvimage*(*A*,*f*,*r*))
apply (*simp* (*no-asm-use*) *add*: *rvimage-def wf-eq-minimal*)
apply *clarify*
apply (*subgoal-tac EX w. w* : {*w*: {*f*'*x*. *x*:*Q*}. *EX x. x*: *Q* & (*f*'*x* = *w*) })
 apply (*erule allE*)
 apply (*erule impE*)
 apply *assumption*
 apply *blast*
apply *blast*
done

But note that the combination of *wf-imp-wf-on* and *wf-rvimage* gives *wf*(*r*) $\Longrightarrow$ *wf*[*C*](*rvimage*(*A*, *f*, *r*))

lemma *wf-on-rvimage*: [| *f*: *A*−>*B*; *wf*[*B*](*r*) |] ==> *wf*[*A*](*rvimage*(*A*,*f*,*r*))
apply (*rule wf-onI2*)
apply (*subgoal-tac ALL z*:*A*. *f*'*z*=*f*'*y* −−> *z*: *Ba*)
 apply *blast*
apply (*erule-tac a* = *f*'*y* **in** *wf-on-induct*)
 apply (*blast intro*!: *apply-funtype*)
apply (*blast intro*!: *apply-funtype dest*!: *rvimage-iff* [*THEN iffD1*])
done

lemma *well-ord-rvimage*:
  [| *f*: *inj*(*A*,*B*); *well-ord*(*B*,*r*) |] ==> *well-ord*(*A*, *rvimage*(*A*,*f*,*r*))
apply (*rule well-ordI*)
apply (*unfold well-ord-def tot-ord-def*)
apply (*blast intro*!: *wf-on-rvimage inj-is-fun*)
apply (*blast intro*!: *linear-rvimage*)
done

lemma *ord-iso-rvimage*:
  f: *bij*(*A*,*B*) ==> *f*: *ord-iso*(*A*, *rvimage*(*A*,*f*,*s*), *B*, *s*)
apply (*unfold ord-iso-def*)
apply (*simp add*: *rvimage-iff*)
done

lemma *ord-iso-rvimage-eq*:
  f: *ord-iso*(*A*,*r*, *B*,*s*) ==> *rvimage*(*A*,*f*,*s*) = *r Int A*∗*A*
by (*unfold ord-iso-def rvimage-def*, *blast*)

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*)))
by (*blast intro*: *wf-rvimage wf-Memrel*)

definition
wfrank :: [*i*,*i*]=>*i* **where**
wfrank(*r*,*a*) == *wfrec*(*r*, *a*, %*x f*. ⋃ *y* ∈ *r*−‘‘{*x*}. *succ*(*f*‘*y*))

definition
wftype :: *i*=>*i* **where**
wftype(*r*) == ⋃ *y* ∈ *range*(*r*). *succ*(*wfrank*(*r*,*y*))

lemma *wfrank*: *wf*(*r*) ==> *wfrank*(*r*,*a*) = (⋃ *y* ∈ *r*−‘‘{*a*}. *succ*(*wfrank*(*r*,*y*)))
by (*subst wfrank-def* [*THEN def-wfrec*], *simp-all*)

lemma *Ord-wfrank*: *wf*(*r*) ==> *Ord*(*wfrank*(*r*,*a*))
apply (*rule-tac a*=*a* **in** *wf-induct*, *assumption*)
apply (*subst wfrank*, *assumption*)
apply (*rule Ord-succ* [*THEN Ord-UN*], *blast*)
done

lemma *wfrank-lt*: [|*wf*(*r*); <*a*,*b*> ∈ *r*|] ==> *wfrank*(*r*,*a*) < *wfrank*(*r*,*b*)
apply (*rule-tac a1* = *b* **in** *wfrank* [*THEN ssubst*], *assumption*)
apply (*rule UN-I* [*THEN ltI*])
apply (*simp add*: *Ord-wfrank vimage-iff*)+
done

lemma *Ord-wftype*: *wf*(*r*) ==> *Ord*(*wftype*(*r*))
by (*simp add*: *wftype-def Ord-wfrank*)

lemma *wftypeI*: ⟦*wf*(*r*); *x* ∈ *field*(*r*)⟧ ⟹ *wfrank*(*r*,*x*) ∈ *wftype*(*r*)
apply (*simp add*: *wftype-def*)
apply (*blast intro*: *wfrank-lt* [*THEN ltD*])
done

lemma *wf-imp-subset-rvimage*:
[|*wf*(*r*); *r* ⊆ *A*∗*A*|] ==> ∃ *i f*. *Ord*(*i*) & *r* <= *rvimage*(*A*, *f*, *Memrel*(*i*))
apply (*rule-tac x*=*wftype*(*r*) **in** *exI*)
apply (*rule-tac x*=λ*x*∈*A*. *wfrank*(*r*,*x*) **in** *exI*)
apply (*simp add*: *Ord-wftype*, *clarify*)
apply (*frule subsetD*, *assumption*, *clarify*)
apply (*simp add*: *rvimage-iff wfrank-lt* [*THEN ltD*])
apply (*blast intro*: *wftypeI*)
done

theorem *wf-iff-subset-rvimage*:
relation(*r*) ==> *wf*(*r*) <−> (∃ *i f A*. *Ord*(*i*) & *r* <= *rvimage*(*A*, *f*, *Memrel*(*i*)))
by (*blast dest*!: *relation-field-times-field wf-imp-subset-rvimage*
intro: *wf-rvimage-Ord* [*THEN wf-subset*])

19.5 Other Results

lemma *wf-times*: *A Int B = 0 ==> wf(A*B)*
by (*simp add*: *wf-def*, *blast*)

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)*
apply (*simp add*: *wf-def*, *clarify*)
apply (*rule equalityI*)
 prefer *2* **apply** *blast*
apply *clarify*
apply (*drule-tac x=Z* **in** *spec*)
apply (*drule-tac x=Z Int domain(s)* **in** *spec*)
apply *simp*
apply (*blast intro*: *elim*: *equalityE*)
done

19.5.1 The Empty Relation

lemma *wf0*: *wf(0)*
by (*simp add*: *wf-def*, *blast*)

lemma *linear0*: *linear(0,0)*
by (*simp add*: *linear-def*)

lemma *well-ord0*: *well-ord(0,0)*
by (*blast intro*: *wf-imp-wf-on well-ordI wf0 linear0*)

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)))
apply (*simp* (*no-asm*) *add*: *measure-def rvimage-def Memrel-iff*)
apply (*rule equalityI*, *auto*)
apply (*auto intro*: *Ord-in-Ord simp add*: *lt-def*)
done

lemma *wf-measure* [*iff*]: *wf(measure(A,f))*
by (*simp* (*no-asm*) *add*: *measure-eq-rvimage-Memrel wf-Memrel wf-rvimage*)

lemma *measure-iff* [*iff*]: <*x*,*y*> : *measure(A,f)* <-> *x:A* & *y:A* & *f(x)*<*f(y)*
by (*simp* (*no-asm*) *add*: *measure-def*)

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))*
apply (*auto simp add*: *linear-def*)
apply (*rule-tac i=f(x)* **and** *j=f(y)* **in** *Ord-linear-lt*)

apply (*simp-all add*: *Ordf*)
apply (*blast intro*: *inj*)
done

lemma *wf-on-measure*: *wf*[*B*](*measure*(*A*,*f*))
by (*rule wf-imp-wf-on* [*OF wf-measure*])

lemma *well-ord-measure*:
assumes *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*))
apply (*rule well-ordI*)
apply (*rule wf-on-measure*)
apply (*blast intro*: *linear-measure Ordf inj*)
done

lemma *measure-type*: *measure*(*A*,*f*) <= *A*∗*A*
by (*auto simp add*: *measure-def*)

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*[⋃ *a*∈*A*. *B*(*a*)](*s*)
apply (*rule wf-onI2*)
apply (*erule UN-E*)
apply (*subgoal-tac* ∀ *z* ∈ *B*(*a*). *z* ∈ *Ba*, *blast*)
apply (*rule-tac a* = *a* **in** *wf-on-induct* [*OF wfA*], *assumption*)
apply (*rule ballI*)
apply (*rule-tac a* = *z* **in** *wf-on-induct* [*OF wfB*], *assumption*, *assumption*)
apply (*rename-tac u*)
apply (*drule-tac x*=*u* **in** *bspec*, *blast*)
apply (*erule mp*, *clarify*)
apply (*frule ok*, *assumption*+, *blast*)
done

19.5.4 Bijections involving Powersets

lemma *Pow-sum-bij*:
(λ*Z* ∈ *Pow*(*A*+*B*). <{*x* ∈ *A*. *Inl*(*x*) ∈ *Z*}, {*y* ∈ *B*. *Inr*(*y*) ∈ *Z*}>)
∈ *bij*(*Pow*(*A*+*B*), *Pow*(*A*)∗*Pow*(*B*))
apply (*rule-tac d* = %<*X*,*Y*>. {*Inl* (*x*). *x* ∈ *X*} *Un* {*Inr* (*y*). *y* ∈ *Y*}
in *lam-bijective*)
apply *force*+
done

As a special case, we have $bij(Pow(A \times B),\ A \to Pow(B))$

lemma *Pow-Sigma-bij*:
  $(\lambda r \in Pow(Sigma(A,B)).\ \lambda x \in A.\ r``\{x\})$
  $\in bij(Pow(Sigma(A,B)), \Pi\ x \in A.\ Pow(B(x)))$
apply (*rule-tac d* = %*f*. $\bigcup x \in A. \bigcup y \in f`x.\ \{$<*x*,*y*>$\}$ **in** *lam-bijective*)
apply (*blast intro*: *lam-type*)
apply (*blast dest*: *apply-type*, *simp-all*)
apply *fast*
apply (*rule fun-extension*, *auto*)
by *blast*

end

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-alt(*X*) == *well-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*))
apply (*rule well-ordI*)
apply (*rule wf-Memrel* [*THEN wf-imp-wf-on*])
apply (*simp add*: *ltD lt-Ord linear-def*
ltI [*THEN lt-trans2* [*of - j i*]])
apply (*intro ballI Ord-linear*)
apply (*blast intro*: *Ord-in-Ord lt-Ord*)+
done

lemmas *well-ord-Memrel* = *le-refl* [*THEN le-well-ord-Memrel*]

lemma *lt-pred-Memrel*:
j<*i* ==> *pred*(*i*, *j*, *Memrel*(*i*)) = *j*
apply (*unfold pred-def lt-def*)
apply (*simp* (*no-asm-simp*))
apply (*blast intro*: *Ord-trans*)
done

lemma *pred-Memrel*:
x:*A* ==> *pred*(*A*, *x*, *Memrel*(*A*)) = *A Int x*
by (*unfold pred-def Memrel-def*, *blast*)

lemma *Ord-iso-implies-eq-lemma*:
[| *j*<*i*; *f*: *ord-iso*(*i*,*Memrel*(*i*),*j*,*Memrel*(*j*)) |] ==> *R*
apply (*frule lt-pred-Memrel*)
apply (*erule ltE*)

apply (*rule well-ord-Memrel* [*THEN well-ord-iso-predE*, *of i f j*], *auto*)
apply (*unfold ord-iso-def*)

apply (*simp* (*no-asm-simp*))
apply (*blast intro*: *bij-is-fun* [*THEN apply-type*] *Ord-trans*)
done

lemma *Ord-iso-implies-eq*:
  [| *Ord*(*i*); *Ord*(*j*); *f*: *ord-iso*(*i*,*Memrel*(*i*),*j*,*Memrel*(*j*)) |]
  ==> *i*=*j*
apply (*rule-tac i* = *i* **and** *j* = *j* **in** *Ord-linear-lt*)
apply (*blast intro*: *ord-iso-sym Ord-iso-implies-eq-lemma*)+
done

20.2 Ordermap and ordertype

lemma *ordermap-type*:
  ordermap(*A*,*r*) : *A* −> *ordertype*(*A*,*r*)
apply (*unfold ordermap-def ordertype-def*)
apply (*rule lam-type*)
apply (*rule lamI* [*THEN imageI*], *assumption*+)
done

20.2.1 Unfolding of ordermap

lemma *ordermap-eq-image*:
  [| *wf*[*A*](*r*); *x*:*A* |]
  ==> *ordermap*(*A*,*r*) ‘ *x* = *ordermap*(*A*,*r*) ‘‘ *pred*(*A*,*x*,*r*)
apply (*unfold ordermap-def pred-def*)
apply (*simp* (*no-asm-simp*))
apply (*erule wfrec-on* [*THEN trans*], *assumption*)
apply (*simp* (*no-asm-simp*) *add*: *subset-iff image-lam vimage-singleton-iff*)
done

lemma *ordermap-pred-unfold*:
  [| *wf*[*A*](*r*); *x*:*A* |]
  ==> *ordermap*(*A*,*r*) ‘ *x* = {*ordermap*(*A*,*r*)‘*y* . *y* : *pred*(*A*,*x*,*r*)}
by (*simp add*: *ordermap-eq-image pred-subset ordermap-type* [*THEN image-fun*])

lemmas *ordermap-unfold* = *ordermap-pred-unfold* [*simplified pred-def*]

20.2.2 Showing that ordermap, ordertype yield ordinals

lemma *Ord-ordermap*:
  [| *well-ord*(*A*,*r*); *x*:*A* |] ==> *Ord*(*ordermap*(*A*,*r*) ‘ *x*)
apply (*unfold well-ord-def tot-ord-def part-ord-def*, *safe*)
apply (*rule-tac a*=*x* **in** *wf-on-induct*, *assumption*+)

apply (*simp* (*no-asm-simp*) *add*: *ordermap-pred-unfold*)
apply (*rule OrdI* [*OF - Ord-is-Transset*])
apply (*unfold pred-def Transset-def*)
apply (*blast intro*: *trans-onD*
dest!: *ordermap-unfold* [*THEN equalityD1*])+
done

lemma *Ord-ordertype*:
well-ord(*A*,*r*) ==> *Ord*(*ordertype*(*A*,*r*))
apply (*unfold ordertype-def*)
apply (*subst image-fun* [*OF ordermap-type subset-refl*])
apply (*rule OrdI* [*OF - Ord-is-Transset*])
prefer *2* **apply** (*blast intro*: *Ord-ordermap*)
apply (*unfold Transset-def well-ord-def*)
apply (*blast intro*: *trans-onD*
dest!: *ordermap-unfold* [*THEN equalityD1*])
done

20.2.3 ordermap preserves the orderings in both directions

lemma *ordermap-mono*:
[| <*w*,*x*>: *r*; *wf*[*A*](*r*); *w*: *A*; *x*: *A* |]
==> *ordermap*(*A*,*r*)'*w* : *ordermap*(*A*,*r*)'*x*
apply (*erule-tac x1* = *x* **in** *ordermap-unfold* [*THEN ssubst*], *assumption*, *blast*)
done

lemma *converse-ordermap-mono*:
[| *ordermap*(*A*,*r*)'*w* : *ordermap*(*A*,*r*)'*x*; *well-ord*(*A*,*r*); *w*: *A*; *x*: *A* |]
==> <*w*,*x*>: *r*
apply (*unfold well-ord-def tot-ord-def*, *safe*)
apply (*erule-tac x*=*w* **and** *y*=*x* **in** *linearE*, *assumption*+)
apply (*blast elim*!: *mem-not-refl* [*THEN notE*])
apply (*blast dest*: *ordermap-mono intro*: *mem-asym*)
done

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*))
apply (*unfold well-ord-def tot-ord-def bij-def inj-def*)
apply (*force intro*!: *ordermap-type ordermap-surj*
elim: *linearE dest*: *ordermap-mono*
simp add: *mem-not-refl*)
done

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*)))
apply (*unfold ord-iso-def*)
apply (*safe elim*!: *well-ord-is-wf*
intro!: *ordermap-type* [*THEN apply-type*] *ordermap-mono ordermap-bij*)
apply (*blast dest*!: *converse-ordermap-mono*)
done

lemma *ordertype-eq*:
[| *f*: *ord-iso*(*A*,*r*,*B*,*s*); *well-ord*(*B*,*s*) |]
==> *ordertype*(*A*,*r*) = *ordertype*(*B*,*s*)
apply (*frule well-ord-ord-iso*, *assumption*)
apply (*rule Ord-iso-implies-eq*, (*erule Ord-ordertype*)+)
apply (*blast intro*: *ord-iso-trans ord-iso-sym ordertype-ord-iso*)
done

lemma *ordertype-eq-imp-ord-iso*:
[| *ordertype*(*A*,*r*) = *ordertype*(*B*,*s*); *well-ord*(*A*,*r*); *well-ord*(*B*,*s*) |]
==> *EX f*. *f*: *ord-iso*(*A*,*r*,*B*,*s*)
apply (*rule exI*)
apply (*rule ordertype-ord-iso* [*THEN ord-iso-trans*], *assumption*)
apply (*erule ssubst*)
apply (*erule ordertype-ord-iso* [*THEN ord-iso-sym*])
done

20.2.5 Basic equalities for ordertype

lemma *le-ordertype-Memrel*: *j le i* ==> *ordertype*(*j*,*Memrel*(*i*)) = *j*
apply (*rule Ord-iso-implies-eq* [*symmetric*])
apply (*erule ltE*, *assumption*)
apply (*blast intro*: *le-well-ord-Memrel Ord-ordertype*)
apply (*rule ord-iso-trans*)
apply (*erule-tac* [*2*] *le-well-ord-Memrel* [*THEN ordertype-ord-iso*])
apply (*rule id-bij* [*THEN ord-isoI*])
apply (*simp* (*no-asm-simp*))
apply (*fast elim*: *ltE Ord-in-Ord Ord-trans*)
done

lemmas *ordertype-Memrel* = *le-refl* [*THEN le-ordertype-Memrel*]

lemma *ordertype-0* [*simp*]: *ordertype*(*0*,*r*) = *0*
apply (*rule id-bij* [*THEN ord-isoI*, *THEN ordertype-eq*, *THEN trans*])
apply (*erule emptyE*)
apply (*rule well-ord-0*)
apply (*rule Ord-0* [*THEN ordertype-Memrel*])
done

lemmas *bij-ordertype-vimage* = *ord-iso-rvimage* [*THEN ordertype-eq*]

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*
apply (*frule wf-on-subset-A* [*OF well-ord-is-wf pred-subset*])
apply (*rule-tac a*=*z* **in** *wf-on-induct*, *assumption*+)
apply (*safe elim*!: *predE*)
apply (*simp* (*no-asm-simp*) *add*: *ordermap-pred-unfold well-ord-is-wf pred-iff*)

apply (*simp* (*no-asm-simp*) *add*: *pred-pred-eq*)
apply (*simp add*: *pred-def*)
apply (*rule RepFun-cong* [*OF - refl*])
apply (*drule well-ord-is-trans-on*)
apply (*fast elim*!: *trans-onD*)
done

lemma *ordertype-unfold*:
  ordertype(*A*,*r*) = {*ordermap*(*A*,*r*)‘*y* . *y* : *A*}
apply (*unfold ordertype-def*)
apply (*rule image-fun* [*OF ordermap-type subset-refl*])
done

Theorems by Krzysztof Grabczewski; proofs simplified by lcp

lemma *ordertype-pred-subset*: [| *well-ord*(*A*,*r*); *x*:*A* |] ==>
    ordertype(*pred*(*A*,*x*,*r*),*r*) <= *ordertype*(*A*,*r*)
apply (*simp add*: *ordertype-unfold well-ord-subset* [*OF - pred-subset*])
apply (*fast intro*: *ordermap-pred-eq-ordermap elim*: *predE*)
done

lemma *ordertype-pred-lt*:
  [| *well-ord*(*A*,*r*); *x*:*A* |]
  ==> *ordertype*(*pred*(*A*,*x*,*r*),*r*) < *ordertype*(*A*,*r*)
apply (*rule ordertype-pred-subset* [*THEN subset-imp-le*, *THEN leE*])
apply (*simp-all add*: *Ord-ordertype well-ord-subset* [*OF - pred-subset*])
apply (*erule sym* [*THEN ordertype-eq-imp-ord-iso*, *THEN exE*])
apply (*erule-tac* [*3*] *well-ord-iso-predE*)
apply (*simp-all add*: *well-ord-subset* [*OF - pred-subset*])
done

lemma *ordertype-pred-unfold*:
  well-ord(*A*,*r*)
  ==> *ordertype*(*A*,*r*) = {*ordertype*(*pred*(*A*,*x*,*r*),*r*). *x*:*A*}
apply (*rule equalityI*)
apply (*safe intro*!: *ordertype-pred-lt* [*THEN ltD*])
apply (*auto simp add*: *ordertype-def well-ord-is-wf* [*THEN ordermap-eq-image*]

ordermap-type [*THEN image-fun*]
ordermap-pred-eq-ordermap pred-subset)
done

20.3 Alternative definition of ordinal

lemma *Ord-is-Ord-alt*: *Ord*(*i*) ==> *Ord-alt*(*i*)
apply (*unfold Ord-alt-def*)
apply (*rule conjI*)
apply (*erule well-ord-Memrel*)
apply (*unfold Ord-def Transset-def pred-def Memrel-def*, *blast*)
done

lemma *Ord-alt-is-Ord*:
Ord-alt(*i*) ==> *Ord*(*i*)
apply (*unfold Ord-alt-def Ord-def Transset-def well-ord-def*
tot-ord-def part-ord-def trans-on-def)
apply (*simp add*: *pred-Memrel*)
apply (*blast elim*!: *equalityE*)
done

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*)
apply (*rule-tac d* = *Inl* **in** *lam-bijective*, *safe*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *ordertype-sum-0-eq*:
well-ord(*A*,*r*) ==> *ordertype*(*A*+*0*, *radd*(*A*,*r*,*0*,*s*)) = *ordertype*(*A*,*r*)
apply (*rule bij-sum-0* [*THEN ord-isoI*, *THEN ordertype-eq*])
prefer *2* **apply** *assumption*
apply *force*
done

lemma *bij-0-sum*: (*lam z*:*0*+*A*. *case*(%*x*. *x*, %*y*. *y*, *z*)) : *bij*(*0*+*A*, *A*)
apply (*rule-tac d* = *Inr* **in** *lam-bijective*, *safe*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *ordertype-0-sum-eq*:
well-ord(*A*,*r*) ==> *ordertype*(*0*+*A*, *radd*(*0*,*s*,*A*,*r*)) = *ordertype*(*A*,*r*)
apply (*rule bij-0-sum* [*THEN ord-isoI*, *THEN ordertype-eq*])
prefer *2* **apply** *assumption*
apply *force*

done

Initial segments of radd. Statements by Grabczewski

lemma *pred-Inl-bij*:
a:*A* ==> (*lam x*:*pred*(*A*,*a*,*r*). *Inl*(*x*))
: *bij*(*pred*(*A*,*a*,*r*), *pred*(*A*+*B*, *Inl*(*a*), *radd*(*A*,*r*,*B*,*s*)))
apply (*unfold pred-def*)
apply (*rule-tac d* = *case* (%*x*. *x*, %*y*. *y*) **in** *lam-bijective*)
apply *auto*
done

lemma *ordertype-pred-Inl-eq*:
[| *a*:*A*; *well-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*)
apply (*rule pred-Inl-bij* [*THEN ord-isoI*, *THEN ord-iso-sym*, *THEN ordertype-eq*])
apply (*simp-all add*: *well-ord-subset* [*OF* - *pred-subset*])
apply (*simp add*: *pred-def*)
done

lemma *pred-Inr-bij*:
b:*B* ==>
id(*A*+*pred*(*B*,*b*,*s*))
: *bij*(*A*+*pred*(*B*,*b*,*s*), *pred*(*A*+*B*, *Inr*(*b*), *radd*(*A*,*r*,*B*,*s*)))
apply (*unfold pred-def id-def*)
apply (*rule-tac d* = %*z*. *z* **in** *lam-bijective*, *auto*)
done

lemma *ordertype-pred-Inr-eq*:
[| *b*:*B*; *well-ord*(*A*,*r*); *well-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*))
apply (*rule pred-Inr-bij* [*THEN ord-isoI*, *THEN ord-iso-sym*, *THEN ordertype-eq*])
prefer *2* **apply** (*force simp add*: *pred-def id-def*, *assumption*)
apply (*blast intro*: *well-ord-radd well-ord-subset* [*OF* - *pred-subset*])
done

20.4.2 ordify: trivial coercion to an ordinal

lemma *Ord-ordify* [*iff*, *TC*]: *Ord*(*ordify*(*x*))
by (*simp add*: *ordify-def*)

lemma *ordify-idem* [*simp*]: *ordify*(*ordify*(*x*)) = *ordify*(*x*)
by (*simp add*: *ordify-def*)

20.4.3 Basic laws for ordinal addition

lemma *Ord-raw-oadd*: [|*Ord*(*i*); *Ord*(*j*)|] ==> *Ord*(*raw-oadd*(*i*,*j*))
by (*simp add*: *raw-oadd-def ordify-def Ord-ordertype well-ord-radd*

well-ord-Memrel)

lemma *Ord-oadd* [*iff*, *TC*]: *Ord*(*i*++*j*)
by (*simp add*: *oadd-def Ord-raw-oadd*)

Ordinal addition with zero

lemma *raw-oadd-0*: *Ord*(*i*) ==> *raw-oadd*(*i*,*0*) = *i*
by (*simp add*: *raw-oadd-def ordify-def ordertype-sum-0-eq*
ordertype-Memrel well-ord-Memrel)

lemma *oadd-0* [*simp*]: *Ord*(*i*) ==> *i*++*0* = *i*
apply (*simp* (*no-asm-simp*) *add*: *oadd-def raw-oadd-0 ordify-def*)
done

lemma *raw-oadd-0-left*: *Ord*(*i*) ==> *raw-oadd*(*0*,*i*) = *i*
by (*simp add*: *raw-oadd-def ordify-def ordertype-0-sum-eq ordertype-Memrel*
well-ord-Memrel)

lemma *oadd-0-left* [*simp*]: *Ord*(*i*) ==> *0*++*i* = *i*
by (*simp add*: *oadd-def raw-oadd-0-left ordify-def*)

lemma *oadd-eq-if-raw-oadd*:
i++*j* = (*if Ord*(*i*) *then* (*if Ord*(*j*) *then raw-oadd*(*i*,*j*) *else i*)
else (*if Ord*(*j*) *then j else 0*))
by (*simp add*: *oadd-def ordify-def raw-oadd-0-left raw-oadd-0*)

lemma *raw-oadd-eq-oadd*: [|*Ord*(*i*); *Ord*(*j*)|] ==> *raw-oadd*(*i*,*j*) = *i*++*j*
by (*simp add*: *oadd-def ordify-def*)

lemma *lt-oadd1*: *k*<*i* ==> *k* < *i*++*j*
apply (*simp add*: *oadd-def ordify-def lt-Ord2 raw-oadd-0*, *clarify*)
apply (*simp add*: *raw-oadd-def*)
apply (*rule ltE*, *assumption*)
apply (*rule ltI*)
apply (*force simp add*: *ordertype-pred-unfold well-ord-radd well-ord-Memrel*
ordertype-pred-Inl-eq lt-pred-Memrel leI [*THEN le-ordertype-Memrel*])
apply (*blast intro*: *Ord-ordertype well-ord-radd well-ord-Memrel*)
done

lemma *oadd-le-self*: *Ord*(*i*) ==> *i le i*++*j*
apply (*rule all-lt-imp-le*)
apply (*auto simp add*: *Ord-oadd lt-oadd1*)
done

Various other results

lemma *id-ord-iso-Memrel*: A<=B ==> id(A) : ord-iso(A, Memrel(A), A, Memrel(B))
apply (rule id-bij [THEN ord-isoI])
apply (simp (no-asm-simp))
apply blast
done

lemma *subset-ord-iso-Memrel*:
[| f: ord-iso(A,Memrel(B),C,r); A<=B |] ==> f: ord-iso(A,Memrel(A),C,r)
apply (frule ord-iso-is-bij [THEN bij-is-fun, THEN fun-is-rel])
apply (frule ord-iso-trans [OF id-ord-iso-Memrel], assumption)
apply (simp add: right-comp-id)
done

lemma *restrict-ord-iso*:
[| 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)
apply (frule ltD)
apply (frule ord-iso-is-bij [THEN bij-is-fun, THEN apply-type], assumption)
apply (frule ord-iso-restrict-pred, assumption)
apply (simp add: pred-iff trans-pred-pred-eq lt-pred-Memrel)
apply (blast intro!: subset-ord-iso-Memrel le-imp-subset [OF leI])
done

lemma *restrict-ord-iso2*:
[| 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))
by (blast intro: restrict-ord-iso ord-iso-sym ltI)

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)))
apply (erule ltE)
apply (rule ord-iso-refl [THEN sum-ord-iso-cong, THEN ordertype-eq])
apply (erule OrdmemD [THEN id-ord-iso-Memrel, THEN ord-iso-sym])
apply (simp-all add: well-ord-radd well-ord-Memrel)
done

lemma *oadd-lt-mono2*: k<j ==> i++k < i++j
apply (simp add: oadd-def ordify-def raw-oadd-0-left lt-Ord lt-Ord2, clarify)
apply (simp add: raw-oadd-def)
apply (rule ltE, assumption)
apply (rule ordertype-pred-unfold [THEN equalityD2, THEN subsetD, THEN ltI])
apply (simp-all add: Ord-ordertype well-ord-radd well-ord-Memrel)
apply (rule bexI)

apply (*erule-tac* [*2*] *InrI*)
apply (*simp add*: *ordertype-pred-Inr-eq well-ord-Memrel lt-pred-Memrel*
leI [*THEN le-ordertype-Memrel*] *ordertype-sum-Memrel*)
done

lemma *oadd-lt-cancel2*: [| *i*++*j* < *i*++*k*; *Ord*(*j*) |] ==> *j*<*k*
apply (*simp* (*asm-lr*) *add*: *oadd-eq-if-raw-oadd split add*: *split-if-asm*)
prefer *2*
apply (*frule-tac i* = *i* **and** *j* = *j* **in** *oadd-le-self*)
apply (*simp* (*asm-lr*) *add*: *oadd-def ordify-def lt-Ord not-lt-iff-le* [*THEN iff-sym*])
apply (*rule Ord-linear-lt*, *auto*)
apply (*simp-all add*: *raw-oadd-eq-oadd*)
apply (*blast dest*: *oadd-lt-mono2 elim*: *lt-irrefl lt-asym*)+
done

lemma *oadd-lt-iff2*: *Ord*(*j*) ==> *i*++*j* < *i*++*k* <-> *j*<*k*
by (*blast intro*!: *oadd-lt-mono2 dest*!: *oadd-lt-cancel2*)

lemma *oadd-inject*: [| *i*++*j* = *i*++*k*; *Ord*(*j*); *Ord*(*k*) |] ==> *j*=*k*
apply (*simp add*: *oadd-eq-if-raw-oadd split add*: *split-if-asm*)
apply (*simp add*: *raw-oadd-eq-oadd*)
apply (*rule Ord-linear-lt*, *auto*)
apply (*force dest*: *oadd-lt-mono2* [*of concl*: *i*] *simp add*: *lt-not-refl*)+
done

lemma *lt-oadd-disj*: *k* < *i*++*j* ==> *k*<*i* | (*EX l*:*j*. *k* = *i*++*l*)
apply (*simp add*: *Ord-in-Ord'* [*of - j*] *oadd-eq-if-raw-oadd*
split add: *split-if-asm*)
prefer *2*
apply (*simp add*: *Ord-in-Ord'* [*of - j*] *lt-def*)
apply (*simp add*: *ordertype-pred-unfold well-ord-radd well-ord-Memrel raw-oadd-def*)
apply (*erule ltD* [*THEN RepFunE*])
apply (*force simp add*: *ordertype-pred-Inl-eq well-ord-Memrel ltI*
lt-pred-Memrel le-ordertype-Memrel leI
ordertype-pred-Inr-eq ordertype-sum-Memrel)
done

20.4.4 Ordinal addition with successor – via associativity!

lemma *oadd-assoc*: (*i*++*j*)++*k* = *i*++(*j*++*k*)
apply (*simp add*: *oadd-eq-if-raw-oadd Ord-raw-oadd raw-oadd-0 raw-oadd-0-left*,
clarify)
apply (*simp add*: *raw-oadd-def*)
apply (*rule ordertype-eq* [*THEN trans*])
apply (*rule sum-ord-iso-cong* [*OF ordertype-ord-iso* [*THEN ord-iso-sym*]
ord-iso-refl])
apply (*simp-all add*: *Ord-ordertype well-ord-radd well-ord-Memrel*)
apply (*rule sum-assoc-ord-iso* [*THEN ordertype-eq*, *THEN trans*])
apply (*rule-tac* [*2*] *ordertype-eq*)

apply (*rule-tac* [*2*] *sum-ord-iso-cong* [*OF ord-iso-refl ordertype-ord-iso*])
apply (*blast intro*: *Ord-ordertype well-ord-radd well-ord-Memrel*)+
done

lemma *oadd-unfold*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *i*++*j* = *i* *Un* (⋃*k*∈*j*. {*i*++*k*})
apply (*rule subsetI* [*THEN equalityI*])
apply (*erule ltI* [*THEN lt-oadd-disj*, *THEN disjE*])
apply (*blast intro*: *Ord-oadd*)
apply (*blast elim*!: *ltE*, *blast*)
apply (*force intro*: *lt-oadd1 oadd-lt-mono2 simp add*: *Ord-mem-iff-lt*)
done

lemma *oadd-1*: *Ord*(*i*) ==> *i*++*1* = *succ*(*i*)
apply (*simp* (*no-asm-simp*) *add*: *oadd-unfold Ord-1 oadd-0*)
apply *blast*
done

lemma *oadd-succ* [*simp*]: *Ord*(*j*) ==> *i*++*succ*(*j*) = *succ*(*i*++*j*)
apply (*simp add*: *oadd-eq-if-raw-oadd*, *clarify*)
apply (*simp add*: *raw-oadd-eq-oadd*)
apply (*simp add*: *oadd-1* [*of j*, *symmetric*] *oadd-1* [*of i*++*j*, *symmetric*]
oadd-assoc)
done

Ordinal addition with limit ordinals

lemma *oadd-UN*:
[| !!*x*. *x*:*A* ==> *Ord*(*j*(*x*)); *a*:*A* |]
==> *i* ++ (⋃*x*∈*A*. *j*(*x*)) = (⋃*x*∈*A*. *i*++*j*(*x*))
by (*blast intro*: *ltI Ord-UN Ord-oadd lt-oadd1* [*THEN ltD*]
oadd-lt-mono2 [*THEN ltD*]
elim!: *ltE dest*!: *ltI* [*THEN lt-oadd-disj*])

lemma *oadd-Limit*: *Limit*(*j*) ==> *i*++*j* = (⋃*k*∈*j*. *i*++*k*)
apply (*frule Limit-has-0* [*THEN ltD*])
apply (*simp add*: *Limit-is-Ord* [*THEN Ord-in-Ord*] *oadd-UN* [*symmetric*]
Union-eq-UN [*symmetric*] *Limit-Union-eq*)
done

lemma *oadd-eq-0-iff*: [| *Ord*(*i*); *Ord*(*j*) |] ==> (*i* ++ *j*) = *0* <-> *i*=*0* & *j*=*0*
apply (*erule trans-induct3* [*of j*])
apply (*simp-all add*: *oadd-Limit*)
apply (*simp add*: *Union-empty-iff Limit-def lt-def*, *blast*)
done

lemma *oadd-eq-lt-iff*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *0* < (*i* ++ *j*) <-> *0*<*i* | *0*<*j*
by (*simp add*: *Ord-0-lt-iff* [*symmetric*] *oadd-eq-0-iff*)

lemma *oadd-LimitI*: [| *Ord*(*i*); *Limit*(*j*) |] ==> *Limit*(*i* ++ *j*)
apply (*simp add*: *oadd-Limit*)

apply (*frule Limit-has-1* [*THEN ltD*])
apply (*rule increasing-LimitI*)
apply (*rule Ord-0-lt*)
apply (*blast intro*: *Ord-in-Ord* [*OF Limit-is-Ord*])
apply (*force simp add*: *Union-empty-iff oadd-eq-0-iff*
Limit-is-Ord [*of j*, *THEN Ord-in-Ord*], *auto*)
apply (*rule-tac x=succ*(*y*) **in** *bexI*)
apply (*simp add*: *ltI Limit-is-Ord* [*of j*, *THEN Ord-in-Ord*])
apply (*simp add*: *Limit-def lt-def*)
done

Order/monotonicity properties of ordinal addition

lemma *oadd-le-self2*: *Ord*(*i*) *==> i le j++i*
apply (*erule-tac i = i* **in** *trans-induct3*)
apply (*simp* (*no-asm-simp*) *add*: *Ord-0-le*)
apply (*simp* (*no-asm-simp*) *add*: *oadd-succ succ-leI*)
apply (*simp* (*no-asm-simp*) *add*: *oadd-Limit*)
apply (*rule le-trans*)
apply (*rule-tac* [*2*] *le-implies-UN-le-UN*)
apply (*erule-tac* [*2*] *bspec*)
prefer *2* **apply** *assumption*
apply (*simp add*: *Union-eq-UN* [*symmetric*] *Limit-Union-eq le-refl Limit-is-Ord*)
done

lemma *oadd-le-mono1*: *k le j ==> k++i le j++i*
apply (*frule lt-Ord*)
apply (*frule le-Ord2*)
apply (*simp add*: *oadd-eq-if-raw-oadd*, *clarify*)
apply (*simp add*: *raw-oadd-eq-oadd*)
apply (*erule-tac i = i* **in** *trans-induct3*)
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*) *add*: *oadd-succ succ-le-iff*)
apply (*simp* (*no-asm-simp*) *add*: *oadd-Limit*)
apply (*rule le-implies-UN-le-UN*, *blast*)
done

lemma *oadd-lt-mono*: [| *i' le i*; *j'<j* |] *==> i'++j' < i++j*
by (*blast intro*: *lt-trans1 oadd-le-mono1 oadd-lt-mono2 Ord-succD elim*: *ltE*)

lemma *oadd-le-mono*: [| *i' le i*; *j' le j* |] *==> i'++j' le i++j*
by (*simp del*: *oadd-succ add*: *oadd-succ* [*symmetric*] *le-Ord2 oadd-lt-mono*)

lemma *oadd-le-iff2*: [| *Ord*(*j*); *Ord*(*k*) |] *==> i++j le i++k <-> j le k*
by (*simp del*: *oadd-succ add*: *oadd-lt-iff2 oadd-succ* [*symmetric*] *Ord-succ*)

lemma *oadd-lt-self*: [| *Ord*(*i*); *0<j* |] *==> i < i++j*
apply (*rule lt-trans2*)
apply (*erule le-refl*)
apply (*simp only*: *lt-Ord2 oadd-1* [*of i*, *symmetric*])

apply (*blast intro*: *succ-leI oadd-le-mono*)
done

Every ordinal is exceeded by some limit ordinal.

lemma *Ord-imp-greater-Limit*: *Ord*(*i*) ==> ∃ *k*. *i*<*k* & *Limit*(*k*)
apply (*rule-tac x*=*i* ++ *nat* **in** *exI*)
apply (*blast intro*: *oadd-LimitI oadd-lt-self Limit-nat* [*THEN Limit-has-0*])
done

lemma *Ord2-imp-greater-Limit*: [|*Ord*(*i*); *Ord*(*j*)|] ==> ∃ *k*. *i*<*k* & *j*<*k* & *Limit*(*k*)
apply (*insert Ord-Un* [*of i j*, *THEN Ord-imp-greater-Limit*])
apply (*simp add*: *Un-least-lt-iff*)
done

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*))
apply (*rule-tac d* = *case* (%*x*. *x*, %*y*. *y*) **in** *lam-bijective*)
apply (*blast intro*!: *if-type*)
apply (*fast intro*!: *case-type*)
apply (*erule-tac* [*2*] *sumE*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *ordertype-sum-Diff*:
  i le j ==>
    ordertype(*i*+(*j*−*i*), *radd*(*i*,*Memrel*(*j*),*j*−*i*,*Memrel*(*j*))) =
    ordertype(*j*, *Memrel*(*j*))
apply (*safe dest*!: *le-subset-iff* [*THEN iffD1*])
apply (*rule bij-sum-Diff* [*THEN ord-isoI*, *THEN ord-iso-sym*, *THEN ordertype-eq*])
apply (*erule-tac* [*3*] *well-ord-Memrel*, *assumption*)
apply (*simp* (*no-asm-simp*))
apply (*frule-tac j* = *y* **in** *Ord-in-Ord*, *assumption*)
apply (*frule-tac j* = *x* **in** *Ord-in-Ord*, *assumption*)
apply (*simp* (*no-asm-simp*) *add*: *Ord-mem-iff-lt lt-Ord not-lt-iff-le*)
apply (*blast intro*: *lt-trans2 lt-trans*)
done

lemma *Ord-odiff* [*simp*,*TC*]:
  [| *Ord*(*i*); *Ord*(*j*) |] ==> *Ord*(*i*−−*j*)
apply (*unfold odiff-def*)
apply (*blast intro*: *Ord-ordertype Diff-subset well-ord-subset well-ord-Memrel*)
done

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*)))
apply (*simp add*: *raw-oadd-def odiff-def*)
apply (*safe dest*!: *le-subset-iff* [*THEN iffD1*])
apply (*rule sum-ord-iso-cong* [*THEN ordertype-eq*])
apply (*erule id-ord-iso-Memrel*)
apply (*rule ordertype-ord-iso* [*THEN ord-iso-sym*])
apply (*blast intro*: *well-ord-radd Diff-subset well-ord-subset well-ord-Memrel*)+
done

lemma *oadd-odiff-inverse*: *i le j* ==> *i* ++ (*j*−−*i*) = *j*
by (*simp add*: *lt-Ord le-Ord2 oadd-def ordify-def raw-oadd-ordertype-Diff*
  ordertype-sum-Diff ordertype-Memrel lt-Ord2 [*THEN Ord-succD*])

lemma *odiff-oadd-inverse*: [| *Ord*(*i*); *Ord*(*j*) |] ==> (*i*++*j*) −− *i* = *j*
apply (*rule oadd-inject*)
apply (*blast intro*: *oadd-odiff-inverse oadd-le-self*)
apply (*blast intro*: *Ord-ordertype Ord-oadd Ord-odiff*)+
done

lemma *odiff-lt-mono2*: [| *i*<*j*; *k le i* |] ==> *i*−−*k* < *j*−−*k*
apply (*rule-tac i* = *k* **in** *oadd-lt-cancel2*)
apply (*simp add*: *oadd-odiff-inverse*)
apply (*subst oadd-odiff-inverse*)
apply (*blast intro*: *le-trans leI*, *assumption*)
apply (*simp* (*no-asm-simp*) *add*: *lt-Ord le-Ord2*)
done

20.6 Ordinal Multiplication

lemma *Ord-omult* [*simp*,*TC*]:
  [| *Ord*(*i*); *Ord*(*j*) |] ==> *Ord*(*i****j*)
apply (*unfold omult-def*)
apply (*blast intro*: *Ord-ordertype well-ord-rmult well-ord-Memrel*)
done

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*))
apply (*unfold pred-def*, *blast*)
done

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))
apply (simp (no-asm-simp) add: pred-Pair-eq)
apply (rule ordertype-eq [symmetric])
apply (rule prod-sum-singleton-ord-iso)
apply (simp-all add: pred-subset well-ord-rmult [THEN well-ord-subset])
apply (blast intro: pred-subset well-ord-rmult [THEN well-ord-subset]
          elim!: predE)
done

lemma ordertype-pred-Pair-lemma:
   [| 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')
apply (unfold raw-oadd-def omult-def)
apply (simp add: ordertype-pred-Pair-eq lt-pred-Memrel ltD lt-Ord2
              well-ord-Memrel)
apply (rule trans)
 apply (rule-tac [2] ordertype-ord-iso
                 [THEN sum-ord-iso-cong, THEN ordertype-eq])
  apply (rule-tac [3] ord-iso-refl)
apply (rule id-bij [THEN ord-isoI, THEN ordertype-eq])
apply (elim SigmaE sumE ltE ssubst)
apply (simp-all add: well-ord-rmult well-ord-radd well-ord-Memrel
                 Ord-ordertype lt-Ord lt-Ord2)
apply (blast intro: Ord-trans)+
done

lemma lt-omult:
 [| Ord(i);  Ord(j);  k<j**i |]
  ==> EX j' i'. k = j**i' ++ j' & j'<j & i'<i
apply (unfold omult-def)
apply (simp add: ordertype-pred-unfold well-ord-rmult well-ord-Memrel)
apply (safe elim!: ltE)
apply (simp add: ordertype-pred-Pair-lemma ltI raw-oadd-eq-oadd
          omult-def [symmetric] Ord-in-Ord' [of - i] Ord-in-Ord' [of - j])
apply (blast intro: ltI)
done

lemma omult-oadd-lt:
    [| j'<j;  i'<i |] ==> j**i' ++ j'  <  j**i
apply (unfold omult-def)
apply (rule ltI)
 prefer 2
 apply (simp add: Ord-ordertype well-ord-rmult well-ord-Memrel lt-Ord2)
apply (simp add: ordertype-pred-unfold well-ord-rmult well-ord-Memrel lt-Ord2)
apply (rule bexI [of - i'])
apply (rule bexI [of - j'])
apply (simp add: ordertype-pred-Pair-lemma ltI omult-def [symmetric])
```

apply (*simp add*: *lt-Ord lt-Ord2 raw-oadd-eq-oadd*)
apply (*simp-all add*: *lt-def*)
done

lemma *omult-unfold*:
  [| *Ord*(*i*); *Ord*(*j*) |] ==> $j{*}{*}i = (\bigcup j'{\in}j.\ \bigcup i'{\in}i.\ \{j{*}{*}i' \mathbin{+}\!\!+ j'\})$
apply (*rule subsetI* [*THEN equalityI*])
apply (*rule lt-omult* [*THEN exE*])
apply (*erule-tac* [*3*] *ltI*)
apply (*simp-all add*: *Ord-omult*)
apply (*blast elim*!: *ltE*)
apply (*blast intro*: *omult-oadd-lt* [*THEN ltD*] *ltI*)
done

20.6.2 Basic laws for ordinal multiplication

Ordinal multiplication by zero

lemma *omult-0* [*simp*]: *i*∗∗*0* = *0*
apply (*unfold omult-def*)
apply (*simp* (*no-asm-simp*))
done

lemma *omult-0-left* [*simp*]: *0*∗∗*i* = *0*
apply (*unfold omult-def*)
apply (*simp* (*no-asm-simp*))
done

Ordinal multiplication by 1

lemma *omult-1* [*simp*]: *Ord*(*i*) ==> *i*∗∗*1* = *i*
apply (*unfold omult-def*)
apply (*rule-tac s1*=*Memrel*(*i*)
  in *ord-isoI* [*THEN ordertype-eq*, *THEN trans*])
apply (*rule-tac c* = *snd* **and** *d* = %*z*.<*0*,*z*> **in** *lam-bijective*)
apply (*auto elim*!: *snd-type well-ord-Memrel ordertype-Memrel*)
done

lemma *omult-1-left* [*simp*]: *Ord*(*i*) ==> *1*∗∗*i* = *i*
apply (*unfold omult-def*)
apply (*rule-tac s1*=*Memrel*(*i*)
  in *ord-isoI* [*THEN ordertype-eq*, *THEN trans*])
apply (*rule-tac c* = *fst* **and** *d* = %*z*.<*z*,*0*> **in** *lam-bijective*)
apply (*auto elim*!: *fst-type well-ord-Memrel ordertype-Memrel*)
done

Distributive law for ordinal multiplication and addition

lemma *oadd-omult-distrib*:
  [| *Ord*(*i*); *Ord*(*j*); *Ord*(*k*) |] ==> *i*∗∗(*j*++*k*) = (*i*∗∗*j*)++(*i*∗∗*k*)
apply (*simp add*: *oadd-eq-if-raw-oadd*)

apply (*simp add*: *omult-def raw-oadd-def*)
apply (*rule ordertype-eq* [*THEN trans*])
apply (*rule prod-ord-iso-cong* [*OF ordertype-ord-iso* [*THEN ord-iso-sym*]
ord-iso-refl])
apply (*simp-all add*: *well-ord-rmult well-ord-radd well-ord-Memrel*
Ord-ordertype)
apply (*rule sum-prod-distrib-ord-iso* [*THEN ordertype-eq*, *THEN trans*])
apply (*rule-tac* [*2*] *ordertype-eq*)
apply (*rule-tac* [*2*] *sum-ord-iso-cong* [*OF ordertype-ord-iso ordertype-ord-iso*])
apply (*simp-all add*: *well-ord-rmult well-ord-radd well-ord-Memrel*
Ord-ordertype)
done

lemma *omult-succ*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *i****succ*(*j*) = (*i****j*)++*i*
by (*simp del*: *oadd-succ add*: *oadd-1* [*of j*, *symmetric*] *oadd-omult-distrib*)

Associative law

lemma *omult-assoc*:
[| *Ord*(*i*); *Ord*(*j*); *Ord*(*k*) |] ==> (*i****j*)***k* = *i***(*j****k*)
apply (*unfold omult-def*)
apply (*rule ordertype-eq* [*THEN trans*])
apply (*rule prod-ord-iso-cong* [*OF ord-iso-refl*
ordertype-ord-iso [*THEN ord-iso-sym*]])
apply (*blast intro*: *well-ord-rmult well-ord-Memrel*)+
apply (*rule prod-assoc-ord-iso*
[*THEN ord-iso-sym*, *THEN ordertype-eq*, *THEN trans*])
apply (*rule-tac* [*2*] *ordertype-eq*)
apply (*rule-tac* [*2*] *prod-ord-iso-cong* [*OF ordertype-ord-iso ord-iso-refl*])
apply (*blast intro*: *well-ord-rmult well-ord-Memrel Ord-ordertype*)+
done

Ordinal multiplication with limit ordinals

lemma *omult-UN*:
[| *Ord*(*i*); !!*x*. *x*:*A* ==> *Ord*(*j*(*x*)) |]
==> *i* ** ($\bigcup$*x*∈*A*. *j*(*x*)) = ($\bigcup$*x*∈*A*. *i****j*(*x*))
by (*simp* (*no-asm-simp*) *add*: *Ord-UN omult-unfold*, *blast*)

lemma *omult-Limit*: [| *Ord*(*i*); *Limit*(*j*) |] ==> *i****j* = ($\bigcup$*k*∈*j*. *i****k*)
by (*simp add*: *Limit-is-Ord* [*THEN Ord-in-Ord*] *omult-UN* [*symmetric*]
Union-eq-UN [*symmetric*] *Limit-Union-eq*)

20.6.3 Ordering/monotonicity properties of ordinal multiplication

lemma *lt-omult1*: [| *k*<*i*; *0*<*j* |] ==> *k* < *i****j*
apply (*safe elim*!: *ltE intro*!: *ltI Ord-omult*)
apply (*force simp add*: *omult-unfold*)
done

lemma *omult-le-self*: [| *Ord*(*i*); *0*<*j* |] ==> *i le i*∗∗*j*
by (*blast intro*: *all-lt-imp-le Ord-omult lt-omult1 lt-Ord2*)

lemma *omult-le-mono1*: [| *k le j*; *Ord*(*i*) |] ==> *k*∗∗*i le j*∗∗*i*
apply (*frule lt-Ord*)
apply (*frule le-Ord2*)
apply (*erule trans-induct3*)
apply (*simp* (*no-asm-simp*) *add*: *le-refl Ord-0*)
apply (*simp* (*no-asm-simp*) *add*: *omult-succ oadd-le-mono*)
apply (*simp* (*no-asm-simp*) *add*: *omult-Limit*)
apply (*rule le-implies-UN-le-UN*, *blast*)
done

lemma *omult-lt-mono2*: [| *k*<*j*; *0*<*i* |] ==> *i*∗∗*k* < *i*∗∗*j*
apply (*rule ltI*)
apply (*simp* (*no-asm-simp*) *add*: *omult-unfold lt-Ord2*)
apply (*safe elim*!: *ltE intro*!: *Ord-omult*)
apply (*force simp add*: *Ord-omult*)
done

lemma *omult-le-mono2*: [| *k le j*; *Ord*(*i*) |] ==> *i*∗∗*k le i*∗∗*j*
apply (*rule subset-imp-le*)
apply (*safe elim*!: *ltE dest*!: *Ord-succD intro*!: *Ord-omult*)
apply (*simp add*: *omult-unfold*)
apply (*blast intro*: *Ord-trans*)
done

lemma *omult-le-mono*: [| *i′ le i*; *j′ le j* |] ==> *i′*∗∗*j′ le i*∗∗*j*
by (*blast intro*: *le-trans omult-le-mono1 omult-le-mono2 Ord-succD elim*: *ltE*)

lemma *omult-lt-mono*: [| *i′ le i*; *j′*<*j*; *0*<*i* |] ==> *i′*∗∗*j′* < *i*∗∗*j*
by (*blast intro*: *lt-trans1 omult-le-mono1 omult-lt-mono2 Ord-succD elim*: *ltE*)

lemma *omult-le-self2*: [| *Ord*(*i*); *0*<*j* |] ==> *i le j*∗∗*i*
apply (*frule lt-Ord2*)
apply (*erule-tac i* = *i* **in** *trans-induct3*)
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*) *add*: *omult-succ*)
apply (*erule lt-trans1*)
apply (*rule-tac b* = *j*∗∗*x* **in** *oadd-0* [*THEN subst*], *rule-tac* [*2*] *oadd-lt-mono2*)
apply (*blast intro*: *Ord-omult*, *assumption*)
apply (*simp* (*no-asm-simp*) *add*: *omult-Limit*)
apply (*rule le-trans*)
apply (*rule-tac* [*2*] *le-implies-UN-le-UN*)
prefer *2* **apply** *blast*
apply (*simp* (*no-asm-simp*) *add*: *Union-eq-UN* [*symmetric*] *Limit-Union-eq Limit-is-Ord*)
done

Further properties of ordinal multiplication

lemma *omult-inject*: [| *i****j* = *i****k*; *0*<*i*; *Ord*(*j*); *Ord*(*k*) |] ==> *j*=*k*
apply (*rule Ord-linear-lt*)
prefer *4* **apply** *assumption*
apply *auto*
apply (*force dest*: *omult-lt-mono2 simp add*: *lt-not-refl*)+
done

20.7 The Relation *Lt*

lemma *wf-Lt*: *wf*(*Lt*)
apply (*rule wf-subset*)
apply (*rule wf-Memrel*)
apply (*auto simp add*: *Lt-def Memrel-def lt-def*)
done

lemma *irrefl-Lt*: *irrefl*(*A*,*Lt*)
by (*auto simp add*: *Lt-def irrefl-def*)

lemma *trans-Lt*: *trans*[*A*](*Lt*)
apply (*simp add*: *Lt-def trans-on-def*)
apply (*blast intro*: *lt-trans*)
done

lemma *part-ord-Lt*: *part-ord*(*A*,*Lt*)
by (*simp add*: *part-ord-def irrefl-Lt trans-Lt*)

lemma *linear-Lt*: *linear*(*nat*,*Lt*)
apply (*auto dest*!: *not-lt-imp-le simp add*: *Lt-def linear-def le-iff*)
apply (*drule lt-asym*, *auto*)
done

lemma *tot-ord-Lt*: *tot-ord*(*nat*,*Lt*)
by (*simp add*: *tot-ord-def linear-Lt part-ord-Lt*)

lemma *well-ord-Lt*: *well-ord*(*nat*,*Lt*)
by (*simp add*: *well-ord-def wf-Lt wf-imp-wf-on tot-ord-Lt*)

end

21 Finite: Finite Powerset Operator and Finite Function Space

theory *Finite* **imports** *Inductive-ZF Epsilon Nat-ZF* **begin**

rep-datatype
  elimination *natE*

induction *nat-induct*
case-eqns *nat-case-0 nat-case-succ*
recursor-eqns *recursor-0 recursor-succ*

consts
Fin :: *i*=>*i*
FiniteFun :: [*i*,*i*]=>*i* ((- −||>/ -) [*61*, *60*] *60*)

inductive
domains *Fin*(*A*) <= *Pow*(*A*)
intros
emptyI: *0* : *Fin*(*A*)
consI: [| *a*: *A*; *b*: *Fin*(*A*) |] ==> *cons*(*a*,*b*) : *Fin*(*A*)
type-intros *empty-subsetI cons-subsetI PowI*
type-elims *PowD* [*THEN revcut-rl*]

inductive
domains *FiniteFun*(*A*,*B*) <= *Fin*(*A*∗*B*)
intros
emptyI: *0* : *A* −||> *B*
consI: [| *a*: *A*; *b*: *B*; *h*: *A* −||> *B*; *a* ~: *domain*(*h*) |]
==> *cons*(<*a*,*b*>,*h*) : *A* −||> *B*
type-intros *Fin.intros*

21.1 Finite Powerset Operator

lemma *Fin-mono*: *A*<=*B* ==> *Fin*(*A*) <= *Fin*(*B*)
apply (*unfold Fin.defs*)
apply (*rule lfp-mono*)
apply (*rule Fin.bnd-mono*)+
apply *blast*
done

lemmas *FinD* = *Fin.dom-subset* [*THEN subsetD*, *THEN PowD*, *standard*]

lemma *Fin-induct* [*case-names 0 cons*, *induct set*: *Fin*]:
[| *b*: *Fin*(*A*);
P(*0*);
!!*x y*. [| *x*: *A*; *y*: *Fin*(*A*); *x*~:*y*; *P*(*y*) |] ==> *P*(*cons*(*x*,*y*))
|] ==> *P*(*b*)
apply (*erule Fin.induct*, *simp*)
apply (*case-tac a*:*b*)
apply (*erule cons-absorb* [*THEN ssubst*], *assumption*)
apply *simp*

done

declare *Fin.intros* [*simp*]

lemma *Fin-0*: *Fin*(*0*) = {*0*}
by (*blast intro*: *Fin.emptyI dest*: *FinD*)

lemma *Fin-UnI* [*simp*]: [| *b*: *Fin*(*A*); *c*: *Fin*(*A*) |] ==> *b Un c* : *Fin*(*A*)
apply (*erule Fin-induct*)
apply (*simp-all add*: *Un-cons*)
done

lemma *Fin-UnionI*: *C* : *Fin*(*Fin*(*A*)) ==> *Union*(*C*) : *Fin*(*A*)
by (*erule Fin-induct*, *simp-all*)

lemma *Fin-subset-lemma* [*rule-format*]: *b*: *Fin*(*A*) ==> ∀ *z*. *z*<=*b* −−> *z*: *Fin*(*A*)
apply (*erule Fin-induct*)
apply (*simp add*: *subset-empty-iff*)
apply (*simp add*: *subset-cons-iff distrib-simps*, *safe*)
apply (*erule-tac b* = *z* **in** *cons-Diff* [*THEN subst*], *simp*)
done

lemma *Fin-subset*: [| *c*<=*b*; *b*: *Fin*(*A*) |] ==> *c*: *Fin*(*A*)
by (*blast intro*: *Fin-subset-lemma*)

lemma *Fin-IntI1* [*intro*,*simp*]: *b*: *Fin*(*A*) ==> *b Int c* : *Fin*(*A*)
by (*blast intro*: *Fin-subset*)

lemma *Fin-IntI2* [*intro*,*simp*]: *c*: *Fin*(*A*) ==> *b Int c* : *Fin*(*A*)
by (*blast intro*: *Fin-subset*)

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*)
apply (*erule Fin-induct*, *simp*)
apply (*subst Diff-cons*)
apply (*simp add*: *cons-subset-iff Diff-subset* [*THEN Fin-subset*])
done

lemma *Fin-0-induct*:
  [| *b*: *Fin*(*A*);
    P(*b*);

!!*x y*. [| *x*: *A*; *y*: *Fin*(*A*); *x*:*y*; *P*(*y*) |] ==> *P*(*y*−{*x*})
|] ==> *P*(*0*)
apply (*rule Diff-cancel* [*THEN subst*])
apply (*blast intro*: *Fin-0-induct-lemma*)
done

lemma *nat-fun-subset-Fin*: *n*: *nat* ==> *n*−>*A* <= *Fin*(*nat*∗*A*)
apply (*induct-tac n*)
apply (*simp add*: *subset-iff*)
apply (*simp add*: *succ-def mem-not-refl* [*THEN cons-fun-eq*])
apply (*fast intro*!: *Fin.consI*)
done

21.2 Finite Function Space

lemma *FiniteFun-mono*:
[| *A*<=*C*; *B*<=*D* |] ==> *A* −||> *B* <= *C* −||> *D*
apply (*unfold FiniteFun.defs*)
apply (*rule lfp-mono*)
apply (*rule FiniteFun.bnd-mono*)+
apply (*intro Fin-mono Sigma-mono basic-monos*, *assumption*+)
done

lemma *FiniteFun-mono1*: *A*<=*B* ==> *A* −||> *A* <= *B* −||> *B*
by (*blast dest*: *FiniteFun-mono*)

lemma *FiniteFun-is-fun*: *h*: *A* −||>*B* ==> *h*: *domain*(*h*) −> *B*
apply (*erule FiniteFun.induct*, *simp*)
apply (*simp add*: *fun-extend3*)
done

lemma *FiniteFun-domain-Fin*: *h*: *A* −||>*B* ==> *domain*(*h*) : *Fin*(*A*)
by (*erule FiniteFun.induct*, *simp*, *simp*)

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*
apply (*erule FiniteFun.induct*)
apply (*simp add*: *subset-empty-iff FiniteFun.intros*)
apply (*simp add*: *subset-cons-iff distrib-simps*, *safe*)
apply (*erule-tac b* = *z* **in** *cons-Diff* [*THEN subst*])
apply (*drule spec* [*THEN mp*], *assumption*)
apply (*fast intro*!: *FiniteFun.intros*)
done

lemma *FiniteFun-subset*: [| *c*<=*b*; *b*: *A*−||>*B* |] ==> *c*: *A*−||>*B*

by (*blast intro*: *FiniteFun-subset-lemma*)

lemma *fun-FiniteFunI* [*rule-format*]: *A*:*Fin*(*X*) ==> *ALL f*. *f*:*A*−>*B* −−> *f*:*A*−||>*B*
apply (*erule Fin.induct*)
 apply (*simp add*: *FiniteFun.intros*, *clarify*)
apply (*case-tac a*:*b*)
 apply (*simp add*: *cons-absorb*)
apply (*subgoal-tac restrict* (*f*,*b*) : *b* −||> *B*)
 prefer *2* **apply** (*blast intro*: *restrict-type2*)
apply (*subst fun-cons-restrict-eq*, *assumption*)
apply (*simp add*: *restrict-def lam-def*)
apply (*blast intro*: *apply-funtype FiniteFun.intros*
                    FiniteFun-mono [*THEN* [*2*] *rev-subsetD*])
done

lemma *lam-FiniteFun*: *A*: *Fin*(*X*) ==> (*lam x*:*A*. *b*(*x*)) : *A* −||> {*b*(*x*). *x*:*A*}
by (*blast intro*: *fun-FiniteFunI lam-funtype*)

lemma *FiniteFun-Collect-iff*:
     f : *FiniteFun*(*A*, {*y*:*B*. *P*(*y*)})
      <−> *f* : *FiniteFun*(*A*,*B*) & (*ALL x*:*domain*(*f*). *P*(*f*'*x*))
apply *auto*
apply (*blast intro*: *FiniteFun-mono* [*THEN* [*2*] *rev-subsetD*])
apply (*blast dest*: *Pair-mem-PiD FiniteFun-is-fun*)
apply (*rule-tac A1*=*domain*(*f*) **in**
       subset-refl [*THEN* [*2*] *FiniteFun-mono*, *THEN subsetD*])
 apply (*fast dest*: *FiniteFun-domain-Fin Fin.dom-subset* [*THEN subsetD*])
apply (*rule fun-FiniteFunI*)
apply (*erule FiniteFun-domain-Fin*)
apply (*rule-tac B* = *range* (*f*) **in** *fun-weaken-type*)
 apply (*blast dest*: *FiniteFun-is-fun range-of-fun range-type apply-equality*)+
done

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

end

22 Cardinal: Cardinal Numbers Without the Axiom of Choice

theory *Cardinal* **imports** *OrderType Finite Nat-ZF Sum* **begin**

definition

Least :: $(i{=}{>}o) => i$ (**binder** *LEAST* *10*) **where**
Least(P) == *THE* i. *Ord*(i) & $P(i)$ & (*ALL* j. $j<i$ --> ~$P(j)$)

definition
eqpoll :: [i,i] => o (**infixl** *eqpoll* *50*) **where**
A *eqpoll* B == *EX* f. f: *bij*(A,B)

definition
lepoll :: [i,i] => o (**infixl** *lepoll* *50*) **where**
A *lepoll* B == *EX* f. f: *inj*(A,B)

definition
lesspoll :: [i,i] => o (**infixl** *lesspoll* *50*) **where**
A *lesspoll* B == A *lepoll* B & ~(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-mono*(X, %W. X − g‘‘(Y − f‘‘W))

by (*rule bnd-monoI*, *blast+*)

lemma *Banach-last-equation*:
  g: *Y*−>*X*
  ==> *g*‘‘(*Y* − *f*‘‘ *lfp*(*X*, %*W*. *X* − *g*‘‘(*Y* − *f*‘‘*W*))) =
    X − *lfp*(*X*, %*W*. *X* − *g*‘‘(*Y* − *f*‘‘*W*))
apply (*rule-tac P* = %*u*. *?v* = *X*−*u*
  in *decomp-bnd-mono* [*THEN lfp-unfold*, *THEN ssubst*])
apply (*simp add*: *double-complement fun-is-rel* [*THEN image-subset*])
done

lemma *decomposition*:
  [| *f*: *X*−>*Y*; *g*: *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*
apply (*intro exI conjI*)
apply (*rule-tac* [*6*] *Banach-last-equation*)
apply (*rule-tac* [*5*] *refl*)
apply (*assumption* |
  rule Diff-disjoint Diff-partition fun-is-rel image-subset lfp-subset)+
done

lemma *schroeder-bernstein*:
  [| *f*: *inj*(*X*,*Y*); *g*: *inj*(*Y*,*X*) |] ==> *EX h*. *h*: *bij*(*X*,*Y*)
apply (*insert decomposition* [*of f X Y g*])
apply (*simp add*: *inj-is-fun*)
apply (*blast intro*!: *restrict-bij bij-disjoint-Un intro*: *bij-converse-bij*)

done

lemma *bij-imp-eqpoll*: *f*: *bij*(*A*,*B*) ==> *A* ≈ *B*
apply (*unfold eqpoll-def*)
apply (*erule exI*)
done

lemmas *eqpoll-refl* = *id-bij* [*THEN bij-imp-eqpoll*, *standard*, *simp*]

lemma *eqpoll-sym*: *X* ≈ *Y* ==> *Y* ≈ *X*
apply (*unfold eqpoll-def*)
apply (*blast intro*: *bij-converse-bij*)
done

lemma *eqpoll-trans*:
  [| *X* ≈ *Y*; *Y* ≈ *Z* |] ==> *X* ≈ *Z*

apply (*unfold eqpoll-def*)
apply (*blast intro*: *comp-bij*)
done

lemma *subset-imp-lepoll*: $X<=Y ==> X \lesssim Y$
apply (*unfold lepoll-def*)
apply (*rule exI*)
apply (*erule id-subset-inj*)
done

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*: $X \approx Y ==> X \lesssim Y$
by (*unfold eqpoll-def bij-def lepoll-def*, *blast*)

lemma *lepoll-trans*: $[| X \lesssim Y; \ Y \lesssim Z |] ==> X \lesssim Z$
apply (*unfold lepoll-def*)
apply (*blast intro*: *comp-inj*)
done

lemma *eqpollI*: $[| X \lesssim Y; \ Y \lesssim X |] ==> X \approx Y$
apply (*unfold lepoll-def eqpoll-def*)
apply (*elim exE*)
apply (*rule schroeder-bernstein*, *assumption+*)
done

lemma *eqpollE*:
  $[| X \approx Y; [| X \lesssim Y; Y \lesssim X |] ==> P |] ==> P$
by (*blast intro*: *eqpoll-imp-lepoll eqpoll-sym*)

lemma *eqpoll-iff*: $X \approx Y <-> X \lesssim Y \ \& \ Y \lesssim X$
by (*blast intro*: *eqpollI elim!*: *eqpollE*)

lemma *lepoll-0-is-0*: $A \lesssim 0 ==> A = 0$
apply (*unfold lepoll-def inj-def*)
apply (*blast dest*: *apply-type*)
done

lemmas *empty-lepollI* = *empty-subsetI* [*THEN subset-imp-lepoll*, *standard*]

lemma *lepoll-0-iff*: $A \lesssim 0 <-> A=0$
by (*blast intro*: *lepoll-0-is-0 lepoll-refl*)

lemma *Un-lepoll-Un*:
  [| $A \lesssim B$; $C \lesssim D$; *B Int D* = *0* |] ==> *A Un C* $\lesssim$ *B Un D*
apply (*unfold lepoll-def*)
apply (*blast intro*: *inj-disjoint-Un*)
done

lemmas *eqpoll-0-is-0* = *eqpoll-imp-lepoll* [*THEN lepoll-0-is-0*, *standard*]

lemma *eqpoll-0-iff*: $A \approx 0$ <−> *A*=*0*
by (*blast intro*: *eqpoll-0-is-0 eqpoll-refl*)

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*
apply (*unfold eqpoll-def*)
apply (*blast intro*: *bij-disjoint-Un*)
done

22.2 lesspoll: contributions by Krzysztof Grabczewski

lemma *lesspoll-not-refl*: ~ ($i \prec i$)
by (*simp add*: *lesspoll-def*)

lemma *lesspoll-irrefl* [*elim*!]: $i \prec i$ ==> *P*
by (*simp add*: *lesspoll-def*)

lemma *lesspoll-imp-lepoll*: $A \prec B$ ==> $A \lesssim B$
by (*unfold lesspoll-def*, *blast*)

lemma *lepoll-well-ord*: [| $A \lesssim B$; *well-ord*(*B*,*r*) |] ==> *EX s*. *well-ord*(*A*,*s*)
apply (*unfold lepoll-def*)
apply (*blast intro*: *well-ord-rvimage*)
done

lemma *lepoll-iff-leqpoll*: $A \lesssim B$ <−> $A \prec B$ | $A \approx B$
apply (*unfold lesspoll-def*)
apply (*blast intro*!: *eqpollI elim*!: *eqpollE*)
done

lemma *inj-not-surj-succ*:
  [| *f* : *inj*(*A*, *succ*(*m*)); *f* ~: *surj*(*A*, *succ*(*m*)) |] ==> *EX f*. *f*:*inj*(*A*,*m*)
apply (*unfold inj-def surj-def*)
apply (*safe del*: *succE*)
apply (*erule swap*, *rule exI*)
apply (*rule-tac a* = *lam z*:*A*. *if f'z*=*m then y else f'z* **in** *CollectI*)

the typing condition

apply (*best intro*!: *if-type* [*THEN lam-type*] *elim*: *apply-funtype* [*THEN succE*])

Proving it's injective

apply *simp*
apply *blast*
done

lemma *lesspoll-trans*:
  [| $X \prec Y$; $Y \prec Z$ |] ==> $X \prec Z$
apply (*unfold lesspoll-def*)
apply (*blast elim*!: *eqpollE intro*: *eqpollI lepoll-trans*)
done

lemma *lesspoll-trans1*:
  [| $X \lesssim Y$; $Y \prec Z$ |] ==> $X \prec Z$
apply (*unfold lesspoll-def*)
apply (*blast elim*!: *eqpollE intro*: *eqpollI lepoll-trans*)
done

lemma *lesspoll-trans2*:
  [| $X \prec Y$; $Y \lesssim Z$ |] ==> $X \prec Z$
apply (*unfold lesspoll-def*)
apply (*blast elim*!: *eqpollE intro*: *eqpollI lepoll-trans*)
done

lemma *Least-equality*:
  [| $P(i)$; $Ord(i)$; !!x. x<i ==> ~$P(x)$ |] ==> ($LEAST\ x.\ P(x)$) = i
apply (*unfold Least-def*)
apply (*rule the-equality*, *blast*)
apply (*elim conjE*)
apply (*erule Ord-linear-lt*, *assumption*, *blast*+)
done

lemma *LeastI*: [| $P(i)$; $Ord(i)$ |] ==> $P(LEAST\ x.\ P(x))$
apply (*erule rev-mp*)
apply (*erule-tac i*=*i* **in** *trans-induct*)
apply (*rule impI*)
apply (*rule classical*)
apply (*blast intro*: *Least-equality* [*THEN ssubst*] *elim*!: *ltE*)
done

lemma *Least-le*: [| $P(i)$; $Ord(i)$ |] ==> ($LEAST\ x.\ P(x)$) *le* i
apply (*erule rev-mp*)
apply (*erule-tac i*=*i* **in** *trans-induct*)
apply (*rule impI*)

apply (*rule classical*)
apply (*subst Least-equality*, *assumption+*)
apply (*erule-tac* [*2*] *le-refl*)
apply (*blast elim*: *ltE intro*: *leI ltI lt-trans1*)
done

lemma *less-LeastE*: [| $P(i)$; $i <$ (*LEAST* x. $P(x)$) |] ==> Q
apply (*rule Least-le* [*THEN* [*2*] *lt-trans2*, *THEN lt-irrefl*], *assumption+*)
apply (*simp add*: *lt-Ord*)
done

lemma *LeastI2*:
[| $P(i)$; *Ord*(i); !!j. $P(j)$ ==> $Q(j)$ |] ==> Q(*LEAST* j. $P(j)$)
by (*blast intro*: *LeastI*)

lemma *Least-0*:
[| ~ (*EX* i. *Ord*(i) & $P(i)$) |] ==> (*LEAST* x. $P(x)$) = *0*
apply (*unfold Least-def*)
apply (*rule the-0*, *blast*)
done

lemma *Ord-Least* [*intro*,*simp*,*TC*]: *Ord*(*LEAST* x. $P(x)$)
apply (*case-tac* $\exists$ i. *Ord*(i) & $P(i)$)
apply *safe*
apply (*rule Least-le* [*THEN ltE*])
prefer *3* **apply** *assumption+*
apply (*erule Least-0* [*THEN ssubst*])
apply (*rule Ord-0*)
done

lemma *Least-cong*:
(!!y. $P(y)$ <-> $Q(y)$) ==> (*LEAST* x. $P(x)$) = (*LEAST* x. $Q(x)$)
by *simp*

lemma *cardinal-cong*: $X \approx Y$ ==> $|X| = |Y|$
apply (*unfold eqpoll-def cardinal-def*)
apply (*rule Least-cong*)
apply (*blast intro*: *comp-bij bij-converse-bij*)
done

lemma *well-ord-cardinal-eqpoll*:
well-ord(*A*,*r*) ==> |*A*| ≈ *A*
apply (*unfold cardinal-def*)
apply (*rule LeastI*)
apply (*erule-tac* [2] *Ord-ordertype*)
apply (*erule ordermap-bij* [*THEN bij-converse-bij*, *THEN bij-imp-eqpoll*])
done

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* ≈ *Y*
apply (*rule eqpoll-sym* [*THEN eqpoll-trans*])
apply (*erule well-ord-cardinal-eqpoll*)
apply (*simp* (*no-asm-simp*) *add*: *well-ord-cardinal-eqpoll*)
done

lemma *well-ord-cardinal-eqpoll-iff*:
[| *well-ord*(*X*,*r*); *well-ord*(*Y*,*s*) |] ==> |*X*| = |*Y*| <-> *X* ≈ *Y*
by (*blast intro*: *cardinal-cong well-ord-cardinal-eqE*)

lemma *Ord-cardinal-le*: *Ord*(*i*) ==> |*i*| *le i*
apply (*unfold cardinal-def*)
apply (*erule eqpoll-refl* [*THEN Least-le*])
done

lemma *Card-cardinal-eq*: *Card*(*K*) ==> |*K*| = *K*
apply (*unfold Card-def*)
apply (*erule sym*)
done

lemma *CardI*: [| *Ord*(*i*); !!*j*. *j*<*i* ==> ~(*j* ≈ *i*) |] ==> *Card*(*i*)
apply (*unfold Card-def cardinal-def*)
apply (*subst Least-equality*)
apply (*blast intro*: *eqpoll-refl*)+
done

lemma *Card-is-Ord*: *Card*(*i*) ==> *Ord*(*i*)
apply (*unfold Card-def cardinal-def*)
apply (*erule ssubst*)
apply (*rule Ord-Least*)
done

lemma *Card-cardinal-le*: *Card*(*K*) ==> *K le* |*K*|

apply (*simp* (*no-asm-simp*) *add*: *Card-is-Ord Card-cardinal-eq*)
done

lemma *Ord-cardinal* [*simp*,*intro*!]: *Ord*(|*A*|)
apply (*unfold cardinal-def*)
apply (*rule Ord-Least*)
done

lemma *Card-iff-initial*: *Card*(*K*) <−> *Ord*(*K*) & (*ALL j*. *j*<*K* −−> ~ *j* ≈ *K*)
apply (*safe intro*!: *CardI Card-is-Ord*)
prefer *2* **apply** *blast*
apply (*unfold Card-def cardinal-def*)
apply (*rule less-LeastE*)
apply (*erule-tac* [*2*] *subst*, *assumption*+)
done

lemma *lt-Card-imp-lesspoll*: [| *Card*(*a*); *i*<*a* |] ==> *i* ≺ *a*
apply (*unfold lesspoll-def*)
apply (*drule Card-iff-initial* [*THEN iffD1*])
apply (*blast intro*!: *leI* [*THEN le-imp-lepoll*])
done

lemma *Card-0*: *Card*(*0*)
apply (*rule Ord-0* [*THEN CardI*])
apply (*blast elim*!: *ltE*)
done

lemma *Card-Un*: [| *Card*(*K*); *Card*(*L*) |] ==> *Card*(*K Un L*)
apply (*rule Ord-linear-le* [*of K L*])
apply (*simp-all add*: *subset-Un-iff* [*THEN iffD1*] *Card-is-Ord le-imp-subset*
subset-Un-iff2 [*THEN iffD1*])
done

lemma *Card-cardinal*: *Card*(|*A*|)
apply (*unfold cardinal-def*)
apply (*case-tac EX i*. *Ord* (*i*) & *i* ≈ *A*)

degenerate case

prefer *2* **apply** (*erule Least-0* [*THEN ssubst*], *rule Card-0*)

real case: A is isomorphic to some ordinal

apply (*rule Ord-Least* [*THEN CardI*], *safe*)
apply (*rule less-LeastE*)
prefer *2* **apply** *assumption*
apply (*erule eqpoll-trans*)
apply (*best intro*: *LeastI*)

done

lemma *cardinal-eq-lemma*: [| |*i*| *le j*; *j le i* |] ==> |*j*| = |*i*|
apply (*rule eqpollI* [*THEN cardinal-cong*])
apply (*erule le-imp-lepoll*)
apply (*rule lepoll-trans*)
apply (*erule-tac* [*2*] *le-imp-lepoll*)
apply (*rule eqpoll-sym* [*THEN eqpoll-imp-lepoll*])
apply (*rule Ord-cardinal-eqpoll*)
apply (*elim ltE Ord-succD*)
done

lemma *cardinal-mono*: *i le j* ==> |*i*| *le* |*j*|
apply (*rule-tac i* = |*i*| **and** *j* = |*j*| **in** *Ord-linear-le*)
apply (*safe intro*!: *Ord-cardinal le-eqI*)
apply (*rule cardinal-eq-lemma*)
prefer *2* **apply** *assumption*
apply (*erule le-trans*)
apply (*erule ltE*)
apply (*erule Ord-cardinal-le*)
done

lemma *cardinal-lt-imp-lt*: [| |*i*| < |*j*|; *Ord*(*i*); *Ord*(*j*) |] ==> *i* < *j*
apply (*rule Ord-linear2* [*of i j*], *assumption*+)
apply (*erule lt-trans2* [*THEN lt-irrefl*])
apply (*erule cardinal-mono*)
done

lemma *Card-lt-imp-lt*: [| |*i*| < *K*; *Ord*(*i*); *Card*(*K*) |] ==> *i* < *K*
apply (*simp* (*no-asm-simp*) *add*: *cardinal-lt-imp-lt Card-is-Ord Card-cardinal-eq*)
done

lemma *Card-lt-iff*: [| *Ord*(*i*); *Card*(*K*) |] ==> (|*i*| < *K*) <−> (*i* < *K*)
by (*blast intro*: *Card-lt-imp-lt Ord-cardinal-le* [*THEN lt-trans1*])

lemma *Card-le-iff*: [| *Ord*(*i*); *Card*(*K*) |] ==> (*K le* |*i*|) <−> (*K le i*)
by (*simp add*: *Card-lt-iff Card-is-Ord Ord-cardinal not-lt-iff-le* [*THEN iff-sym*])

lemma *well-ord-lepoll-imp-Card-le*:
[| *well-ord*(*B*,*r*); $A \lesssim B$ |] ==> |*A*| *le* |*B*|
apply (*rule-tac i* = |*A*| **and** *j* = |*B*| **in** *Ord-linear-le*)
apply (*safe intro*!: *Ord-cardinal le-eqI*)
apply (*rule eqpollI* [*THEN cardinal-cong*], *assumption*)
apply (*rule lepoll-trans*)
apply (*rule well-ord-cardinal-eqpoll* [*THEN eqpoll-sym*, *THEN eqpoll-imp-lepoll*], *assumption*)

apply (*erule le-imp-lepoll* [*THEN lepoll-trans*])
apply (*rule eqpoll-imp-lepoll*)
apply (*unfold lepoll-def*)
apply (*erule exE*)
apply (*rule well-ord-cardinal-eqpoll*)
apply (*erule well-ord-rvimage*, *assumption*)
done

lemma *lepoll-cardinal-le*: [| $A \lesssim i$; *Ord*(*i*) |] ==> |*A*| *le i*
apply (*rule le-trans*)
apply (*erule well-ord-Memrel* [*THEN well-ord-lepoll-imp-Card-le*], *assumption*)
apply (*erule Ord-cardinal-le*)
done

lemma *lepoll-Ord-imp-eqpoll*: [| $A \lesssim i$; *Ord*(*i*) |] ==> |*A*| $\approx$ *A*
by (*blast intro*: *lepoll-cardinal-le well-ord-Memrel well-ord-cardinal-eqpoll dest*!: *lepoll-well-ord*)

lemma *lesspoll-imp-eqpoll*: [| $A \prec i$; *Ord*(*i*) |] ==> |*A*| $\approx$ *A*
apply (*unfold lesspoll-def*)
apply (*blast intro*: *lepoll-Ord-imp-eqpoll*)
done

lemma *cardinal-subset-Ord*: [|*A*<=*i*; *Ord*(*i*)|] ==> |*A*| <= *i*
apply (*drule subset-imp-lepoll* [*THEN lepoll-cardinal-le*])
apply (*auto simp add*: *lt-def*)
apply (*blast intro*: *Ord-trans*)
done

22.3 The finite cardinals

lemma *cons-lepoll-consD*:
[| *cons*(*u*,*A*) $\lesssim$ *cons*(*v*,*B*); *u*~:*A*; *v*~:*B* |] ==> $A \lesssim B$
apply (*unfold lepoll-def inj-def*, *safe*)
apply (*rule-tac x* = *lam x*:*A*. *if f*‘*x*=*v then f*‘*u else f*‘*x* **in** *exI*)
apply (*rule CollectI*)

apply (*rule if-type* [*THEN lam-type*])
apply (*blast dest*: *apply-funtype*)
apply (*blast elim*!: *mem-irrefl dest*: *apply-funtype*)

apply (*simp* (*no-asm-simp*))
apply *blast*
done

lemma *cons-eqpoll-consD*: [| *cons*(*u*,*A*) $\approx$ *cons*(*v*,*B*); *u*~:*A*; *v*~:*B* |] ==> *A* $\approx$
B
apply (*simp add*: *eqpoll-iff*)
apply (*blast intro*: *cons-lepoll-consD*)
done

lemma *succ-lepoll-succD*: *succ*(*m*) $\lesssim$ *succ*(*n*) ==> *m* $\lesssim$ *n*
apply (*unfold succ-def*)
apply (*erule cons-lepoll-consD*)
apply (*rule mem-not-refl*)+
done

lemma *nat-lepoll-imp-le* [*rule-format*]:
m:*nat* ==> *ALL n*: *nat*. *m* $\lesssim$ *n* −−> *m le n*
apply (*induct-tac m*)
apply (*blast intro*!: *nat-0-le*)
apply (*rule ballI*)
apply (*erule-tac n* = *n* **in** *natE*)
apply (*simp* (*no-asm-simp*) *add*: *lepoll-def inj-def*)
apply (*blast intro*!: *succ-leI dest*!: *succ-lepoll-succD*)
done

lemma *nat-eqpoll-iff*: [| *m*:*nat*; *n*: *nat* |] ==> *m* $\approx$ *n* <−> *m* = *n*
apply (*rule iffI*)
apply (*blast intro*: *nat-lepoll-imp-le le-anti-sym elim*!: *eqpollE*)
apply (*simp add*: *eqpoll-refl*)
done

lemma *nat-into-Card*:
n: *nat* ==> *Card*(*n*)
apply (*unfold Card-def cardinal-def*)
apply (*subst Least-equality*)
apply (*rule eqpoll-refl*)
apply (*erule nat-into-Ord*)
apply (*simp* (*no-asm-simp*) *add*: *lt-nat-in-nat* [*THEN nat-eqpoll-iff*])
apply (*blast elim*!: *lt-irrefl*)+
done

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*
by (*rule nat-lepoll-imp-le* [*THEN lt-irrefl*], *auto*)

lemma *n-lesspoll-nat*: *n* $\in$ *nat* ==> *n* $\prec$ *nat*
apply (*unfold lesspoll-def*)
apply (*fast elim*!: *Ord-nat* [*THEN* [*2*] *ltI* [*THEN leI*, *THEN le-imp-lepoll*]]
eqpoll-sym [*THEN eqpoll-imp-lepoll*]
intro: *Ord-nat* [*THEN* [*2*] *nat-succI* [*THEN ltI*], *THEN leI*,
THEN le-imp-lepoll, *THEN lepoll-trans*, *THEN succ-lepoll-natE*])

done

lemma *nat-lepoll-imp-ex-eqpoll-n*:
  $[| n \in nat; \ nat \lesssim X |] ==> \exists Y. Y \subseteq X \ \& \ n \approx Y$
apply (*unfold lepoll-def eqpoll-def*)
apply (*fast del*: *subsetI subsetCE*
  intro!: *subset-SIs*
  dest!: *Ord-nat* [*THEN* [*2*] *OrdmemD*, *THEN* [*2*] *restrict-inj*]
  elim!: *restrict-bij*
   inj-is-fun [*THEN fun-is-rel*, *THEN image-subset*])
done

lemma *lepoll-imp-lesspoll-succ*:
  $[| A \lesssim m; m{:}nat |] ==> A \prec succ(m)$
apply (*unfold lesspoll-def*)
apply (*rule conjI*)
apply (*blast intro*: *subset-imp-lepoll* [*THEN* [*2*] *lepoll-trans*])
apply (*rule notI*)
apply (*drule eqpoll-sym* [*THEN eqpoll-imp-lepoll*])
apply (*drule lepoll-trans*, *assumption*)
apply (*erule succ-lepoll-natE*, *assumption*)
done

lemma *lesspoll-succ-imp-lepoll*:
  $[| A \prec succ(m); m{:}nat |] ==> A \lesssim m$
apply (*unfold lesspoll-def lepoll-def eqpoll-def bij-def*, *clarify*)
apply (*blast intro*!: *inj-not-surj-succ*)
done

lemma *lesspoll-succ-iff*: $m{:}nat ==> A \prec succ(m) <-> A \lesssim m$
by (*blast intro*!: *lepoll-imp-lesspoll-succ lesspoll-succ-imp-lepoll*)

lemma *lepoll-succ-disj*: $[| A \lesssim succ(m); \ m{:}nat |] ==> A \lesssim m \mid A \approx succ(m)$
apply (*rule disjCI*)
apply (*rule lesspoll-succ-imp-lepoll*)
prefer *2* **apply** *assumption*
apply (*simp* (*no-asm-simp*) *add*: *lesspoll-def*)
done

lemma *lesspoll-cardinal-lt*: $[| A \prec i; Ord(i) |] ==> |A| < i$
apply (*unfold lesspoll-def*, *clarify*)
apply (*frule lepoll-cardinal-le*, *assumption*)
apply (*blast intro*: *well-ord-Memrel well-ord-cardinal-eqpoll* [*THEN eqpoll-sym*]
  dest: *lepoll-well-ord* *elim*!: *leE*)
done

22.4 The first infinite cardinal: Omega, or nat

lemma *lt-not-lepoll*: [| $n<i$; n:*nat* |] ==> ~ $i \lesssim n$
apply (*rule notI*)
apply (*rule succ-lepoll-natE* [*of n*])
apply (*rule lepoll-trans* [*of - i*])
apply (*erule ltE*)
apply (*rule Ord-succ-subsetI* [*THEN subset-imp-lepoll*], *assumption*+)
done

lemma *Ord-nat-eqpoll-iff*: [| *Ord*(i); n:*nat* |] ==> $i \approx n$ <-> $i=n$
apply (*rule iffI*)
prefer *2* **apply** (*simp add*: *eqpoll-refl*)
apply (*rule Ord-linear-lt* [*of i n*])
apply (*simp-all add*: *nat-into-Ord*)
apply (*erule lt-nat-in-nat* [*THEN nat-eqpoll-iff*, *THEN iffD1*], *assumption*+)
apply (*rule lt-not-lepoll* [*THEN notE*], *assumption*+)
apply (*erule eqpoll-imp-lepoll*)
done

lemma *Card-nat*: *Card*(*nat*)
apply (*unfold Card-def cardinal-def*)
apply (*subst Least-equality*)
apply (*rule eqpoll-refl*)
apply (*rule Ord-nat*)
apply (*erule ltE*)
apply (*simp-all add*: *eqpoll-iff lt-not-lepoll ltI*)
done

lemma *nat-le-cardinal*: *nat le i* ==> *nat le* $|i|$
apply (*rule Card-nat* [*THEN Card-cardinal-eq*, *THEN subst*])
apply (*erule cardinal-mono*)
done

22.5 Towards Cardinal Arithmetic

lemma *cons-lepoll-cong*:
  [| $A \lesssim B$; b ~: B |] ==> *cons*(a,A) $\lesssim$ *cons*(b,B)
apply (*unfold lepoll-def*, *safe*)
apply (*rule-tac x = lam y*: *cons* (a,A) . *if y=a then b else f‘y* **in** *exI*)
apply (*rule-tac d = %z. if z*:*B then converse* (f) *‘z else a* **in** *lam-injective*)
apply (*safe elim*!: *consE′*)
  apply *simp-all*
apply (*blast intro*: *inj-is-fun* [*THEN apply-type*])+
done

lemma *cons-eqpoll-cong*:
  [| $A \approx B$; a ~: A; b ~: B |] ==> *cons*(a,A) $\approx$ *cons*(b,B)
by (*simp add*: *eqpoll-iff cons-lepoll-cong*)

lemma *cons-lepoll-cons-iff*:
  [| *a* ~: *A*; *b* ~: *B* |] ==> *cons*(*a*,*A*) $\lesssim$ *cons*(*b*,*B*) <-> *A* $\lesssim$ *B*
by (*blast intro*: *cons-lepoll-cong cons-lepoll-consD*)

lemma *cons-eqpoll-cons-iff*:
  [| *a* ~: *A*; *b* ~: *B* |] ==> *cons*(*a*,*A*) ≈ *cons*(*b*,*B*) <-> *A* ≈ *B*
by (*blast intro*: *cons-eqpoll-cong cons-eqpoll-consD*)

lemma *singleton-eqpoll-1*: {*a*} ≈ *1*
apply (*unfold succ-def*)
apply (*blast intro*!: *eqpoll-refl* [*THEN cons-eqpoll-cong*])
done

lemma *cardinal-singleton*: |{*a*}| = *1*
apply (*rule singleton-eqpoll-1* [*THEN cardinal-cong*, *THEN trans*])
apply (*simp* (*no-asm*) *add*: *nat-into-Card* [*THEN Card-cardinal-eq*])
done

lemma *not-0-is-lepoll-1*: *A* ~= *0* ==> *1* $\lesssim$ *A*
apply (*erule not-emptyE*)
apply (*rule-tac a* = *cons* (*x*, *A*−{*x*}) **in** *subst*)
apply (*rule-tac* [*2*] *a* = *cons*(*0*,*0*) **and** *P*= %*y*. *y* $\lesssim$ *cons* (*x*, *A*−{*x*}) **in** *subst*)
prefer *3* **apply** (*blast intro*: *cons-lepoll-cong subset-imp-lepoll*, *auto*)
done

lemma *succ-eqpoll-cong*: *A* ≈ *B* ==> *succ*(*A*) ≈ *succ*(*B*)
apply (*unfold succ-def*)
apply (*simp add*: *cons-eqpoll-cong mem-not-refl*)
done

lemma *sum-eqpoll-cong*: [| *A* ≈ *C*; *B* ≈ *D* |] ==> *A*+*B* ≈ *C*+*D*
apply (*unfold eqpoll-def*)
apply (*blast intro*!: *sum-bij*)
done

lemma *prod-eqpoll-cong*:
  [| *A* ≈ *C*; *B* ≈ *D* |] ==> *A*∗*B* ≈ *C*∗*D*
apply (*unfold eqpoll-def*)
apply (*blast intro*!: *prod-bij*)
done

lemma *inj-disjoint-eqpoll*:
  [| *f*: *inj*(*A*,*B*); *A Int B* = *0* |] ==> *A Un* (*B* − *range*(*f*)) ≈ *B*
apply (*unfold eqpoll-def*)
apply (*rule exI*)

apply (*rule-tac c* = %*x*. *if x*:*A then f*'*x else x*
  and *d* = %*y*. *if y*: *range* (*f*) *then converse* (*f*) '*y else y*
 in *lam-bijective*)
apply (*blast intro*!: *if-type inj-is-fun* [*THEN apply-type*])
apply (*simp* (*no-asm-simp*) *add*: *inj-converse-fun* [*THEN apply-funtype*])
apply (*safe elim*!: *UnE*′)
 apply (*simp-all add*: *inj-is-fun* [*THEN apply-rangeI*])
apply (*blast intro*: *inj-converse-fun* [*THEN apply-type*])+
done

22.6 Lemmas by Krzysztof Grabczewski

lemma *Diff-sing-lepoll*:
 [| *a*:*A*; $A \lesssim succ(n)$ |] ==> $A - \{a\} \lesssim n$
apply (*unfold succ-def*)
apply (*rule cons-lepoll-consD*)
apply (*rule-tac* [*3*] *mem-not-refl*)
apply (*erule cons-Diff* [*THEN ssubst*], *safe*)
done

lemma *lepoll-Diff-sing*:
 [| $succ(n) \lesssim A$ |] ==> $n \lesssim A - \{a\}$
apply (*unfold succ-def*)
apply (*rule cons-lepoll-consD*)
apply (*rule-tac* [*2*] *mem-not-refl*)
prefer *2* **apply** *blast*
apply (*blast intro*: *subset-imp-lepoll* [*THEN* [*2*] *lepoll-trans*])
done

lemma *Diff-sing-eqpoll*: [| *a*:*A*; $A \approx succ(n)$ |] ==> $A - \{a\} \approx n$
by (*blast intro*!: *eqpollI*
  elim!: *eqpollE*
  intro: *Diff-sing-lepoll lepoll-Diff-sing*)

lemma *lepoll-1-is-sing*: [| $A \lesssim 1$; *a*:*A* |] ==> $A = \{a\}$
apply (*frule Diff-sing-lepoll*, *assumption*)
apply (*drule lepoll-0-is-0*)
apply (*blast elim*: *equalityE*)
done

lemma *Un-lepoll-sum*: *A Un B* $\lesssim$ *A*+*B*
apply (*unfold lepoll-def*)
apply (*rule-tac x* = *lam x*: *A Un B*. *if x*:*A then Inl* (*x*) *else Inr* (*x*) **in** *exI*)
apply (*rule-tac d* = %*z*. *snd* (*z*) **in** *lam-injective*)
apply *force*
apply (*simp add*: *Inl-def Inr-def*)
done

lemma *well-ord-Un*:
  [| *well-ord*(*X*,*R*); *well-ord*(*Y*,*S*) |] ==> *EX T*. *well-ord*(*X Un Y*, *T*)
by (*erule well-ord-radd* [*THEN Un-lepoll-sum* [*THEN lepoll-well-ord*]],
  assumption)

lemma *disj-Un-eqpoll-sum*: *A Int B* = *0* ==> *A Un B* $\approx$ *A* + *B*
apply (*unfold eqpoll-def*)
apply (*rule-tac x* = *lam a*:*A Un B*. *if a*:*A then Inl* (*a*) *else Inr* (*a*) **in** *exI*)
apply (*rule-tac d* = %*z*. *case* (%*x*. *x*, %*x*. *x*, *z*) **in** *lam-bijective*)
apply *auto*
done

22.7 Finite and infinite sets

lemma *Finite-0* [*simp*]: *Finite*(*0*)
apply (*unfold Finite-def*)
apply (*blast intro*!: *eqpoll-refl nat-0I*)
done

lemma *lepoll-nat-imp-Finite*: [| *A* $\lesssim$ *n*; *n*:*nat* |] ==> *Finite*(*A*)
apply (*unfold Finite-def*)
apply (*erule rev-mp*)
apply (*erule nat-induct*)
apply (*blast dest*!: *lepoll-0-is-0 intro*!: *eqpoll-refl nat-0I*)
apply (*blast dest*!: *lepoll-succ-disj*)
done

lemma *lesspoll-nat-is-Finite*:
  A $\prec$ *nat* ==> *Finite*(*A*)
apply (*unfold Finite-def*)
apply (*blast dest*: *ltD lesspoll-cardinal-lt*
    lesspoll-imp-eqpoll [*THEN eqpoll-sym*])
done

lemma *lepoll-Finite*:
  [| *Y* $\lesssim$ *X*; *Finite*(*X*) |] ==> *Finite*(*Y*)
apply (*unfold Finite-def*)
apply (*blast elim*!: *eqpollE*
    intro: *lepoll-trans* [*THEN lepoll-nat-imp-Finite*
      [*unfolded Finite-def*]])
done

lemmas *subset-Finite* = *subset-imp-lepoll* [*THEN lepoll-Finite*, *standard*]

lemma *Finite-Int*: *Finite*(*A*) | *Finite*(*B*) ==> *Finite*(*A Int B*)
by (*blast intro*: *subset-Finite*)

lemmas *Finite-Diff* = *Diff-subset* [*THEN subset-Finite*, *standard*]

lemma *Finite-cons*: *Finite*(*x*) ==> *Finite*(*cons*(*y*,*x*))
apply (*unfold Finite-def*)
apply (*case-tac y*:*x*)
apply (*simp add*: *cons-absorb*)
apply (*erule bexE*)
apply (*rule bexI*)
apply (*erule-tac* [*2*] *nat-succI*)
apply (*simp* (*no-asm-simp*) *add*: *succ-def cons-eqpoll-cong mem-not-refl*)
done

lemma *Finite-succ*: *Finite*(*x*) ==> *Finite*(*succ*(*x*))
apply (*unfold succ-def*)
apply (*erule Finite-cons*)
done

lemma *Finite-cons-iff* [*iff*]: *Finite*(*cons*(*y*,*x*)) <−> *Finite*(*x*)
by (*blast intro*: *Finite-cons subset-Finite*)

lemma *Finite-succ-iff* [*iff*]: *Finite*(*succ*(*x*)) <−> *Finite*(*x*)
by (*simp add*: *succ-def*)

lemma *nat-le-infinite-Ord*:
[| *Ord*(*i*); ~ *Finite*(*i*) |] ==> *nat le i*
apply (*unfold Finite-def*)
apply (*erule Ord-nat* [*THEN* [*2*] *Ord-linear2*])
prefer *2* **apply** *assumption*
apply (*blast intro*!: *eqpoll-refl elim*!: *ltE*)
done

lemma *Finite-imp-well-ord*:
Finite(*A*) ==> *EX r*. *well-ord*(*A*,*r*)
apply (*unfold Finite-def eqpoll-def*)
apply (*blast intro*: *well-ord-rvimage bij-is-inj well-ord-Memrel nat-into-Ord*)
done

lemma *succ-lepoll-imp-not-empty*: *succ*(*x*) $\lesssim$ *y* ==> *y* ≠ *0*
by (*fast dest*!: *lepoll-0-is-0*)

lemma *eqpoll-succ-imp-not-empty*: *x* ≈ *succ*(*n*) ==> *x* ≠ *0*
by (*fast elim*!: *eqpoll-sym* [*THEN eqpoll-0-is-0*, *THEN succ-neq-0*])

lemma *Finite-Fin-lemma* [*rule-format*]:
n ∈ *nat* ==> ∀ *A*. (*A*≈*n* & *A* ⊆ *X*) −−> *A* ∈ *Fin*(*X*)
apply (*induct-tac n*)
apply (*rule allI*)
apply (*fast intro*!: *Fin.emptyI dest*!: *eqpoll-imp-lepoll* [*THEN lepoll-0-is-0*])
apply (*rule allI*)
apply (*rule impI*)

apply (*erule conjE*)
apply (*rule eqpoll-succ-imp-not-empty* [*THEN not-emptyE*], *assumption*)
apply (*frule Diff-sing-eqpoll*, *assumption*)
apply (*erule allE*)
apply (*erule impE*, *fast*)
apply (*drule subsetD*, *assumption*)
apply (*drule Fin.consI*, *assumption*)
apply (*simp add*: *cons-Diff*)
done

lemma *Finite-Fin*: [| *Finite*(*A*); $A \subseteq X$ |] ==> $A \in Fin(X)$
by (*unfold Finite-def*, *blast intro*: *Finite-Fin-lemma*)

lemma *eqpoll-imp-Finite-iff*: $A \approx B$ ==> *Finite*(*A*) <-> *Finite*(*B*)
apply (*unfold Finite-def*)
apply (*blast intro*: *eqpoll-trans eqpoll-sym*)
done

lemma *Fin-lemma* [*rule-format*]: *n*: *nat* ==> *ALL A*. $A \approx n$ --> *A* : *Fin*(*A*)
apply (*induct-tac n*)
apply (*simp add*: *eqpoll-0-iff*, *clarify*)
apply (*subgoal-tac EX u. u*:*A*)
apply (*erule exE*)
apply (*rule Diff-sing-eqpoll* [*THEN revcut-rl*])
prefer *2* **apply** *assumption*
apply *assumption*
apply (*rule-tac b* = *A* **in** *cons-Diff* [*THEN subst*], *assumption*)
apply (*rule Fin.consI*, *blast*)
apply (*blast intro*: *subset-consI* [*THEN Fin-mono*, *THEN subsetD*])

apply (*unfold eqpoll-def*)
apply (*blast intro*: *bij-converse-bij* [*THEN bij-is-fun*, *THEN apply-type*])
done

lemma *Finite-into-Fin*: *Finite*(*A*) ==> *A* : *Fin*(*A*)
apply (*unfold Finite-def*)
apply (*blast intro*: *Fin-lemma*)
done

lemma *Fin-into-Finite*: *A* : *Fin*(*U*) ==> *Finite*(*A*)
by (*fast intro*!: *Finite-0 Finite-cons elim*: *Fin-induct*)

lemma *Finite-Fin-iff*: *Finite*(*A*) <-> *A* : *Fin*(*A*)
by (*blast intro*: *Finite-into-Fin Fin-into-Finite*)

lemma *Finite-Un*: [| *Finite*(*A*); *Finite*(*B*) |] ==> *Finite*(*A Un B*)
by (*blast intro*!: *Fin-into-Finite Fin-UnI*
dest!: *Finite-into-Fin*
intro: *Un-upper1* [*THEN Fin-mono*, *THEN subsetD*]

Un-upper2 [*THEN Fin-mono*, *THEN subsetD*])

lemma *Finite-Un-iff* [*simp*]: *Finite*(*A Un B*) <−> (*Finite*(*A*) & *Finite*(*B*))
by (*blast intro*: *subset-Finite Finite-Un*)

The converse must hold too.

lemma *Finite-Union*: [| *ALL y*:*X*. *Finite*(*y*); *Finite*(*X*) |] ==> *Finite*(*Union*(*X*))
apply (*simp add*: *Finite-Fin-iff*)
apply (*rule Fin-UnionI*)
apply (*erule Fin-induct*, *simp*)
apply (*blast intro*: *Fin.consI Fin-mono* [*THEN* [*2*] *rev-subsetD*])
done

lemma *Finite-induct* [*case-names 0 cons*, *induct set*: *Finite*]:
[| *Finite*(*A*); *P*(*0*);
!! *x B*. [| *Finite*(*B*); *x* ~: *B*; *P*(*B*) |] ==> *P*(*cons*(*x*, *B*)) |]
==> *P*(*A*)
apply (*erule Finite-into-Fin* [*THEN Fin-induct*])
apply (*blast intro*: *Fin-into-Finite*)+
done

lemma *Diff-sing-Finite*: *Finite*(*A* − {*a*}) ==> *Finite*(*A*)
apply (*unfold Finite-def*)
apply (*case-tac a*:*A*)
apply (*subgoal-tac* [*2*] *A*−{*a*}=*A*, *auto*)
apply (*rule-tac x* = *succ* (*n*) **in** *bexI*)
apply (*subgoal-tac cons* (*a*, *A* − {*a*}) = *A* & *cons* (*n*, *n*) = *succ* (*n*))
apply (*drule-tac a* = *a* **and** *b* = *n* **in** *cons-eqpoll-cong*)
apply (*auto dest*: *mem-irrefl*)
done

lemma *Diff-Finite* [*rule-format*]: *Finite*(*B*) ==> *Finite*(*A*−*B*) −−> *Finite*(*A*)
apply (*erule Finite-induct*, *auto*)
apply (*case-tac x*:*A*)
apply (*subgoal-tac* [*2*] *A*−*cons* (*x*, *B*) = *A* − *B*)
apply (*subgoal-tac A* − *cons* (*x*, *B*) = (*A* − *B*) − {*x*}, *simp*)
apply (*drule Diff-sing-Finite*, *auto*)
done

lemma *Finite-RepFun*: *Finite*(*A*) ==> *Finite*(*RepFun*(*A*,*f*))
by (*erule Finite-induct*, *simp-all*)

lemma *Finite-RepFun-iff-lemma* [*rule-format*]:
[|*Finite*(*x*); !!*x y*. *f*(*x*)=*f*(*y*) ==> *x*=*y*|]
==> ∀ *A*. *x* = *RepFun*(*A*,*f*) −−> *Finite*(*A*)
apply (*erule Finite-induct*)

apply *clarify*
apply (*case-tac A=0*, *simp*)
apply (*blast del*: *allE*, *clarify*)
apply (*subgoal-tac* ∃ *z*∈*A*. *x* = *f*(*z*))
prefer *2* **apply** (*blast del*: *allE elim*: *equalityE*, *clarify*)
apply (*subgoal-tac B* = {*f*(*u*) . *u* ∈ *A* − {*z*}})
apply (*blast intro*: *Diff-sing-Finite*)
apply (*thin-tac* ∀ *A*. *?P*(*A*) −−> *Finite*(*A*))
apply (*rule equalityI*)
apply (*blast intro*: *elim*: *equalityE*)
apply (*blast intro*: *elim*: *equalityCE*)
done

I don't know why, but if the premise is expressed using meta-connectives then the simplifier cannot prove it automatically in conditional rewriting.

lemma *Finite-RepFun-iff*:
(∀ *x y*. *f*(*x*)=*f*(*y*) −−> *x*=*y*) ==> *Finite*(*RepFun*(*A*,*f*)) <−> *Finite*(*A*)
by (*blast intro*: *Finite-RepFun Finite-RepFun-iff-lemma* [*of* - *f*])

lemma *Finite-Pow*: *Finite*(*A*) ==> *Finite*(*Pow*(*A*))
apply (*erule Finite-induct*)
apply (*simp-all add*: *Pow-insert Finite-Un Finite-RepFun*)
done

lemma *Finite-Pow-imp-Finite*: *Finite*(*Pow*(*A*)) ==> *Finite*(*A*)
apply (*subgoal-tac Finite*({{*x*} . *x* ∈ *A*}))
apply (*simp add*: *Finite-RepFun-iff*)
apply (*blast intro*: *subset-Finite*)
done

lemma *Finite-Pow-iff* [*iff*]: *Finite*(*Pow*(*A*)) <−> *Finite*(*A*)
by (*blast intro*: *Finite-Pow Finite-Pow-imp-Finite*)

lemma *nat-wf-on-converse-Memrel*: *n*:*nat* ==> *wf*[*n*](*converse*(*Memrel*(*n*)))
apply (*erule nat-induct*)
apply (*blast intro*: *wf-onI*)
apply (*rule wf-onI*)
apply (*simp add*: *wf-on-def wf-def*)
apply (*case-tac x*:*Z*)

x:Z case

apply (*drule-tac x* = *x* **in** *bspec*, *assumption*)
apply (*blast elim*: *mem-irrefl mem-asym*)

other case

apply (*drule-tac x* = *Z* **in** *spec*, *blast*)
done

lemma *nat-well-ord-converse-Memrel*: *n*:*nat* ==> *well-ord*(*n*,*converse*(*Memrel*(*n*)))
apply (*frule Ord-nat* [*THEN Ord-in-Ord*, *THEN well-ord-Memrel*])
apply (*unfold well-ord-def*)
apply (*blast intro*!: *tot-ord-converse nat-wf-on-converse-Memrel*)
done

lemma *well-ord-converse*:
  [|*well-ord*(*A*,*r*);
    well-ord(*ordertype*(*A*,*r*), *converse*(*Memrel*(*ordertype*(*A*, *r*)))) |]
  ==> *well-ord*(*A*,*converse*(*r*))
apply (*rule well-ord-Int-iff* [*THEN iffD1*])
apply (*frule ordermap-bij* [*THEN bij-is-inj*, *THEN well-ord-rvimage*], *assumption*)
apply (*simp add*: *rvimage-converse converse-Int converse-prod*
    ordertype-ord-iso [*THEN ord-iso-rvimage-eq*])
done

lemma *ordertype-eq-n*:
  [| *well-ord*(*A*,*r*); *A* ≈ *n*; *n*:*nat* |] ==> *ordertype*(*A*,*r*)=*n*
apply (*rule Ord-ordertype* [*THEN Ord-nat-eqpoll-iff*, *THEN iffD1*], *assumption*+)
apply (*rule eqpoll-trans*)
 prefer *2* **apply** *assumption*
apply (*unfold eqpoll-def*)
apply (*blast intro*!: *ordermap-bij* [*THEN bij-converse-bij*])
done

lemma *Finite-well-ord-converse*:
  [| *Finite*(*A*); *well-ord*(*A*,*r*) |] ==> *well-ord*(*A*,*converse*(*r*))
apply (*unfold Finite-def*)
apply (*rule well-ord-converse*, *assumption*)
apply (*blast dest*: *ordertype-eq-n intro*!: *nat-well-ord-converse-Memrel*)
done

lemma *nat-into-Finite*: *n*:*nat* ==> *Finite*(*n*)
apply (*unfold Finite-def*)
apply (*fast intro*!: *eqpoll-refl*)
done

lemma *nat-not-Finite*: ~*Finite*(*nat*)
apply (*unfold Finite-def*, *clarify*)
apply (*drule eqpoll-imp-lepoll* [*THEN lepoll-cardinal-le*], *simp*)
apply (*insert Card-nat*)
apply (*simp add*: *Card-def*)
apply (*drule le-imp-subset*)
apply (*blast elim*: *mem-irrefl*)
done

ML
⟪
val Least-def = thm Least-def;
val eqpoll-def = thm eqpoll-def;
val lepoll-def = thm lepoll-def;
val lesspoll-def = thm lesspoll-def;
val cardinal-def = thm cardinal-def;
val Finite-def = thm Finite-def;
val Card-def = thm Card-def;
val eq-imp-not-mem = thm eq-imp-not-mem;
val decomp-bnd-mono = thm decomp-bnd-mono;
val Banach-last-equation = thm Banach-last-equation;
val decomposition = thm decomposition;
val schroeder-bernstein = thm schroeder-bernstein;
val bij-imp-eqpoll = thm bij-imp-eqpoll;
val eqpoll-refl = thm eqpoll-refl;
val eqpoll-sym = thm eqpoll-sym;
val eqpoll-trans = thm eqpoll-trans;
val subset-imp-lepoll = thm subset-imp-lepoll;
val lepoll-refl = thm lepoll-refl;
val le-imp-lepoll = thm le-imp-lepoll;
val eqpoll-imp-lepoll = thm eqpoll-imp-lepoll;
val lepoll-trans = thm lepoll-trans;
val eqpollI = thm eqpollI;
val eqpollE = thm eqpollE;
val eqpoll-iff = thm eqpoll-iff;
val lepoll-0-is-0 = thm lepoll-0-is-0;
val empty-lepollI = thm empty-lepollI;
val lepoll-0-iff = thm lepoll-0-iff;
val Un-lepoll-Un = thm Un-lepoll-Un;
val eqpoll-0-is-0 = thm eqpoll-0-is-0;
val eqpoll-0-iff = thm eqpoll-0-iff;
val eqpoll-disjoint-Un = thm eqpoll-disjoint-Un;
val lesspoll-not-refl = thm lesspoll-not-refl;
val lesspoll-irrefl = thm lesspoll-irrefl;
val lesspoll-imp-lepoll = thm lesspoll-imp-lepoll;
val lepoll-well-ord = thm lepoll-well-ord;
val lepoll-iff-leqpoll = thm lepoll-iff-leqpoll;
val inj-not-surj-succ = thm inj-not-surj-succ;
val lesspoll-trans = thm lesspoll-trans;
val lesspoll-trans1 = thm lesspoll-trans1;
val lesspoll-trans2 = thm lesspoll-trans2;
val Least-equality = thm Least-equality;
val LeastI = thm LeastI;
val Least-le = thm Least-le;
val less-LeastE = thm less-LeastE;
val LeastI2 = thm LeastI2;
val Least-0 = thm Least-0;

val Ord-Least = thm Ord-Least;
val Least-cong = thm Least-cong;
val cardinal-cong = thm cardinal-cong;
val well-ord-cardinal-eqpoll = thm well-ord-cardinal-eqpoll;
val Ord-cardinal-eqpoll = thm Ord-cardinal-eqpoll;
val well-ord-cardinal-eqE = thm well-ord-cardinal-eqE;
val well-ord-cardinal-eqpoll-iff = thm well-ord-cardinal-eqpoll-iff;
val Ord-cardinal-le = thm Ord-cardinal-le;
val Card-cardinal-eq = thm Card-cardinal-eq;
val CardI = thm CardI;
val Card-is-Ord = thm Card-is-Ord;
val Card-cardinal-le = thm Card-cardinal-le;
val Ord-cardinal = thm Ord-cardinal;
val Card-iff-initial = thm Card-iff-initial;
val lt-Card-imp-lesspoll = thm lt-Card-imp-lesspoll;
val Card-0 = thm Card-0;
val Card-Un = thm Card-Un;
val Card-cardinal = thm Card-cardinal;
val cardinal-mono = thm cardinal-mono;
val cardinal-lt-imp-lt = thm cardinal-lt-imp-lt;
val Card-lt-imp-lt = thm Card-lt-imp-lt;
val Card-lt-iff = thm Card-lt-iff;
val Card-le-iff = thm Card-le-iff;
val well-ord-lepoll-imp-Card-le = thm well-ord-lepoll-imp-Card-le;
val lepoll-cardinal-le = thm lepoll-cardinal-le;
val lepoll-Ord-imp-eqpoll = thm lepoll-Ord-imp-eqpoll;
val lesspoll-imp-eqpoll = thm lesspoll-imp-eqpoll;
val cardinal-subset-Ord = thm cardinal-subset-Ord;
val cons-lepoll-consD = thm cons-lepoll-consD;
val cons-eqpoll-consD = thm cons-eqpoll-consD;
val succ-lepoll-succD = thm succ-lepoll-succD;
val nat-lepoll-imp-le = thm nat-lepoll-imp-le;
val nat-eqpoll-iff = thm nat-eqpoll-iff;
val nat-into-Card = thm nat-into-Card;
val cardinal-0 = thm cardinal-0;
val cardinal-1 = thm cardinal-1;
val succ-lepoll-natE = thm succ-lepoll-natE;
val n-lesspoll-nat = thm n-lesspoll-nat;
val nat-lepoll-imp-ex-eqpoll-n = thm nat-lepoll-imp-ex-eqpoll-n;
val lepoll-imp-lesspoll-succ = thm lepoll-imp-lesspoll-succ;
val lesspoll-succ-imp-lepoll = thm lesspoll-succ-imp-lepoll;
val lesspoll-succ-iff = thm lesspoll-succ-iff;
val lepoll-succ-disj = thm lepoll-succ-disj;
val lesspoll-cardinal-lt = thm lesspoll-cardinal-lt;
val lt-not-lepoll = thm lt-not-lepoll;
val Ord-nat-eqpoll-iff = thm Ord-nat-eqpoll-iff;
val Card-nat = thm Card-nat;
val nat-le-cardinal = thm nat-le-cardinal;
val cons-lepoll-cong = thm cons-lepoll-cong;

val cons-eqpoll-cong = thm cons-eqpoll-cong;
val cons-lepoll-cons-iff = thm cons-lepoll-cons-iff;
val cons-eqpoll-cons-iff = thm cons-eqpoll-cons-iff;
val singleton-eqpoll-1 = thm singleton-eqpoll-1;
val cardinal-singleton = thm cardinal-singleton;
val not-0-is-lepoll-1 = thm not-0-is-lepoll-1;
val succ-eqpoll-cong = thm succ-eqpoll-cong;
val sum-eqpoll-cong = thm sum-eqpoll-cong;
val prod-eqpoll-cong = thm prod-eqpoll-cong;
val inj-disjoint-eqpoll = thm inj-disjoint-eqpoll;
val Diff-sing-lepoll = thm Diff-sing-lepoll;
val lepoll-Diff-sing = thm lepoll-Diff-sing;
val Diff-sing-eqpoll = thm Diff-sing-eqpoll;
val lepoll-1-is-sing = thm lepoll-1-is-sing;
val Un-lepoll-sum = thm Un-lepoll-sum;
val well-ord-Un = thm well-ord-Un;
val disj-Un-eqpoll-sum = thm disj-Un-eqpoll-sum;
val Finite-0 = thm Finite-0;
val lepoll-nat-imp-Finite = thm lepoll-nat-imp-Finite;
val lesspoll-nat-is-Finite = thm lesspoll-nat-is-Finite;
val lepoll-Finite = thm lepoll-Finite;
val subset-Finite = thm subset-Finite;
val Finite-Diff = thm Finite-Diff;
val Finite-cons = thm Finite-cons;
val Finite-succ = thm Finite-succ;
val nat-le-infinite-Ord = thm nat-le-infinite-Ord;
val Finite-imp-well-ord = thm Finite-imp-well-ord;
val nat-wf-on-converse-Memrel = thm nat-wf-on-converse-Memrel;
val nat-well-ord-converse-Memrel = thm nat-well-ord-converse-Memrel;
val well-ord-converse = thm well-ord-converse;
val ordertype-eq-n = thm ordertype-eq-n;
val Finite-well-ord-converse = thm Finite-well-ord-converse;
val nat-into-Finite = thm nat-into-Finite;
⟫

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* ($\bigcup$*j*∈*i*. *Pow*(*Vfrom*(*A*,*j*)))
by (*subst Vfrom-def* [*THEN def-transrec*], *simp*)

23.1.1 Monotonicity

lemma *Vfrom-mono* [*rule-format*]:
A<=*B* ==> ∀ *j*. *i*<=*j* −−> *Vfrom*(*A*,*i*) <= *Vfrom*(*B*,*j*)
apply (*rule-tac a=i* **in** *eps-induct*)
apply (*rule impI* [*THEN allI*])
apply (*subst Vfrom* [*of A*])
apply (*subst Vfrom* [*of B*])
apply (*erule Un-mono*)
apply (*erule UN-mono*, *blast*)
done

lemma *VfromI*: [| *a* ∈ *Vfrom*(*A*,*j*); *j*<*i* |] ==> *a* ∈ *Vfrom*(*A*,*i*)
by (*blast dest*: *Vfrom-mono* [*OF subset-refl le-imp-subset* [*OF leI*]])

23.1.2 A fundamental equality: Vfrom does not require ordinals!

lemma *Vfrom-rank-subset1*: *Vfrom*(*A*,*x*) <= *Vfrom*(*A*,*rank*(*x*))
proof (*induct x rule*: *eps-induct*)
fix *x*
assume ∀ *y*∈*x*. *Vfrom*(*A*,*y*) ⊆ *Vfrom*(*A*,*rank*(*y*))
thus *Vfrom*(*A*, *x*) ⊆ *Vfrom*(*A*, *rank*(*x*))
by (*simp add*: *Vfrom* [*of - x*] *Vfrom* [*of - rank*(*x*)],
blast intro!: *rank-lt* [*THEN ltD*])

qed

lemma *Vfrom-rank-subset2*: *Vfrom*(*A*,*rank*(*x*)) <= *Vfrom*(*A*,*x*)
apply (*rule-tac a*=*x* **in** *eps-induct*)
apply (*subst Vfrom*)
apply (*subst Vfrom*, *rule subset-refl* [*THEN Un-mono*])
apply (*rule UN-least*)

expand *rank*(*x1*) = ($\bigcup$ *y*∈*x1*. *succ*(*rank*(*y*))) in assumptions

apply (*erule rank* [*THEN equalityD1*, *THEN subsetD*, *THEN UN-E*])
apply (*rule subset-trans*)
apply (*erule-tac* [*2*] *UN-upper*)
apply (*rule subset-refl* [*THEN Vfrom-mono*, *THEN subset-trans*, *THEN Pow-mono*])
apply (*erule ltI* [*THEN le-imp-subset*])
apply (*rule Ord-rank* [*THEN Ord-succ*])
apply (*erule bspec*, *assumption*)
done

lemma *Vfrom-rank-eq*: *Vfrom*(*A*,*rank*(*x*)) = *Vfrom*(*A*,*x*)
apply (*rule equalityI*)
apply (*rule Vfrom-rank-subset2*)
apply (*rule Vfrom-rank-subset1*)
done

23.2 Basic Closure Properties

lemma *zero-in-Vfrom*: *y*:*x* ==> *0* ∈ *Vfrom*(*A*,*x*)
by (*subst Vfrom*, *blast*)

lemma *i-subset-Vfrom*: *i* <= *Vfrom*(*A*,*i*)
apply (*rule-tac a*=*i* **in** *eps-induct*)
apply (*subst Vfrom*, *blast*)
done

lemma *A-subset-Vfrom*: *A* <= *Vfrom*(*A*,*i*)
apply (*subst Vfrom*)
apply (*rule Un-upper1*)
done

lemmas *A-into-Vfrom* = *A-subset-Vfrom* [*THEN subsetD*]

lemma *subset-mem-Vfrom*: *a* <= *Vfrom*(*A*,*i*) ==> *a* ∈ *Vfrom*(*A*,*succ*(*i*))
by (*subst Vfrom*, *blast*)

23.2.1 Finite sets and ordered pairs

lemma *singleton-in-Vfrom*: *a* ∈ *Vfrom*(*A*,*i*) ==> {*a*} ∈ *Vfrom*(*A*,*succ*(*i*))
by (*rule subset-mem-Vfrom*, *safe*)

lemma *doubleton-in-Vfrom*:

[| a ∈ Vfrom(A,i); b ∈ Vfrom(A,i) |] ==> {a,b} ∈ Vfrom(A,succ(i))
by (*rule subset-mem-Vfrom*, *safe*)

lemma *Pair-in-Vfrom*:
[| a ∈ Vfrom(A,i); b ∈ Vfrom(A,i) |] ==> <a,b> ∈ Vfrom(A,succ(succ(i)))
apply (*unfold Pair-def*)
apply (*blast intro*: *doubleton-in-Vfrom*)
done

lemma *succ-in-Vfrom*: a <= Vfrom(A,i) ==> succ(a) ∈ Vfrom(A,succ(succ(i)))
apply (*intro subset-mem-Vfrom succ-subsetI*, *assumption*)
apply (*erule subset-trans*)
apply (*rule Vfrom-mono* [*OF subset-refl subset-succI*])
done

23.3 0, Successor and Limit Equations for *Vfrom*

lemma *Vfrom-0*: Vfrom(A,0) = A
by (*subst Vfrom*, *blast*)

lemma *Vfrom-succ-lemma*: Ord(i) ==> Vfrom(A,succ(i)) = A Un Pow(Vfrom(A,i))
apply (*rule Vfrom* [*THEN trans*])
apply (*rule equalityI* [*THEN subst-context*,
OF - *succI1* [*THEN RepFunI*, *THEN Union-upper*]])
apply (*rule UN-least*)
apply (*rule subset-refl* [*THEN Vfrom-mono*, *THEN Pow-mono*])
apply (*erule ltI* [*THEN le-imp-subset*])
apply (*erule Ord-succ*)
done

lemma *Vfrom-succ*: Vfrom(A,succ(i)) = A Un Pow(Vfrom(A,i))
apply (*rule-tac x1 = succ (i)* **in** *Vfrom-rank-eq* [*THEN subst*])
apply (*rule-tac x1 = i* **in** *Vfrom-rank-eq* [*THEN subst*])
apply (*subst rank-succ*)
apply (*rule Ord-rank* [*THEN Vfrom-succ-lemma*])
done

lemma *Vfrom-Union*: y:X ==> Vfrom(A,Union(X)) = (⋃ y∈X. Vfrom(A,y))
apply (*subst Vfrom*)
apply (*rule equalityI*)

first inclusion

apply (*rule Un-least*)
apply (*rule A-subset-Vfrom* [*THEN subset-trans*])
apply (*rule UN-upper*, *assumption*)
apply (*rule UN-least*)
apply (*erule UnionE*)
apply (*rule subset-trans*)
apply (*erule-tac* [*2*] *UN-upper*,

subst Vfrom, *erule subset-trans* [*OF UN-upper Un-upper2*])

opposite inclusion

apply (*rule UN-least*)
apply (*subst Vfrom*, *blast*)
done

23.4 *Vfrom* applied to Limit Ordinals

lemma *Limit-Vfrom-eq*:
Limit(*i*) ==> *Vfrom*(*A*,*i*) = ($\bigcup$ *y*∈*i*. *Vfrom*(*A*,*y*))
apply (*rule Limit-has-0* [*THEN ltD*, *THEN Vfrom-Union*, *THEN subst*], *assumption*)
apply (*simp add*: *Limit-Union-eq*)
done

lemma *Limit-VfromE*:
[| *a* ∈ *Vfrom*(*A*,*i*); ~*R* ==> *Limit*(*i*);
!!*x*. [| *x*<*i*; *a* ∈ *Vfrom*(*A*,*x*) |] ==> *R*
|] ==> *R*
apply (*rule classical*)
apply (*rule Limit-Vfrom-eq* [*THEN equalityD1*, *THEN subsetD*, *THEN UN-E*])
prefer *2* **apply** *assumption*
apply *blast*
apply (*blast intro*: *ltI Limit-is-Ord*)
done

lemma *singleton-in-VLimit*:
[| *a* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> {*a*} ∈ *Vfrom*(*A*,*i*)
apply (*erule Limit-VfromE*, *assumption*)
apply (*erule singleton-in-Vfrom* [*THEN VfromI*])
apply (*blast intro*: *Limit-has-succ*)
done

lemmas *Vfrom-UnI1* =
Un-upper1 [*THEN subset-refl* [*THEN Vfrom-mono*, *THEN subsetD*], *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* ∈ *Vfrom*(*A*,*i*); *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> {*a*,*b*} ∈ *Vfrom*(*A*,*i*)
apply (*erule Limit-VfromE*, *assumption*)
apply (*erule Limit-VfromE*, *assumption*)
apply (*blast intro*: *VfromI* [*OF doubleton-in-Vfrom*]
Vfrom-UnI1 Vfrom-UnI2 Limit-has-succ Un-least-lt)
done

lemma *Pair-in-VLimit*:

[| *a* ∈ *Vfrom*(*A*,*i*); *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> <*a*,*b*> ∈ *Vfrom*(*A*,*i*)

Infer that a, b occur at ordinals x,xa ¡ i.

apply (*erule Limit-VfromE*, *assumption*)
apply (*erule Limit-VfromE*, *assumption*)

Infer that succ(succ(x Un xa)) ¡ i

apply (*blast intro*: *VfromI* [*OF Pair-in-Vfrom*]
Vfrom-UnI1 Vfrom-UnI2 Limit-has-succ Un-least-lt)
done

lemma *product-VLimit*: *Limit*(*i*) ==> *Vfrom*(*A*,*i*) ∗ *Vfrom*(*A*,*i*) <= *Vfrom*(*A*,*i*)
by (*blast intro*: *Pair-in-VLimit*)

lemmas *Sigma-subset-VLimit* =
subset-trans [*OF Sigma-mono product-VLimit*]

lemmas *nat-subset-VLimit* =
subset-trans [*OF nat-le-Limit* [*THEN le-imp-subset*] *i-subset-Vfrom*]

lemma *nat-into-VLimit*: [| *n*: *nat*; *Limit*(*i*) |] ==> *n* ∈ *Vfrom*(*A*,*i*)
by (*blast intro*: *nat-subset-VLimit* [*THEN subsetD*])

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*)
by (*blast intro*: *nat-into-VLimit*)

lemma *Inl-in-VLimit*:
[| *a* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> *Inl*(*a*) ∈ *Vfrom*(*A*,*i*)
apply (*unfold Inl-def*)
apply (*blast intro*: *zero-in-VLimit Pair-in-VLimit*)
done

lemma *Inr-in-VLimit*:
[| *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> *Inr*(*b*) ∈ *Vfrom*(*A*,*i*)
apply (*unfold Inr-def*)
apply (*blast intro*: *one-in-VLimit Pair-in-VLimit*)
done

lemma *sum-VLimit*: *Limit*(*i*) ==> *Vfrom*(*C*,*i*)+*Vfrom*(*C*,*i*) <= *Vfrom*(*C*,*i*)
by (*blast intro*!: *Inl-in-VLimit Inr-in-VLimit*)

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*))
apply (*rule-tac a*=*i* **in** *eps-induct*)
apply (*subst Vfrom*)
apply (*blast intro*!: *Transset-Union-family Transset-Un Transset-Pow*)
done

lemma *Transset-Vfrom-succ*:
  Transset(*A*) ==> *Vfrom*(*A*, *succ*(*i*)) = *Pow*(*Vfrom*(*A*,*i*))
apply (*rule Vfrom-succ* [*THEN trans*])
apply (*rule equalityI* [*OF - Un-upper2*])
apply (*rule Un-least* [*OF - subset-refl*])
apply (*rule A-subset-Vfrom* [*THEN subset-trans*])
apply (*erule Transset-Vfrom* [*THEN Transset-iff-Pow* [*THEN iffD1*]])
done

lemma *Transset-Pair-subset*: [| <*a*,*b*> <= *C*; *Transset*(*C*) |] ==> *a*: *C* & *b*: *C*
by (*unfold Pair-def Transset-def*, *blast*)

lemma *Transset-Pair-subset-VLimit*:
  [| <*a*,*b*> <= *Vfrom*(*A*,*i*); *Transset*(*A*); *Limit*(*i*) |]
  ==> <*a*,*b*> ∈ *Vfrom*(*A*,*i*)
apply (*erule Transset-Pair-subset* [*THEN conjE*])
apply (*erule Transset-Vfrom*)
apply (*blast intro*: *Pair-in-VLimit*)
done

lemma *Union-in-Vfrom*:
  [| *X* ∈ *Vfrom*(*A*,*j*); *Transset*(*A*) |] ==> *Union*(*X*) ∈ *Vfrom*(*A*, *succ*(*j*))
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*unfold Transset-def*, *blast*)
done

lemma *Union-in-VLimit*:
  [| *X* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*); *Transset*(*A*) |] ==> *Union*(*X*) ∈ *Vfrom*(*A*,*i*)
apply (*rule Limit-VfromE*, *assumption*+)
apply (*blast intro*: *Limit-has-succ VfromI Union-in-Vfrom*)
done

General theorem for membership in Vfrom(A,i) when i is a limit ordinal

lemma *in-VLimit*:
 [| *a* ∈ *Vfrom*(*A*,*i*); *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*);
  !!*x y j*. [| *j*<*i*; *1*:*j*; *x* ∈ *Vfrom*(*A*,*j*); *y* ∈ *Vfrom*(*A*,*j*) |]
    ==> *EX k*. *h*(*x*,*y*) ∈ *Vfrom*(*A*,*k*) & *k*<*i* |]
 ==> *h*(*a*,*b*) ∈ *Vfrom*(*A*,*i*)

Infer that a, b occur at ordinals x,xa ¡ i.

apply (*erule Limit-VfromE*, *assumption*)

apply (*erule Limit-VfromE*, *assumption*, *atomize*)
apply (*drule-tac x=a* **in** *spec*)
apply (*drule-tac x=b* **in** *spec*)
apply (*drule-tac x=x Un xa Un 2* **in** *spec*)
apply (*simp add*: *Un-least-lt-iff lt-Ord Vfrom-UnI1 Vfrom-UnI2*)
apply (*blast intro*: *Limit-has-0 Limit-has-succ VfromI*)
done

23.5.1 Products

lemma *prod-in-Vfrom*:
[| *a* ∈ *Vfrom*(*A*,*j*); *b* ∈ *Vfrom*(*A*,*j*); *Transset*(*A*) |]
==> *a*∗*b* ∈ *Vfrom*(*A*, *succ*(*succ*(*succ*(*j*))))
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*unfold Transset-def*)
apply (*blast intro*: *Pair-in-Vfrom*)
done

lemma *prod-in-VLimit*:
[| *a* ∈ *Vfrom*(*A*,*i*); *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*); *Transset*(*A*) |]
==> *a*∗*b* ∈ *Vfrom*(*A*,*i*)
apply (*erule in-VLimit*, *assumption*+)
apply (*blast intro*: *prod-in-Vfrom Limit-has-succ*)
done

23.5.2 Disjoint Sums, or Quine Ordered Pairs

lemma *sum-in-Vfrom*:
[| *a* ∈ *Vfrom*(*A*,*j*); *b* ∈ *Vfrom*(*A*,*j*); *Transset*(*A*); *1:j* |]
==> *a*+*b* ∈ *Vfrom*(*A*, *succ*(*succ*(*succ*(*j*))))
apply (*unfold sum-def*)
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*unfold Transset-def*)
apply (*blast intro*: *zero-in-Vfrom Pair-in-Vfrom i-subset-Vfrom* [*THEN subsetD*])
done

lemma *sum-in-VLimit*:
[| *a* ∈ *Vfrom*(*A*,*i*); *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*); *Transset*(*A*) |]
==> *a*+*b* ∈ *Vfrom*(*A*,*i*)
apply (*erule in-VLimit*, *assumption*+)
apply (*blast intro*: *sum-in-Vfrom Limit-has-succ*)
done

23.5.3 Function Space!

lemma *fun-in-Vfrom*:
[| *a* ∈ *Vfrom*(*A*,*j*); *b* ∈ *Vfrom*(*A*,*j*); *Transset*(*A*) |] ==>
a−>*b* ∈ *Vfrom*(*A*, *succ*(*succ*(*succ*(*succ*(*j*)))))

apply (*unfold Pi-def*)
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*rule Collect-subset* [*THEN subset-trans*])
apply (*subst Vfrom*)
apply (*rule subset-trans* [*THEN subset-trans*])
apply (*rule-tac* [*3*] *Un-upper2*)
apply (*rule-tac* [*2*] *succI1* [*THEN UN-upper*])
apply (*rule Pow-mono*)
apply (*unfold Transset-def*)
apply (*blast intro*: *Pair-in-Vfrom*)
done

lemma *fun-in-VLimit*:
  [| *a* ∈ *Vfrom*(*A*,*i*); *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*); *Transset*(*A*) |]
  ==> *a*−>*b* ∈ *Vfrom*(*A*,*i*)
apply (*erule in-VLimit*, *assumption*+)
apply (*blast intro*: *fun-in-Vfrom Limit-has-succ*)
done

lemma *Pow-in-Vfrom*:
  [| *a* ∈ *Vfrom*(*A*,*j*); *Transset*(*A*) |] ==> *Pow*(*a*) ∈ *Vfrom*(*A*, *succ*(*succ*(*j*)))
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*unfold Transset-def*)
apply (*subst Vfrom*, *blast*)
done

lemma *Pow-in-VLimit*:
  [| *a* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*); *Transset*(*A*) |] ==> *Pow*(*a*) ∈ *Vfrom*(*A*,*i*)
by (*blast elim*: *Limit-VfromE intro*: *Limit-has-succ Pow-in-Vfrom VfromI*)

23.6 The Set *Vset*(*i*)

lemma *Vset*: *Vset*(*i*) = (⋃*j*∈*i*. *Pow*(*Vset*(*j*)))
by (*subst Vfrom*, *blast*)

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*) ==> ∀ *b*. *b* ∈ *Vset*(*i*) −−> *rank*(*b*) < *i*
apply (*erule trans-induct*)
apply (*subst Vset*, *safe*)
apply (*subst rank*)
apply (*blast intro*: *ltI UN-succ-least-lt*)
done

lemma *VsetI-lemma* [*rule-format*]:

Ord(*i*) ==> ∀ *b*. *rank*(*b*) ∈ *i* −−> *b* ∈ *Vset*(*i*)
apply (*erule trans-induct*)
apply (*rule allI*)
apply (*subst Vset*)
apply (*blast intro*!: *rank-lt* [*THEN ltD*])
done

lemma *VsetI*: *rank*(*x*)<*i* ==> *x* ∈ *Vset*(*i*)
by (*blast intro*: *VsetI-lemma elim*: *ltE*)

Merely a lemma for the next result

lemma *Vset-Ord-rank-iff*: *Ord*(*i*) ==> *b* ∈ *Vset*(*i*) <−> *rank*(*b*) < *i*
by (*blast intro*: *VsetD VsetI*)

lemma *Vset-rank-iff* [*simp*]: *b* ∈ *Vset*(*a*) <−> *rank*(*b*) < *rank*(*a*)
apply (*rule Vfrom-rank-eq* [*THEN subst*])
apply (*rule Ord-rank* [*THEN Vset-Ord-rank-iff*])
done

This is rank(rank(a)) = rank(a)

declare *Ord-rank* [*THEN rank-of-Ord*, *simp*]

lemma *rank-Vset*: *Ord*(*i*) ==> *rank*(*Vset*(*i*)) = *i*
apply (*subst rank*)
apply (*rule equalityI*, *safe*)
apply (*blast intro*: *VsetD* [*THEN ltD*])
apply (*blast intro*: *VsetD* [*THEN ltD*] *Ord-trans*)
apply (*blast intro*: *i-subset-Vfrom* [*THEN subsetD*]
Ord-in-Ord [*THEN rank-of-Ord*, *THEN ssubst*])
done

lemma *Finite-Vset*: *i* ∈ *nat* ==> *Finite*(*Vset*(*i*))
apply (*erule nat-induct*)
apply (*simp add*: *Vfrom-0*)
apply (*simp add*: *Vset-succ*)
done

23.6.2 Reasoning about Sets in Terms of Their Elements' Ranks

lemma *arg-subset-Vset-rank*: *a* <= *Vset*(*rank*(*a*))
apply (*rule subsetI*)
apply (*erule rank-lt* [*THEN VsetI*])
done

lemma *Int-Vset-subset*:
[| !!*i*. *Ord*(*i*) ==> *a Int Vset*(*i*) <= *b* |] ==> *a* <= *b*
apply (*rule subset-trans*)
apply (*rule Int-greatest* [*OF subset-refl arg-subset-Vset-rank*])
apply (*blast intro*: *Ord-rank*)

done

23.6.3 Set Up an Environment for Simplification

lemma *rank-Inl*: *rank*(*a*) < *rank*(*Inl*(*a*))
apply (*unfold Inl-def*)
apply (*rule rank-pair2*)
done

lemma *rank-Inr*: *rank*(*a*) < *rank*(*Inr*(*a*))
apply (*unfold Inr-def*)
apply (*rule rank-pair2*)
done

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*))
apply (*unfold Vrec-def*)
apply (*subst transrec*, *simp*)
apply (*rule refl* [*THEN lam-cong*, *THEN subst-context*], *simp add*: *lt-def*)
done

This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-Vrec*:
  [| !!*x*. *h*(*x*)==*Vrec*(*x*,*H*) |] ==>
   h(*a*) = *H*(*a*, *lam x*: *Vset*(*rank*(*a*)). *h*(*x*))
apply *simp*
apply (*rule Vrec*)
done

NOT SUITABLE FOR REWRITING: recursive!

lemma *Vrecursor*:
   Vrecursor(*H*,*a*) = *H*(*lam x*:*Vset*(*rank*(*a*)). *Vrecursor*(*H*,*x*), *a*)
apply (*unfold Vrecursor-def*)
apply (*subst transrec*, *simp*)
apply (*rule refl* [*THEN lam-cong*, *THEN subst-context*], *simp add*: *lt-def*)
done

This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-Vrecursor*:
   h == *Vrecursor*(*H*) ==> *h*(*a*) = *H*(*lam x*: *Vset*(*rank*(*a*)). *h*(*x*), *a*)
apply *simp*
apply (*rule Vrecursor*)
done

23.7 The Datatype Universe: *univ*(*A*)

lemma *univ-mono*: *A*<=*B* ==> *univ*(*A*) <= *univ*(*B*)
apply (*unfold univ-def*)
apply (*erule Vfrom-mono*)
apply (*rule subset-refl*)
done

lemma *Transset-univ*: *Transset*(*A*) ==> *Transset*(*univ*(*A*))
apply (*unfold univ-def*)
apply (*erule Transset-Vfrom*)
done

23.7.1 The Set *univ*(*A*) as a Limit

lemma *univ-eq-UN*: *univ*(*A*) = ($\bigcup$*i*∈*nat*. *Vfrom*(*A*,*i*))
apply (*unfold univ-def*)
apply (*rule Limit-nat* [*THEN Limit-Vfrom-eq*])
done

lemma *subset-univ-eq-Int*: *c* <= *univ*(*A*) ==> *c* = ($\bigcup$*i*∈*nat*. *c Int Vfrom*(*A*,*i*))
apply (*rule subset-UN-iff-eq* [*THEN iffD1*])
apply (*erule univ-eq-UN* [*THEN subst*])
done

lemma *univ-Int-Vfrom-subset*:
  [| *a* <= *univ*(*X*);
    !!*i*. *i*:*nat* ==> *a Int Vfrom*(*X*,*i*) <= *b* |]
  ==> *a* <= *b*
apply (*subst subset-univ-eq-Int*, *assumption*)
apply (*rule UN-least*, *simp*)
done

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*
apply (*rule equalityI*)
apply (*rule univ-Int-Vfrom-subset*, *assumption*)
apply (*blast elim*: *equalityCE*)
apply (*rule univ-Int-Vfrom-subset*, *assumption*)
apply (*blast elim*: *equalityCE*)
done

23.8 Closure Properties for *univ*(*A*)

lemma *zero-in-univ*: *0* ∈ *univ*(*A*)
apply (*unfold univ-def*)
apply (*rule nat-0I* [*THEN zero-in-Vfrom*])
done

lemma *zero-subset-univ*: {*0*} <= *univ*(*A*)
by (*blast intro*: *zero-in-univ*)

lemma *A-subset-univ*: *A* <= *univ*(*A*)
apply (*unfold univ-def*)
apply (*rule A-subset-Vfrom*)
done

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*} ∈ *univ*(*A*)
apply (*unfold univ-def*)
apply (*blast intro*: *singleton-in-VLimit Limit-nat*)
done

lemma *doubleton-in-univ*:
  [| *a*: *univ*(*A*); *b*: *univ*(*A*) |] ==> {*a*,*b*} ∈ *univ*(*A*)
apply (*unfold univ-def*)
apply (*blast intro*: *doubleton-in-VLimit Limit-nat*)
done

lemma *Pair-in-univ*:
  [| *a*: *univ*(*A*); *b*: *univ*(*A*) |] ==> <*a*,*b*> ∈ *univ*(*A*)
apply (*unfold univ-def*)
apply (*blast intro*: *Pair-in-VLimit Limit-nat*)
done

lemma *Union-in-univ*:
  [| *X*: *univ*(*A*); *Transset*(*A*) |] ==> *Union*(*X*) ∈ *univ*(*A*)
apply (*unfold univ-def*)
apply (*blast intro*: *Union-in-VLimit Limit-nat*)
done

lemma *product-univ*: *univ*(*A*)∗*univ*(*A*) <= *univ*(*A*)
apply (*unfold univ-def*)
apply (*rule Limit-nat* [*THEN product-VLimit*])
done

23.8.2 The Natural Numbers

lemma *nat-subset-univ*: *nat* <= *univ*(*A*)
apply (*unfold univ-def*)
apply (*rule i-subset-Vfrom*)
done

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*)
apply (*unfold univ-def*)
apply (*rule Limit-nat* [*THEN one-in-VLimit*])
done

unused!

lemma *two-in-univ*: *2* ∈ *univ*(*A*)
by (*blast intro*: *nat-into-univ*)

lemma *bool-subset-univ*: *bool* <= *univ*(*A*)
apply (*unfold bool-def*)
apply (*blast intro*!: *zero-in-univ one-in-univ*)
done

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*)
apply (*unfold univ-def*)
apply (*erule Inl-in-VLimit* [*OF - Limit-nat*])
done

lemma *Inr-in-univ*: *b*: *univ*(*A*) ==> *Inr*(*b*) ∈ *univ*(*A*)
apply (*unfold univ-def*)
apply (*erule Inr-in-VLimit* [*OF - Limit-nat*])
done

lemma *sum-univ*: *univ*(*C*)+*univ*(*C*) <= *univ*(*C*)
apply (*unfold univ-def*)
apply (*rule Limit-nat* [*THEN sum-VLimit*])
done

lemmas *sum-subset-univ* = *subset-trans* [*OF sum-mono sum-univ*]

lemma *Sigma-subset-univ*:
[|*A* ⊆ *univ*(*D*); ⋀*x*. *x* ∈ *A* ⟹ *B*(*x*) ⊆ *univ*(*D*)|] ==> *Sigma*(*A*,*B*) ⊆ *univ*(*D*)
apply (*simp add*: *univ-def*)
apply (*blast intro*: *Sigma-subset-VLimit del*: *subsetI*)
done

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*
apply (*erule Fin-induct*)

apply (*blast dest*!: *Limit-has-0*, *safe*)
apply (*erule Limit-VfromE*, *assumption*)
apply (*blast intro*!: *Un-least-lt intro*: *Vfrom-UnI1 Vfrom-UnI2*)
done

lemma *Fin-VLimit*: *Limit*(*i*) ==> *Fin*(*Vfrom*(*A*,*i*)) <= *Vfrom*(*A*,*i*)
apply (*rule subsetI*)
apply (*drule Fin-Vfrom-lemma*, *safe*)
apply (*rule Vfrom* [*THEN ssubst*])
apply (*blast dest*!: *ltD*)
done

lemmas *Fin-subset-VLimit* = *subset-trans* [*OF Fin-mono Fin-VLimit*]

lemma *Fin-univ*: *Fin*(*univ*(*A*)) <= *univ*(*A*)
apply (*unfold univ-def*)
apply (*rule Limit-nat* [*THEN Fin-VLimit*])
done

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*)
apply (*erule nat-fun-subset-Fin* [*THEN subset-trans*])
apply (*blast del*: *subsetI*
  intro: *subset-refl Fin-subset-VLimit Sigma-subset-VLimit nat-subset-VLimit*)
done

lemmas *nat-fun-subset-VLimit* = *subset-trans* [*OF Pi-mono nat-fun-VLimit*]

lemma *nat-fun-univ*: *n*: *nat* ==> *n* −> *univ*(*A*) <= *univ*(*A*)
apply (*unfold univ-def*)
apply (*erule nat-fun-VLimit* [*OF - Limit-nat*])
done

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*)
apply (*rule FiniteFun.dom-subset* [*THEN subset-trans*])
apply (*blast del*: *subsetI*
    intro: *Fin-subset-VLimit Sigma-subset-VLimit subset-refl*)
done

lemma *FiniteFun-univ1*: *univ*(*A*) −||> *univ*(*A*) <= *univ*(*A*)
apply (*unfold univ-def*)
apply (*rule Limit-nat* [*THEN FiniteFun-VLimit1*])

done

Version for a fixed domain

lemma *FiniteFun-VLimit*:
[| *W* <= *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> *W* −||> *Vfrom*(*A*,*i*) <= *Vfrom*(*A*,*i*)
apply (*rule subset-trans*)
apply (*erule FiniteFun-mono* [*OF - subset-refl*])
apply (*erule FiniteFun-VLimit1*)
done

lemma *FiniteFun-univ*:
W <= *univ*(*A*) ==> *W* −||> *univ*(*A*) <= *univ*(*A*)
apply (*unfold univ-def*)
apply (*erule FiniteFun-VLimit* [*OF - Limit-nat*])
done

lemma *FiniteFun-in-univ*:
[| *f*: *W* −||> *univ*(*A*); *W* <= *univ*(*A*) |] ==> *f* ∈ *univ*(*A*)
by (*erule FiniteFun-univ* [*THEN subsetD*], *assumption*)

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*)
by (*drule Transset-Vfrom-succ* [*THEN equalityD1*, *THEN subsetD*, *THEN PowD*],

assumption, *fast*)

This weaker version says a, b exist at the same level

lemmas *Vfrom-doubleton-D* = *Transset-Vfrom* [*THEN Transset-doubleton-D*, *standard*]

lemma *Pair-in-Vfrom-D*:
[| <*a*,*b*> ∈ *Vfrom*(*X*,*succ*(*i*)); *Transset*(*X*) |]
==> *a* ∈ *Vfrom*(*X*,*i*) & *b* ∈ *Vfrom*(*X*,*i*)
apply (*unfold Pair-def*)
apply (*blast dest*!: *doubleton-in-Vfrom-D Vfrom-doubleton-D*)
done

lemma *product-Int-Vfrom-subset*:
  Transset(*X*) ==>
  (*a*∗*b*) *Int Vfrom*(*X*, *succ*(*i*)) <= (*a Int Vfrom*(*X*,*i*)) ∗ (*b Int Vfrom*(*X*,*i*))
by (*blast dest*!: *Pair-in-Vfrom-D*)

ML
⟪
val rank-ss = @{*simpset*} *addsimps* [@{*thm VsetI*}]
  addsimps @{*thms rank-rls*} @ (@{*thms rank-rls*} *RLN* (*2*, [@{*thm lt-trans*}]));
⟫

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*
apply (*simp add*: *sum-def Un-subset-iff*)
apply (*blast dest*: *Transset-includes-range*)
done

lemma *Transset-sum-Int-subset*:
  Transset(*C*) ==> (*A*+*B*) *Int C* <= (*A Int C*) + (*B Int C*)

apply (*simp add*: *sum-def Int-Un-distrib2*)
apply (*blast dest*: *Transset-Pair-D*)
done

24.2 Introduction and Elimination Rules

lemma *qunivI*: *X* <= *univ*(*eclose*(*A*)) ==> *X* : *quniv*(*A*)
by (*simp add*: *quniv-def*)

lemma *qunivD*: *X* : *quniv*(*A*) ==> *X* <= *univ*(*eclose*(*A*))
by (*simp add*: *quniv-def*)

lemma *quniv-mono*: *A*<=*B* ==> *quniv*(*A*) <= *quniv*(*B*)
apply (*unfold quniv-def*)
apply (*erule eclose-mono* [*THEN univ-mono*, *THEN Pow-mono*])
done

24.3 Closure Properties

lemma *univ-eclose-subset-quniv*: *univ*(*eclose*(*A*)) <= *quniv*(*A*)
apply (*simp add*: *quniv-def Transset-iff-Pow* [*symmetric*])
apply (*rule Transset-eclose* [*THEN Transset-univ*])
done

lemma *univ-subset-quniv*: *univ*(*A*) <= *quniv*(*A*)
apply (*rule arg-subset-eclose* [*THEN univ-mono*, *THEN subset-trans*])
apply (*rule univ-eclose-subset-quniv*)
done

lemmas *univ-into-quniv* = *univ-subset-quniv* [*THEN subsetD*, *standard*]

lemma *Pow-univ-subset-quniv*: *Pow*(*univ*(*A*)) <= *quniv*(*A*)
apply (*unfold quniv-def*)
apply (*rule arg-subset-eclose* [*THEN univ-mono*, *THEN Pow-mono*])
done

lemmas *univ-subset-into-quniv* =
PowI [*THEN Pow-univ-subset-quniv* [*THEN subsetD*], *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)
by (*simp add*: *QPair-def sum-subset-univ*)

24.4 Quine Disjoint Sum

lemma *QInl-subset-univ*: a <= univ(A) ==> QInl(a) <= univ(A)
apply (*unfold QInl-def*)
apply (*erule empty-subsetI* [*THEN QPair-subset-univ*])
done

lemmas *naturals-subset-nat* =
Ord-nat [*THEN Ord-is-Transset*, *unfolded Transset-def*, *THEN bspec*, *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)
apply (*unfold QInr-def*)
apply (*erule nat-1I* [*THEN naturals-subset-univ*, *THEN QPair-subset-univ*])
done

24.5 Closure for Quine-Inspired Products and Sums

lemma *QPair-in-quniv*:
[| a: quniv(A); b: quniv(A) |] ==> <a;b> : quniv(A)
by (*simp add*: *quniv-def QPair-def sum-subset-univ*)

lemma *QSigma-quniv*: quniv(A) <*> quniv(A) <= quniv(A)
by (*blast intro*: *QPair-in-quniv*)

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)
apply (*unfold quniv-def QPair-def*)
apply (*rule Transset-includes-summands* [*THEN conjE*])
apply (*rule Transset-eclose* [*THEN Transset-univ*])
apply (*erule PowD*, *blast*)
done

lemmas *quniv-QPair-E* = *quniv-QPair-D* [*THEN conjE*, *standard*]

lemma *quniv-QPair-iff*: <a;b> : quniv(A) <-> a: quniv(A) & b: quniv(A)
by (*blast intro*: *QPair-in-quniv dest*: *quniv-QPair-D*)

24.6 Quine Disjoint Sum

lemma *QInl-in-quniv*: *a*: *quniv*(*A*) ==> *QInl*(*a*) : *quniv*(*A*)
by (*simp add*: *QInl-def zero-in-quniv QPair-in-quniv*)

lemma *QInr-in-quniv*: *b*: *quniv*(*A*) ==> *QInr*(*b*) : *quniv*(*A*)
by (*simp add*: *QInr-def one-in-quniv QPair-in-quniv*)

lemma *qsum-quniv*: *quniv*(*C*) <+> *quniv*(*C*) <= *quniv*(*C*)
by (*blast intro*: *QInl-in-quniv QInr-in-quniv*)

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*)>
by (*simp add*: *QPair-def sum-def Int-Un-distrib2 Un-mono*
product-Int-Vfrom-subset [*THEN subset-trans*]
Sigma-mono [*OF Int-lower1 subset-refl*])

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*)>
apply (*unfold QPair-def*)
apply (*erule Transset-Vfrom* [*THEN Transset-sum-Int-subset*])
done

lemmas *QPair-Int-Vset-subset-trans* =
subset-trans [*OF Transset-0* [*THEN QPair-Int-Vfrom-subset*] *QPair-mono*]

lemma *QPair-Int-Vset-subset-UN*:
Ord(*i*) ==> <*a*;*b*> *Int Vset*(*i*) <= ($\bigcup$*j*∈*i*. <*a Int Vset*(*j*); *b Int Vset*(*j*)>)
apply (*erule Ord-cases*)

apply (*simp add*: *Vfrom-0*)

apply (*erule ssubst*)
apply (*rule Transset-0* [*THEN QPair-Int-Vfrom-succ-subset*, *THEN subset-trans*])
apply (*rule succI1* [*THEN UN-upper*])

apply (*simp del*: *UN-simps*
    add: *Limit-Vfrom-eq Int-UN-distrib UN-mono QPair-Int-Vset-subset-trans*)
done

end

25 Datatype-ZF: Datatype and CoDatatype Definitions

theory *Datatype-ZF*
imports *Inductive-ZF Univ QUniv*
uses *Tools/datatype-package.ML*
begin

ML ⟪
(∗*Typechecking rules for most datatypes involving univ*∗)
structure Data-Arg =
  struct
  val intrs =
      [@{*thm SigmaI*}, @{*thm InlI*}, @{*thm InrI*},
       @{*thm Pair-in-univ*}, @{*thm Inl-in-univ*}, @{*thm Inr-in-univ*},
        @{*thm zero-in-univ*}, @{*thm A-into-univ*}, @{*thm nat-into-univ*}, @{*thm UnCI*}];

  val elims = [*make-elim* @{*thm InlD*}, *make-elim* @{*thm InrD*},   (∗*for mutual recursion*∗)
            @{*thm SigmaE*}, @{*thm sumE*}];   (∗*allows* ∗ *and* + *in spec*∗)
  end;

structure Data-Package =
  Add-datatype-def-Fun
   (*structure Fp*=*Lfp and Pr*=*Standard-Prod and CP*=*Standard-CP*
    and Su=*Standard-Sum*
    and Ind-Package = *Ind-Package*
    and Datatype-Arg = *Data-Arg*
    val coind = *false*);

```
(*Typechecking rules for most codatatypes involving quniv*)
structure CoData-Arg =
  struct
  val intrs =
      [@{thm QSigmaI}, @{thm QInlI}, @{thm QInrI},
       @{thm QPair-in-quniv}, @{thm QInl-in-quniv}, @{thm QInr-in-quniv},
      @{thm zero-in-quniv}, @{thm A-into-quniv}, @{thm nat-into-quniv}, @{thm
UnCI}];

  val elims = [make-elim @{thm QInlD}, make-elim @{thm QInrD},   (*for mutual
recursion*)
               @{thm QSigmaE}, @{thm qsumE}];                    (*allows * and +
in spec*)
  end;

structure CoData-Package =
  Add-datatype-def-Fun
   (structure Fp=Gfp and Pr=Quine-Prod and CP=Quine-CP
    and Su=Quine-Sum
    and Ind-Package = CoInd-Package
    and Datatype-Arg = CoData-Arg
    val coind = true);



(*Simproc for freeness reasoning: compare datatype constructors for equality*)
structure DataFree =
struct
  val trace = Unsynchronized.ref false;

  fun mk-new ([],[]) = Const(True,FOLogic.oT)
    | mk-new (largs,rargs) =
        Balanced-Tree.make FOLogic.mk-conj
                 (map FOLogic.mk-eq (ListPair.zip (largs,rargs)));

 val datatype-ss = @{simpset};

 fun proc sg ss old =
   let val - =
         if !trace then writeln (data-free: OLD =  ^ Syntax.string-of-term-global sg
old)
         else ()
       val (lhs,rhs) = FOLogic.dest-eq old
       val (lhead, largs) = strip-comb lhs
       and (rhead, rargs) = strip-comb rhs
       val lname = #1 (dest-Const lhead) handle TERM - => raise Match;
       val rname = #1 (dest-Const rhead) handle TERM - => raise Match;
       val lcon-info = the (Symtab.lookup (ConstructorsData.get sg) lname)
```

```
        handle Option => raise Match;
      val rcon-info = the (Symtab.lookup (ConstructorsData.get sg) rname)
        handle Option => raise Match;
      val new =
          if #big-rec-name lcon-info = #big-rec-name rcon-info
              andalso not (null (#free-iffs lcon-info)) then
              if lname = rname then mk-new (largs, rargs)
              else Const(False,FOLogic.oT)
          else raise Match
      val - =
        if !trace then writeln (NEW =  ^ Syntax.string-of-term-global sg new)
        else ();
      val goal = Logic.mk-equals (old, new)
      val thm = Goal.prove (Simplifier.the-context ss) [] [] goal
        (fn - => rtac iff-reflection 1 THEN
            simp-tac (Simplifier.inherit-context ss datatype-ss addsimps #free-iffs
lcon-info) 1)
        handle ERROR msg =>
      (warning (msg ^ \ndata-free simproc:\nfailed to prove  ^ Syntax.string-of-term-global
sg goal);
         raise Match)
   in SOME thm end
   handle Match => NONE;


 val conv = Simplifier.simproc @{theory} data-free [(x::i) = y] proc;

end;


Addsimprocs [DataFree.conv];
»
```

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*)
apply (*erule rev-mp*)
apply (*induct-tac k*, *auto*)
done

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*
by (*unfold pred-def*, *auto*)

lemma *natify-succ*: *natify*(*succ*(*x*)) = *succ*(*natify*(*x*))
by (*rule natify-def* [*THEN def-Vrecursor*, *THEN trans*], *auto*)

lemma *natify-0* [*simp*]: *natify*(*0*) = *0*
by (*rule natify-def* [*THEN def-Vrecursor*, *THEN trans*], *auto*)

lemma *natify-non-succ*: ∀ *z*. *x* ~= *succ*(*z*) ==> *natify*(*x*) = *0*
by (*rule natify-def* [*THEN def-Vrecursor*, *THEN trans*], *auto*)

lemma *natify-in-nat* [*iff*,*TC*]: *natify*(*x*) ∈ *nat*
apply (*rule-tac a*=*x* **in** *eps-induct*)
apply (*case-tac* ∃ *z*. *x* = *succ*(*z*))
apply (*auto simp add*: *natify-succ natify-non-succ*)
done

lemma *natify-ident* [*simp*]: *n* ∈ *nat* ==> *natify*(*n*) = *n*
apply (*induct-tac n*)
apply (*auto simp add*: *natify-succ*)
done

lemma *natify-eqE*: [|*natify*(*x*) = *y*; *x* ∈ *nat*|] ==> *x*=*y*
by *auto*

lemma *natify-idem* [*simp*]: *natify*(*natify*(*x*)) = *natify*(*x*)
by *simp*

lemma *add-natify1* [*simp*]: *natify*(*m*) #+ *n* = *m* #+ *n*
by (*simp add*: *add-def*)

lemma *add-natify2* [*simp*]: *m* #+ *natify*(*n*) = *m* #+ *n*
by (*simp add*: *add-def*)

lemma *mult-natify1* [*simp*]: *natify*(*m*) #* *n* = *m* #* *n*
by (*simp add*: *mult-def*)

lemma *mult-natify2* [*simp*]: *m* #* *natify*(*n*) = *m* #* *n*
by (*simp add*: *mult-def*)

lemma *diff-natify1* [*simp*]: *natify*(*m*) #− *n* = *m* #− *n*
by (*simp add*: *diff-def*)

lemma *diff-natify2* [*simp*]: *m* #− *natify*(*n*) = *m* #− *n*
by (*simp add*: *diff-def*)

lemma *mod-natify1* [*simp*]: *natify*(*m*) *mod n* = *m mod n*
by (*simp add*: *mod-def*)

lemma *mod-natify2* [*simp*]: *m mod natify*(*n*) = *m mod n*
by (*simp add*: *mod-def*)

lemma *div-natify1* [*simp*]: *natify*(*m*) *div n* = *m div n*
by (*simp add*: *div-def*)

lemma *div-natify2* [*simp*]: *m div natify*(*n*) = *m div n*
by (*simp add*: *div-def*)

26.2 Typing rules

lemma *raw-add-type*: [| *m*∈*nat*; *n*∈*nat* |] ==> *raw-add* (*m*, *n*) ∈ *nat*

by (*induct-tac m*, *auto*)

lemma *add-type* [*iff*,*TC*]: $m \#+ n \in nat$
by (*simp add*: *add-def raw-add-type*)

lemma *raw-mult-type*: [| $m\in nat$; $n\in nat$ |] ==> *raw-mult* $(m, n) \in nat$
apply (*induct-tac m*)
apply (*simp-all add*: *raw-add-type*)
done

lemma *mult-type* [*iff*,*TC*]: $m \#* n \in nat$
by (*simp add*: *mult-def raw-mult-type*)

lemma *raw-diff-type*: [| $m\in nat$; $n\in nat$ |] ==> *raw-diff* $(m, n) \in nat$
by (*induct-tac n*, *auto*)

lemma *diff-type* [*iff*,*TC*]: $m \#- n \in nat$
by (*simp add*: *diff-def raw-diff-type*)

lemma *diff-0-eq-0* [*simp*]: $0 \#- n = 0$
apply (*unfold diff-def*)
apply (*rule natify-in-nat* [*THEN nat-induct*], *auto*)
done

lemma *diff-succ-succ* [*simp*]: *succ*(m) #− *succ*(n) = $m \#- n$
apply (*simp add*: *natify-succ diff-def*)
apply (*rule-tac x1* = *n* **in** *natify-in-nat* [*THEN nat-induct*], *auto*)
done

declare *raw-diff-succ* [*simp del*]

lemma *diff-0* [*simp*]: $m \#- 0$ = *natify*(m)
by (*simp add*: *diff-def*)

lemma *diff-le-self*: $m\in nat$ ==> $(m \#- n)$ *le* m
apply (*subgoal-tac* (m #− *natify* (n)) *le* m)
apply (*rule-tac* [*2*] $m = m$ **and** n = *natify* (n) **in** *diff-induct*)
apply (*erule-tac* [*6*] *leE*)
apply (*simp-all add*: *le-iff*)
done

26.3 Addition

lemma *add-0-natify* [*simp*]: *0* #+ *m* = *natify*(*m*)
by (*simp add*: *add-def*)

lemma *add-succ* [*simp*]: *succ*(*m*) #+ *n* = *succ*(*m* #+ *n*)
by (*simp add*: *natify-succ add-def*)

lemma *add-0*: *m* ∈ *nat* ==> *0* #+ *m* = *m*
by *simp*

lemma *add-assoc*: (*m* #+ *n*) #+ *k* = *m* #+ (*n* #+ *k*)
apply (*subgoal-tac* (*natify*(*m*) #+ *natify*(*n*)) #+ *natify*(*k*) =
natify(*m*) #+ (*natify*(*n*) #+ *natify*(*k*)))
apply (*rule-tac* [*2*] *n* = *natify*(*m*) **in** *nat-induct*)
apply *auto*
done

lemma *add-0-right-natify* [*simp*]: *m* #+ *0* = *natify*(*m*)
apply (*subgoal-tac natify*(*m*) #+ *0* = *natify*(*m*))
apply (*rule-tac* [*2*] *n* = *natify*(*m*) **in** *nat-induct*)
apply *auto*
done

lemma *add-succ-right* [*simp*]: *m* #+ *succ*(*n*) = *succ*(*m* #+ *n*)
apply (*unfold add-def*)
apply (*rule-tac n* = *natify*(*m*) **in** *nat-induct*)
apply (*auto simp add*: *natify-succ*)
done

lemma *add-0-right*: *m* ∈ *nat* ==> *m* #+ *0* = *m*
by *auto*

lemma *add-commute*: *m* #+ *n* = *n* #+ *m*
apply (*subgoal-tac natify*(*m*) #+ *natify*(*n*) = *natify*(*n*) #+ *natify*(*m*))
apply (*rule-tac* [*2*] *n* = *natify*(*m*) **in** *nat-induct*)
apply *auto*
done

lemma *add-left-commute*: *m*#+(*n*#+*k*)=*n*#+(*m*#+*k*)
apply (*rule add-commute* [*THEN trans*])
apply (*rule add-assoc* [*THEN trans*])
apply (*rule add-commute* [*THEN subst-context*])
done

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

lemma *raw-add-left-cancel*:
[| *raw-add*(*k*, *m*) = *raw-add*(*k*, *n*); *k*∈*nat* |] ==> *m*=*n*
apply (*erule rev-mp*)
apply (*induct-tac k*, *auto*)
done

lemma *add-left-cancel-natify*: *k* #+ *m* = *k* #+ *n* ==> *natify*(*m*) = *natify*(*n*)
apply (*unfold add-def*)
apply (*drule raw-add-left-cancel*, *auto*)
done

lemma *add-left-cancel*:
[| *i* = *j*; *i* #+ *m* = *j* #+ *n*; *m*∈*nat*; *n*∈*nat* |] ==> *m* = *n*
by (*force dest*!: *add-left-cancel-natify*)

lemma *add-le-elim1-natify*: *k*#+*m* *le* *k*#+*n* ==> *natify*(*m*) *le* *natify*(*n*)
apply (*rule-tac P* = *natify*(*k*) #+*m* *le* *natify*(*k*) #+*n* **in** *rev-mp*)
apply (*rule-tac* [*2*] *n* = *natify*(*k*) **in** *nat-induct*)
apply *auto*
done

lemma *add-le-elim1*: [| *k*#+*m* *le* *k*#+*n*; *m* ∈ *nat*; *n* ∈ *nat* |] ==> *m le n*
by (*drule add-le-elim1-natify*, *auto*)

lemma *add-lt-elim1-natify*: *k*#+*m* < *k*#+*n* ==> *natify*(*m*) < *natify*(*n*)
apply (*rule-tac P* = *natify*(*k*) #+*m* < *natify*(*k*) #+*n* **in** *rev-mp*)
apply (*rule-tac* [*2*] *n* = *natify*(*k*) **in** *nat-induct*)
apply *auto*
done

lemma *add-lt-elim1*: [| *k*#+*m* < *k*#+*n*; *m* ∈ *nat*; *n* ∈ *nat* |] ==> *m* < *n*
by (*drule add-lt-elim1-natify*, *auto*)

lemma *zero-less-add*: [| *n* ∈ *nat*; *m* ∈ *nat* |] ==> *0* < *m* #+ *n* <−> (*0*<*m* | *0*<*n*)
by (*induct-tac n*, *auto*)

26.4 Monotonicity of Addition

lemma *add-lt-mono1*: [| *i*<*j*; *j*∈*nat* |] ==> *i*#+*k* < *j*#+*k*
apply (*frule lt-nat-in-nat*, *assumption*)
apply (*erule succ-lt-induct*)
apply (*simp-all add*: *leI*)
done

strict, in second argument

lemma *add-lt-mono2*: [| *i*<*j*; *j*∈*nat* |] ==> *k*#+*i* < *k*#+*j*
by (*simp add*: *add-commute* [*of k*] *add-lt-mono1*)

A [clumsy] way of lifting ¡ monotonicity to ≤ monotonicity

lemma *Ord-lt-mono-imp-le-mono*:
  assumes *lt-mono*: !!*i j*. [| *i*<*j*; *j*:*k* |] ==> *f*(*i*) < *f*(*j*)
      and *ford*:   !!*i*. *i*:*k* ==> *Ord*(*f*(*i*))
      and *leij*:   *i le j*
      and *jink*:   *j*:*k*
  shows *f*(*i*) *le f*(*j*)
apply (*insert leij jink*)
apply (*blast intro*!: *leCI lt-mono ford elim*!: *leE*)
done

≤ monotonicity, 1st argument

lemma *add-le-mono1*: [| *i le j*; *j*∈*nat* |] ==> *i*#+*k le j*#+*k*
apply (*rule-tac f* = %*j*. *j*#+*k* **in** *Ord-lt-mono-imp-le-mono*, *typecheck*)
apply (*blast intro*: *add-lt-mono1 add-type* [*THEN nat-into-Ord*])+
done

≤ monotonicity, both arguments

lemma *add-le-mono*: [| *i le j*; *k le l*; *j*∈*nat*; *l*∈*nat* |] ==> *i*#+*k le j*#+*l*
apply (*rule add-le-mono1* [*THEN le-trans*], *assumption*+)
apply (*subst add-commute*, *subst add-commute*, *rule add-le-mono1*, *assumption*+)
done

Combinations of less-than and less-than-or-equals

lemma *add-lt-le-mono*: [| *i*<*j*; *k*≤*l*; *j*∈*nat*; *l*∈*nat* |] ==> *i*#+*k* < *j*#+*l*
apply (*rule add-lt-mono1* [*THEN lt-trans2*], *assumption*+)
apply (*subst add-commute*, *subst add-commute*, *rule add-le-mono1*, *assumption*+)
done

lemma *add-le-lt-mono*: [| *i*≤*j*; *k*<*l*; *j*∈*nat*; *l*∈*nat* |] ==> *i*#+*k* < *j*#+*l*
by (*subst add-commute*, *subst add-commute*, *erule add-lt-le-mono*, *assumption*+)

Less-than: in other words, strict in both arguments

lemma *add-lt-mono*: [| *i*<*j*; *k*<*l*; *j*∈*nat*; *l*∈*nat* |] ==> *i*#+*k* < *j*#+*l*
apply (*rule add-lt-le-mono*)
apply (*auto intro*: *leI*)
done

lemma *diff-add-inverse*: (*n*#+*m*) #− *n* = *natify*(*m*)
apply (*subgoal-tac* (*natify*(*n*) #+ *m*) #− *natify*(*n*) = *natify*(*m*))
apply (*rule-tac* [*2*] *n* = *natify*(*n*) **in** *nat-induct*)
apply *auto*
done

lemma *diff-add-inverse2*: (m#+n) #− n = natify(m)
by (simp add: add-commute [of m] diff-add-inverse)

lemma *diff-cancel*: (k#+m) #− (k#+n) = m #− n
apply (subgoal-tac (natify(k) #+ natify(m)) #− (natify(k) #+ natify(n)) =
natify(m) #− natify(n))
apply (rule-tac [2] n = natify(k) **in** nat-induct)
apply auto
done

lemma *diff-cancel2*: (m#+k) #− (n#+k) = m #− n
by (simp add: add-commute [of - k] diff-cancel)

lemma *diff-add-0*: n #− (n#+m) = 0
apply (subgoal-tac natify(n) #− (natify(n) #+ natify(m)) = 0)
apply (rule-tac [2] n = natify(n) **in** nat-induct)
apply auto
done

lemma *pred-0* [simp]: pred(0) = 0
by (simp add: pred-def)

lemma *eq-succ-imp-eq-m1*: [|i = succ(j); i∈nat|] ==> j = i #− 1 & j ∈nat
by simp

lemma *pred-Un-distrib*:
[|i∈nat; j∈nat|] ==> pred(i Un j) = pred(i) Un pred(j)
apply (erule-tac n=i **in** natE, simp)
apply (erule-tac n=j **in** natE, simp)
apply (simp add: succ-Un-distrib [symmetric])
done

lemma *pred-type* [TC,simp]:
i ∈ nat ==> pred(i) ∈ nat
by (simp add: pred-def split: split-nat-case)

lemma *nat-diff-pred*: [|i∈nat; j∈nat|] ==> i #− succ(j) = pred(i #− j)
apply (rule-tac m=i **and** n=j **in** diff-induct)
apply (auto simp add: pred-def nat-imp-quasinat split: split-nat-case)
done

lemma *diff-succ-eq-pred*: i #− succ(j) = pred(i #− j)
apply (insert nat-diff-pred [of natify(i) natify(j)])
apply (simp add: natify-succ [symmetric])
done

lemma *nat-diff-Un-distrib*:
[|i∈nat; j∈nat; k∈nat|] ==> (i Un j) #− k = (i#−k) Un (j#−k)

apply (*rule-tac n=k* **in** *nat-induct*)
apply (*simp-all add*: *diff-succ-eq-pred pred-Un-distrib*)
done

lemma *diff-Un-distrib*:
  [|*i*∈*nat*; *j*∈*nat*|] ==> (*i Un j*) #− *k* = (*i*#−*k*) *Un* (*j*#−*k*)
by (*insert nat-diff-Un-distrib* [*of i j natify*(*k*)], *simp*)

We actually prove *i* #− *j* #− *k* = *i* #− (*j* #+ *k*)

lemma *diff-diff-left* [*simplified*]:
  natify(*i*)#−*natify*(*j*)#−*k* = *natify*(*i*) #− (*natify*(*j*)#+*k*)
by (*rule-tac m*=*natify*(*i*) **and** *n*=*natify*(*j*) **in** *diff-induct*, *auto*)

lemma *eq-add-iff*: (*u* #+ *m* = *u* #+ *n*) <−> (*0* #+ *m* = *natify*(*n*))
apply *auto*
apply (*blast dest*: *add-left-cancel-natify*)
apply (*simp add*: *add-def*)
done

lemma *less-add-iff*: (*u* #+ *m* < *u* #+ *n*) <−> (*0* #+ *m* < *natify*(*n*))
apply (*auto simp add*: *add-lt-elim1-natify*)
apply (*drule add-lt-mono1*)
apply (*auto simp add*: *add-commute* [*of u*])
done

lemma *diff-add-eq*: ((*u* #+ *m*) #− (*u* #+ *n*)) = ((*0* #+ *m*) #− *n*)
by (*simp add*: *diff-cancel*)

lemma *eq-cong2*: *u* = *u′* ==> (*t*==*u*) == (*t*==*u′*)
by *auto*

lemma *iff-cong2*: *u* <−> *u′* ==> (*t*==*u*) == (*t*==*u′*)
by *auto*

26.5 Multiplication

lemma *mult-0* [*simp*]: *0* #∗ *m* = *0*
by (*simp add*: *mult-def*)

lemma *mult-succ* [*simp*]: *succ*(*m*) #∗ *n* = *n* #+ (*m* #∗ *n*)
by (*simp add*: *add-def mult-def natify-succ raw-mult-type*)

lemma *mult-0-right* [*simp*]: *m* #∗ *0* = *0*
apply (*unfold mult-def*)

apply (*rule-tac n = natify*(*m*) **in** *nat-induct*)
apply *auto*
done

lemma *mult-succ-right* [*simp*]: *m* #∗ *succ*(*n*) = *m* #+ (*m* #∗ *n*)
apply (*subgoal-tac natify*(*m*) #∗ *succ* (*natify*(*n*)) =
natify(*m*) #+ (*natify*(*m*) #∗ *natify*(*n*)))
apply (*simp* (*no-asm-use*) *add*: *natify-succ add-def mult-def*)
apply (*rule-tac n = natify*(*m*) **in** *nat-induct*)
apply (*simp-all add*: *add-ac*)
done

lemma *mult-1-natify* [*simp*]: *1* #∗ *n* = *natify*(*n*)
by *auto*

lemma *mult-1-right-natify* [*simp*]: *n* #∗ *1* = *natify*(*n*)
by *auto*

lemma *mult-1*: *n* ∈ *nat* ==> *1* #∗ *n* = *n*
by *simp*

lemma *mult-1-right*: *n* ∈ *nat* ==> *n* #∗ *1* = *n*
by *simp*

lemma *mult-commute*: *m* #∗ *n* = *n* #∗ *m*
apply (*subgoal-tac natify*(*m*) #∗ *natify*(*n*) = *natify*(*n*) #∗ *natify*(*m*))
apply (*rule-tac* [*2*] *n = natify*(*m*) **in** *nat-induct*)
apply *auto*
done

lemma *add-mult-distrib*: (*m* #+ *n*) #∗ *k* = (*m* #∗ *k*) #+ (*n* #∗ *k*)
apply (*subgoal-tac* (*natify*(*m*) #+ *natify*(*n*)) #∗ *natify*(*k*) =
(*natify*(*m*) #∗ *natify*(*k*)) #+ (*natify*(*n*) #∗ *natify*(*k*)))
apply (*rule-tac* [*2*] *n = natify*(*m*) **in** *nat-induct*)
apply (*simp-all add*: *add-assoc* [*symmetric*])
done

lemma *add-mult-distrib-left*: *k* #∗ (*m* #+ *n*) = (*k* #∗ *m*) #+ (*k* #∗ *n*)
apply (*subgoal-tac natify*(*k*) #∗ (*natify*(*m*) #+ *natify*(*n*)) =
(*natify*(*k*) #∗ *natify*(*m*)) #+ (*natify*(*k*) #∗ *natify*(*n*)))
apply (*rule-tac* [*2*] *n = natify*(*m*) **in** *nat-induct*)
apply (*simp-all add*: *add-ac*)
done

lemma *mult-assoc*: (*m* #∗ *n*) #∗ *k* = *m* #∗ (*n* #∗ *k*)
apply (*subgoal-tac* (*natify*(*m*) #∗ *natify*(*n*)) #∗ *natify*(*k*) =
natify(*m*) #∗ (*natify*(*n*) #∗ *natify*(*k*)))
apply (*rule-tac* [*2*] *n* = *natify*(*m*) **in** *nat-induct*)
apply (*simp-all add*: *add-mult-distrib*)
done

lemma *mult-left-commute*: *m* #∗ (*n* #∗ *k*) = *n* #∗ (*m* #∗ *k*)
apply (*rule mult-commute* [*THEN trans*])
apply (*rule mult-assoc* [*THEN trans*])
apply (*rule mult-commute* [*THEN subst-context*])
done

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

lemma *lt-succ-eq-0-disj*:
[| *m*∈*nat*; *n*∈*nat* |]
==> (*m* < *succ*(*n*)) <−> (*m* = *0* | (∃ *j*∈*nat*. *m* = *succ*(*j*) & *j* < *n*))
by (*induct-tac m*, *auto*)

lemma *less-diff-conv* [*rule-format*]:
[| *j*∈*nat*; *k*∈*nat* |] ==> ∀ *i*∈*nat*. (*i* < *j* #− *k*) <−> (*i* #+ *k* < *j*)
by (*erule-tac m* = *k* **in** *diff-induct*, *auto*)

lemmas *nat-typechecks* = *rec-type nat-0I nat-1I nat-succI Ord-nat*

end

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*
apply (*subgoal-tac natify* (*m*) #− *natify* (*m*) = *0*)
apply (*rule-tac* [*2*] *natify-in-nat* [*THEN nat-induct*], *auto*)
done

lemma *add-diff-inverse*: [| n le m; m:nat |] ==> n #+ (m#−n) = m
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*erule rev-mp*)
apply (*rule-tac* m = m **and** n = n **in** *diff-induct*, *auto*)
done

lemma *add-diff-inverse2*: [| n le m; m:nat |] ==> (m#−n) #+ n = m
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*simp* (*no-asm-simp*) *add*: *add-commute add-diff-inverse*)
done

lemma *diff-succ*: [| n le m; m:nat |] ==> succ(m) #− n = succ(m#−n)
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*erule rev-mp*)
apply (*rule-tac* m = m **and** n = n **in** *diff-induct*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *zero-less-diff* [*simp*]:
  [| m: nat; n: nat |] ==> 0 < (n #− m) <−> m<n
apply (*rule-tac* m = m **and** n = n **in** *diff-induct*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *diff-mult-distrib*: (m #− n) #* k = (m #* k) #− (n #* k)
apply (*subgoal-tac* (natify (m) #− natify (n)) #* natify (k) = (natify (m) #* natify (k)) #− (natify (n) #* natify (k)))
apply (*rule-tac* [2] m = natify (m) **and** n = natify (n) **in** *diff-induct*)
apply (*simp-all add*: *diff-cancel*)
done

lemma *diff-mult-distrib2*: k #* (m #− n) = (k #* m) #− (k #* n)
apply (*simp* (*no-asm*) *add*: *mult-commute* [*of k*] *diff-mult-distrib*)
done

27.2 Remainder

lemma *div-termination*: [| 0<n; n le m; m:nat |] ==> m #− n < m
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*erule rev-mp*)
apply (*erule rev-mp*)
apply (*rule-tac* m = m **and** n = n **in** *diff-induct*)
apply (*simp-all* (*no-asm-simp*) *add*: *diff-le-self*)
done

lemmas *div-rls* =
nat-typechecks Ord-transrec-type apply-funtype
div-termination [*THEN ltD*]
nat-into-Ord not-lt-iff-le [*THEN iffD1*]

lemma *raw-mod-type*: [| *m*:*nat*; *n*:*nat* |] ==> *raw-mod* (*m*, *n*) : *nat*
apply (*unfold raw-mod-def*)
apply (*rule Ord-transrec-type*)
apply (*auto simp add*: *nat-into-Ord* [*THEN Ord-0-lt-iff*])
apply (*blast intro*: *div-rls*)
done

lemma *mod-type* [*TC*,*iff*]: *m mod n* : *nat*
apply (*unfold mod-def*)
apply (*simp* (*no-asm*) *add*: *mod-def raw-mod-type*)
done

lemma *DIVISION-BY-ZERO-DIV*: *a div 0* = *0*
apply (*unfold div-def*)
apply (*rule raw-div-def* [*THEN def-transrec*, *THEN trans*])
apply (*simp* (*no-asm-simp*))
done

lemma *DIVISION-BY-ZERO-MOD*: *a mod 0* = *natify*(*a*)
apply (*unfold mod-def*)
apply (*rule raw-mod-def* [*THEN def-transrec*, *THEN trans*])
apply (*simp* (*no-asm-simp*))
done

lemma *raw-mod-less*: *m*<*n* ==> *raw-mod* (*m*,*n*) = *m*
apply (*rule raw-mod-def* [*THEN def-transrec*, *THEN trans*])
apply (*simp* (*no-asm-simp*) *add*: *div-termination* [*THEN ltD*])
done

lemma *mod-less* [*simp*]: [| *m*<*n*; *n* : *nat* |] ==> *m mod n* = *m*
apply (*frule lt-nat-in-nat*, *assumption*)
apply (*simp* (*no-asm-simp*) *add*: *mod-def raw-mod-less*)
done

lemma *raw-mod-geq*:
[| *0*<*n*; *n le m*; *m*:*nat* |] ==> *raw-mod* (*m*, *n*) = *raw-mod* (*m*#−*n*, *n*)
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*rule raw-mod-def* [*THEN def-transrec*, *THEN trans*])
apply (*simp* (*no-asm-simp*) *add*: *div-termination* [*THEN ltD*] *not-lt-iff-le* [*THEN*

iffD2], *blast*)
done

lemma *mod-geq*: [| *n le m*; *m*:*nat* |] ==> *m mod n* = (*m*#−*n*) *mod n*
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*case-tac n*=*0*)
apply (*simp add*: *DIVISION-BY-ZERO-MOD*)
apply (*simp add*: *mod-def raw-mod-geq nat-into-Ord* [*THEN Ord-0-lt-iff*])
done

27.3 Division

lemma *raw-div-type*: [| *m*:*nat*; *n*:*nat* |] ==> *raw-div* (*m*, *n*) : *nat*
apply (*unfold raw-div-def*)
apply (*rule Ord-transrec-type*)
apply (*auto simp add*: *nat-into-Ord* [*THEN Ord-0-lt-iff*])
apply (*blast intro*: *div-rls*)
done

lemma *div-type* [*TC*,*iff*]: *m div n* : *nat*
apply (*unfold div-def*)
apply (*simp* (*no-asm*) *add*: *div-def raw-div-type*)
done

lemma *raw-div-less*: *m*<*n* ==> *raw-div* (*m*,*n*) = *0*
apply (*rule raw-div-def* [*THEN def-transrec*, *THEN trans*])
apply (*simp* (*no-asm-simp*) *add*: *div-termination* [*THEN ltD*])
done

lemma *div-less* [*simp*]: [| *m*<*n*; *n* : *nat* |] ==> *m div n* = *0*
apply (*frule lt-nat-in-nat*, *assumption*)
apply (*simp* (*no-asm-simp*) *add*: *div-def raw-div-less*)
done

lemma *raw-div-geq*: [| *0*<*n*; *n le m*; *m*:*nat* |] ==> *raw-div*(*m*,*n*) = *succ*(*raw-div*(*m*#−*n*, *n*))
apply (*subgoal-tac n* ~= *0*)
prefer *2* **apply** *blast*
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*rule raw-div-def* [*THEN def-transrec*, *THEN trans*])
apply (*simp* (*no-asm-simp*) *add*: *div-termination* [*THEN ltD*] *not-lt-iff-le* [*THEN iffD2*])
done

lemma *div-geq* [*simp*]:
[| *0*<*n*; *n le m*; *m*:*nat* |] ==> *m div n* = *succ* ((*m*#−*n*) *div n*)
apply (*frule lt-nat-in-nat*, *erule nat-succI*)
apply (*simp* (*no-asm-simp*) *add*: *div-def raw-div-geq*)

done

declare *div-less* [*simp*] *div-geq* [*simp*]

lemma *mod-div-lemma*: [| *m*: *nat*; *n*: *nat* |] ==> (*m div n*)#**n* #+ *m mod n* = *m*
apply (*case-tac n=0*)
apply (*simp add*: *DIVISION-BY-ZERO-MOD*)
apply (*simp add*: *nat-into-Ord* [*THEN Ord-0-lt-iff*])
apply (*erule complete-induct*)
apply (*case-tac x*<*n*)

case x¡n

apply (*simp* (*no-asm-simp*))

case n le x

apply (*simp add*: *not-lt-iff-le add-assoc mod-geq div-termination* [*THEN ltD*] *add-diff-inverse*)
done

lemma *mod-div-equality-natify*: (*m div n*)#**n* #+ *m mod n* = *natify*(*m*)
apply (*subgoal-tac* (*natify* (*m*) *div natify* (*n*))#**natify* (*n*) #+ *natify* (*m*) *mod natify* (*n*) = *natify* (*m*))
apply *force*
apply (*subst mod-div-lemma*, *auto*)
done

lemma *mod-div-equality*: *m*: *nat* ==> (*m div n*)#**n* #+ *m mod n* = *m*
apply (*simp* (*no-asm-simp*) *add*: *mod-div-equality-natify*)
done

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*))
apply (*erule complete-induct*)
apply (*case-tac succ* (*x*) <*n*)

case succ(x) ¡ n

apply (*simp* (*no-asm-simp*) *add*: *nat-le-refl* [*THEN lt-trans*] *succ-neq-self*)
apply (*simp add*: *ltD* [*THEN mem-imp-not-eq*])

case n le succ(x)

apply (*simp add*: *mod-geq not-lt-iff-le*)
apply (*erule leE*)

apply (*simp* (*no-asm-simp*) *add*: *mod-geq div-termination* [*THEN ltD*] *diff-succ*)

equality case

apply (*simp add*: *diff-self-eq-0*)
done

lemma *mod-succ*:
n:*nat* ==> *succ*(*m*) *mod n* = (*if succ*(*m mod n*) = *n then 0 else succ*(*m mod n*))
apply (*case-tac n=0*)
apply (*simp* (*no-asm-simp*) *add*: *natify-succ DIVISION-BY-ZERO-MOD*)
apply (*subgoal-tac natify* (*succ* (*m*)) *mod n* = (*if succ* (*natify* (*m*) *mod n*) = *n then 0 else succ* (*natify* (*m*) *mod n*)))
prefer *2*
apply (*subst natify-succ*)
apply (*rule mod-succ-lemma*)
apply (*auto simp del*: *natify-succ simp add*: *nat-into-Ord* [*THEN Ord-0-lt-iff*])
done

lemma *mod-less-divisor*: [| *0*<*n*; *n*:*nat* |] ==> *m mod n* < *n*
apply (*subgoal-tac natify* (*m*) *mod n* < *n*)
apply (*rule-tac* [*2*] *i* = *natify* (*m*) **in** *complete-induct*)
apply (*case-tac* [*3*] *x*<*n*, *auto*)

case n le x

apply (*simp add*: *mod-geq not-lt-iff-le div-termination* [*THEN ltD*])
done

lemma *mod-1-eq* [*simp*]: *m mod 1* = *0*
by (*cut-tac n* = *1* **in** *mod-less-divisor*, *auto*)

lemma *mod2-cases*: *b*<*2* ==> *k mod 2* = *b* | *k mod 2* = (*if b=1 then 0 else 1*)
apply (*subgoal-tac k mod 2*: *2*)
prefer *2* **apply** (*simp add*: *mod-less-divisor* [*THEN ltD*])
apply (*drule ltD*, *auto*)
done

lemma *mod2-succ-succ* [*simp*]: *succ*(*succ*(*m*)) *mod 2* = *m mod 2*
apply (*subgoal-tac m mod 2*: *2*)
prefer *2* **apply** (*simp add*: *mod-less-divisor* [*THEN ltD*])
apply (*auto simp add*: *mod-succ*)
done

lemma *mod2-add-more* [*simp*]: (*m*#+*m*#+*n*) *mod 2* = *n mod 2*
apply (*subgoal-tac* (*natify* (*m*) #+*natify* (*m*) #+*n*) *mod 2* = *n mod 2*)
apply (*rule-tac* [*2*] *n* = *natify* (*m*) **in** *nat-induct*)
apply *auto*
done

lemma *mod2-add-self* [*simp*]: (*m*#+*m*) *mod 2* = *0*
by (*cut-tac n* = *0* **in** *mod2-add-more*, *auto*)

27.5 Additional theorems about ≤

lemma *add-le-self*: *m*:*nat* ==> *m le* (*m* #+ *n*)
apply (*simp* (*no-asm-simp*))
done

lemma *add-le-self2*: *m*:*nat* ==> *m le* (*n* #+ *m*)
apply (*simp* (*no-asm-simp*))
done

lemma *mult-le-mono1*: [| *i le j*; *j*:*nat* |] ==> (*i*#**k*) *le* (*j*#**k*)
apply (*subgoal-tac natify* (*i*) #**natify* (*k*) *le j*#**natify* (*k*))
apply (*frule-tac* [*2*] *lt-nat-in-nat*)
apply (*rule-tac* [*3*] *n* = *natify* (*k*) **in** *nat-induct*)
apply (*simp-all add*: *add-le-mono*)
done

lemma *mult-le-mono*: [| *i le j*; *k le l*; *j*:*nat*; *l*:*nat* |] ==> *i*#**k le j*#**l*
apply (*rule mult-le-mono1* [*THEN le-trans*], *assumption*+)
apply (*subst mult-commute*, *subst mult-commute*, *rule mult-le-mono1*, *assumption*+)
done

lemma *mult-lt-mono2*: [| *i*<*j*; *0*<*k*; *j*:*nat*; *k*:*nat* |] ==> *k*#**i* < *k*#**j*
apply (*erule zero-lt-natE*)
apply (*frule-tac* [*2*] *lt-nat-in-nat*)
apply (*simp-all* (*no-asm-simp*))
apply (*induct-tac x*)
apply (*simp-all* (*no-asm-simp*) *add*: *add-lt-mono*)
done

lemma *mult-lt-mono1*: [| *i*<*j*; *0*<*k*; *j*:*nat*; *k*:*nat* |] ==> *i*#**k* < *j*#**k*
apply (*simp* (*no-asm-simp*) *add*: *mult-lt-mono2 mult-commute* [*of - k*])
done

lemma *add-eq-0-iff* [*iff*]: *m*#+*n* = *0* <-> *natify*(*m*)=*0* & *natify*(*n*)=*0*
apply (*subgoal-tac natify* (*m*) #+ *natify* (*n*) = *0* <-> *natify* (*m*) =*0* & *natify* (*n*) =*0*)
apply (*rule-tac* [*2*] *n* = *natify* (*m*) **in** *natE*)
apply (*rule-tac* [*4*] *n* = *natify* (*n*) **in** *natE*)
apply *auto*
done

lemma *zero-lt-mult-iff* [*iff*]: *0* < *m*#**n* <-> *0* < *natify*(*m*) & *0* < *natify*(*n*)
apply (*subgoal-tac 0* < *natify* (*m*) #**natify* (*n*) <-> *0* < *natify* (*m*) & *0* < *natify* (*n*))

apply (*rule-tac* [*2*] *n* = *natify* (*m*) **in** *natE*)
 apply (*rule-tac* [*4*] *n* = *natify* (*n*) **in** *natE*)
  apply (*rule-tac* [*3*] *n* = *natify* (*n*) **in** *natE*)
apply *auto*
done

lemma *mult-eq-1-iff* [*iff*]: *m*#**n* = *1* <−> *natify*(*m*)=*1* & *natify*(*n*)=*1*
apply (*subgoal-tac natify* (*m*) #* *natify* (*n*) = *1* <−> *natify* (*m*) =*1* & *natify* (*n*) =*1*)
apply (*rule-tac* [*2*] *n* = *natify* (*m*) **in** *natE*)
 apply (*rule-tac* [*4*] *n* = *natify* (*n*) **in** *natE*)
apply *auto*
done

lemma *mult-is-zero*: [|*m*: *nat*; *n*: *nat*|] ==> (*m* #* *n* = *0*) <−> (*m* = *0* | *n* = *0*)
apply *auto*
apply (*erule natE*)
apply (*erule-tac* [*2*] *natE*, *auto*)
done

lemma *mult-is-zero-natify* [*iff*]:
    (*m* #* *n* = *0*) <−> (*natify*(*m*) = *0* | *natify*(*n*) = *0*)
apply (*cut-tac m* = *natify* (*m*) **and** *n* = *natify* (*n*) **in** *mult-is-zero*)
apply *auto*
done

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*)
apply (*safe intro*!: *mult-lt-mono1*)
apply (*erule natE*, *auto*)
apply (*rule not-le-iff-lt* [*THEN iffD1*])
apply (*drule-tac* [*3*] *not-le-iff-lt* [*THEN* [*2*] *rev-iffD2*])
prefer *5* **apply** (*blast intro*: *mult-le-mono1*, *auto*)
done

lemma *mult-less-cancel2* [*simp*]:
    (*m*#**k* < *n*#**k*) <−> (*0* < *natify*(*k*) & *natify*(*m*) < *natify*(*n*))
apply (*rule iff-trans*)
apply (*rule-tac* [*2*] *mult-less-cancel-lemma*, *auto*)
done

lemma *mult-less-cancel1* [*simp*]:
    (*k*#**m* < *k*#**n*) <−> (*0* < *natify*(*k*) & *natify*(*m*) < *natify*(*n*))
apply (*simp* (*no-asm*) *add*: *mult-less-cancel2 mult-commute* [*of k*])
done

lemma *mult-le-cancel2* [*simp*]: (*m#*k le n#*k*) <−> (*0* < *natify*(*k*) −−> *natify*(*m*) *le natify*(*n*))
apply (*simp* (*no-asm-simp*) *add*: *not-lt-iff-le* [*THEN iff-sym*])
apply *auto*
done

lemma *mult-le-cancel1* [*simp*]: (*k#*m le k#*n*) <−> (*0* < *natify*(*k*) −−> *natify*(*m*) *le natify*(*n*))
apply (*simp* (*no-asm-simp*) *add*: *not-lt-iff-le* [*THEN iff-sym*])
apply *auto*
done

lemma *mult-le-cancel-le1*: *k* : *nat* ==> *k* #* *m le k* ⟷ (*0* < *k* ⟶ *natify*(*m*) *le 1*)
by (*cut-tac k* = *k* **and** *m* = *m* **and** *n* = *1* **in** *mult-le-cancel1*, *auto*)

lemma *Ord-eq-iff-le*: [| *Ord*(*m*); *Ord*(*n*) |] ==> *m*=*n* <−> (*m le n* & *n le m*)
by (*blast intro*: *le-anti-sym*)

lemma *mult-cancel2-lemma*:
    [| *k*: *nat*; *m*: *nat*; *n*: *nat* |] ==> (*m#*k* = *n#*k*) <−> (*m*=*n* | *k*=*0*)
apply (*simp* (*no-asm-simp*) *add*: *Ord-eq-iff-le* [*of m#*k*] *Ord-eq-iff-le* [*of m*])
apply (*auto simp add*: *Ord-0-lt-iff*)
done

lemma *mult-cancel2* [*simp*]:
    (*m#*k* = *n#*k*) <−> (*natify*(*m*) = *natify*(*n*) | *natify*(*k*) = *0*)
apply (*rule iff-trans*)
apply (*rule-tac* [*2*] *mult-cancel2-lemma*, *auto*)
done

lemma *mult-cancel1* [*simp*]:
    (*k#*m* = *k#*n*) <−> (*natify*(*m*) = *natify*(*n*) | *natify*(*k*) = *0*)
apply (*simp* (*no-asm*) *add*: *mult-cancel2 mult-commute* [*of k*])
done

lemma *div-cancel-raw*:
    [| *0*<*n*; *0*<*k*; *k*:*nat*; *m*:*nat*; *n*:*nat* |] ==> (*k#*m*) *div* (*k#*n*) = *m div n*
apply (*erule-tac i* = *m* **in** *complete-induct*)
apply (*case-tac x*<*n*)
apply (*simp add*: *div-less zero-lt-mult-iff mult-lt-mono2*)
apply (*simp add*: *not-lt-iff-le zero-lt-mult-iff le-refl* [*THEN mult-le-mono*]
        div-geq diff-mult-distrib2 [*symmetric*] *div-termination* [*THEN ltD*])
done

lemma *div-cancel*:
[| *0* < *natify*(*n*); *0* < *natify*(*k*) |] ==> (*k*#∗*m*) *div* (*k*#∗*n*) = *m div n*
apply (*cut-tac k* = *natify* (*k*) **and** *m* = *natify* (*m*) **and** *n* = *natify* (*n*)
in *div-cancel-raw*)
apply *auto*
done

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*)
apply (*case-tac k*=*0*)
apply (*simp add*: *DIVISION-BY-ZERO-MOD*)
apply (*case-tac n*=*0*)
apply (*simp add*: *DIVISION-BY-ZERO-MOD*)
apply (*simp add*: *nat-into-Ord* [*THEN Ord-0-lt-iff*])
apply (*erule-tac i* = *m* **in** *complete-induct*)
apply (*case-tac x*<*n*)
apply (*simp* (*no-asm-simp*) *add*: *mod-less zero-lt-mult-iff mult-lt-mono2*)
apply (*simp add*: *not-lt-iff-le zero-lt-mult-iff le-refl* [*THEN mult-le-mono*]
mod-geq diff-mult-distrib2 [*symmetric*] *div-termination* [*THEN ltD*])
done

lemma *mod-mult-distrib2*: *k* #∗ (*m mod n*) = (*k*#∗*m*) *mod* (*k*#∗*n*)
apply (*cut-tac k* = *natify* (*k*) **and** *m* = *natify* (*m*) **and** *n* = *natify* (*n*)
in *mult-mod-distrib-raw*)
apply *auto*
done

lemma *mult-mod-distrib*: (*m mod n*) #∗ *k* = (*m*#∗*k*) *mod* (*n*#∗*k*)
apply (*simp* (*no-asm*) *add*: *mult-commute mod-mult-distrib2*)
done

lemma *mod-add-self2-raw*: *n* ∈ *nat* ==> (*m* #+ *n*) *mod n* = *m mod n*
apply (*subgoal-tac* (*n* #+ *m*) *mod n* = (*n* #+ *m* #− *n*) *mod n*)
apply (*simp add*: *add-commute*)
apply (*subst mod-geq* [*symmetric*], *auto*)
done

lemma *mod-add-self2* [*simp*]: (*m* #+ *n*) *mod n* = *m mod n*
apply (*cut-tac n* = *natify* (*n*) **in** *mod-add-self2-raw*)
apply *auto*
done

lemma *mod-add-self1* [*simp*]: (*n*#+*m*) *mod n* = *m mod n*
apply (*simp* (*no-asm-simp*) *add*: *add-commute mod-add-self2*)
done

lemma *mod-mult-self1-raw*: *k* ∈ *nat* ==> (*m* #+ *k*#∗*n*) *mod n* = *m mod n*

apply (*erule nat-induct*)
apply (*simp-all* (*no-asm-simp*) *add*: *add-left-commute* [*of - n*])
done

lemma *mod-mult-self1* [*simp*]: (*m* #+ *k*#**n*) *mod n* = *m mod n*
apply (*cut-tac k* = *natify* (*k*) **in** *mod-mult-self1-raw*)
apply *auto*
done

lemma *mod-mult-self2* [*simp*]: (*m* #+ *n*#**k*) *mod n* = *m mod n*
apply (*simp* (*no-asm*) *add*: *mult-commute mod-mult-self1*)
done

lemma *mult-eq-self-implies-10*: *m* = *m*#**n* ==> *natify*(*n*)=*1* | *m*=*0*
apply (*subgoal-tac m*: *nat*)
 prefer *2*
 apply (*erule ssubst*)
 apply *simp*
apply (*rule disjCI*)
apply (*drule sym*)
apply (*rule Ord-linear-lt* [*of natify*(*n*) *1*])
apply *simp-all*
 apply (*subgoal-tac m* #* *n* = *0*, *simp*)
 apply (*subst mult-natify2* [*symmetric*])
 apply (*simp del*: *mult-natify2*)
apply (*drule nat-into-Ord* [*THEN Ord-0-lt*, *THEN* [*2*] *mult-lt-mono2*], *auto*)
done

lemma *less-imp-succ-add* [*rule-format*]:
  [| *m*<*n*; *n*: *nat* |] ==> *EX k*: *nat*. *n* = *succ*(*m*#+*k*)
apply (*frule lt-nat-in-nat*, *assumption*)
apply (*erule rev-mp*)
apply (*induct-tac n*)
apply (*simp-all* (*no-asm*) *add*: *le-iff*)
apply (*blast elim*!: *leE intro*!: *add-0-right* [*symmetric*] *add-succ-right* [*symmetric*])
done

lemma *less-iff-succ-add*:
  [| *m*: *nat*; *n*: *nat* |] ==> (*m*<*n*) <-> (*EX k*: *nat*. *n* = *succ*(*m*#+*k*))
by (*auto intro*: *less-imp-succ-add*)

lemma *add-lt-elim2*:
  $[\![a \ \#+ \ d = b \ \#+ \ c;\ a < b;\ b \in nat;\ c \in nat;\ d \in nat]\!] \Longrightarrow c < d$
by (*drule less-imp-succ-add*, *auto*)

lemma *add-le-elim2*:
  $[\![a \ \#+ \ d = b \ \#+ \ c;\ a\ le\ b;\ b \in nat;\ c \in nat;\ d \in nat]\!] \Longrightarrow c\ le\ d$
by (*drule less-imp-succ-add*, *auto*)

27.7.1 More Lemmas About Difference

lemma *diff-is-0-lemma*:
  [| m: nat; n: nat |] ==> m #− n = 0 <−> m le n
apply (*rule-tac* m = m **and** n = n **in** *diff-induct*, *simp-all*)
done

lemma *diff-is-0-iff*: m #− n = 0 <−> natify(m) le natify(n)
by (*simp add*: *diff-is-0-lemma* [*symmetric*])

lemma *nat-lt-imp-diff-eq-0*:
  [| a:nat; b:nat; a<b |] ==> a #− b = 0
by (*simp add*: *diff-is-0-iff le-iff*)

lemma *raw-nat-diff-split*:
  [| a:nat; b:nat |] ==>
   (P(a #− b)) <−> ((a < b −−>P(0)) & (ALL d:nat. a = b #+ d −−>
P(d)))
apply (*case-tac* a < b)
 apply (*force simp add*: *nat-lt-imp-diff-eq-0*)
apply (*rule iffI*, *force*, *simp*)
apply (*drule-tac* x=a#−b **in** *bspec*)
apply (*simp-all add*: *Ordinal.not-lt-iff-le add-diff-inverse*)
done

lemma *nat-diff-split*:
 (P(a #− b)) <−>
  (natify(a) < natify(b) −−>P(0)) & (ALL d:nat. natify(a) = b #+ d −−>
P(d))
apply (*cut-tac* P=P **and** a=natify(a) **and** b=natify(b) **in** *raw-nat-diff-split*)
apply *simp-all*
done

Difference and less-than

lemma *diff-lt-imp-lt*: [|(k#−i) < (k#−j); i∈nat; j∈nat; k∈nat|] ==> j<i
apply (*erule rev-mp*)
apply (*simp split add*: *nat-diff-split*, *auto*)
 apply (*blast intro*: *add-le-self lt-trans1*)
apply (*rule not-le-iff-lt* [*THEN iffD1*], *auto*)
apply (*subgoal-tac* i #+ da < j #+ d, *force*)
apply (*blast intro*: *add-le-lt-mono*)
done

lemma *lt-imp-diff-lt*: [|j<i; i≤k; k∈nat|] ==> (k#−i) < (k#−j)
apply (*frule le-in-nat*, *assumption*)
apply (*frule lt-nat-in-nat*, *assumption*)
apply (*simp split add*: *nat-diff-split*, *auto*)
  apply (*blast intro*: *lt-asym lt-trans2*)
 apply (*blast intro*: *lt-irrefl lt-trans2*)
apply (*rule not-le-iff-lt* [*THEN iffD1*], *auto*)

apply (*subgoal-tac j #+ d < i #+ da*, *force*)
apply (*blast intro*: *add-lt-le-mono*)
done

lemma *diff-lt-iff-lt*: [|*i*≤*k*; *j*∈*nat*; *k*∈*nat*|] ==> (*k*#−*i*) < (*k*#−*j*) <−> *j*<*i*
apply (*frule le-in-nat*, *assumption*)
apply (*blast intro*: *lt-imp-diff-lt diff-lt-imp-lt*)
done

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-of-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-of-list([]) = *0*
set-of-list(*Cons*(*a*,*l*)) = *cons*(*a*, *set-of-list*(*l*))

primrec
app-Nil: [] @ *ys* = *ys*
app-Cons: (*Cons*(*a*,*l*)) @ *ys* = *Cons*(*a*, *l* @ *ys*)

consts
rev :: *i*=>*i*
flat :: *i*=>*i*
list-add :: *i*=>*i*

primrec
rev([]) = []
rev(*Cons*(*a*,*l*)) = *rev*(*l*) @ [*a*]

primrec
flat([]) = []
flat(*Cons*(*l*,*ls*)) = *l* @ *flat*(*ls*)

primrec
list-add([]) = *0*
list-add(*Cons*(*a*,*l*)) = *a* #+ *list-add*(*l*)

consts
drop :: [*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-rec*(*lam* n:*nat*. [],
%$a\ l\ r$. *lam* n:*nat*. *nat-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-rec*(*lam* n:*nat*. *0*,
%$a\ l\ r$. *lam* n:*nat*. *nat-case*(a, %m. r‘m, n), as) ‘ n

definition
list-update :: $[i, i, i]=>i$ **where**
list-update(xs, i, v) == *list-rec*(*lam* n:*nat*. *Nil*,
%u *us vs*. *lam* n:*nat*. *nat-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*)) =
(*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**
min(x, y) == (*if* x *le* y *then* x *else* y)

definition
max :: $[i, i] =>i$ **where**
max(x, y) == (*if* x *le* y *then* y *else* x)

declare *list.intros* [*simp*,*TC*]

inductive-cases *ConsE*: *Cons*(a,l) : *list*(A)

lemma *Cons-type-iff* [*simp*]: *Cons*(a,l) ∈ *list*(A) <−> $a \in A$ & $l \in$ *list*(A)
by (*blast elim*: *ConsE*)

```
lemma Cons-iff: Cons(a,l)=Cons(a′,l′) <-> a=a′ & l=l′
by auto

lemma Nil-Cons-iff: ~ Nil=Cons(a,l)
by auto

lemma list-unfold: list(A) = {0} + (A * list(A))
by (blast intro!: list.intros [unfolded list.con-defs]
        elim: list.cases [unfolded list.con-defs])


lemma list-mono: A<=B ==> list(A) <= list(B)
apply (unfold list.defs )
apply (rule lfp-mono)
apply (simp-all add: list.bnd-mono)
apply (assumption | rule univ-mono basic-monos)+
done


lemma list-univ: list(univ(A)) <= univ(A)
apply (unfold list.defs list.con-defs)
apply (rule lfp-lowerbound)
apply (rule-tac [2] A-subset-univ [THEN univ-mono])
apply (blast intro!: zero-in-univ Inl-in-univ Inr-in-univ Pair-in-univ)
done


lemmas list-subset-univ = subset-trans [OF list-mono list-univ]

lemma list-into-univ: [| l: list(A);  A <= univ(B) |] ==> l: univ(B)
by (blast intro: list-subset-univ [THEN subsetD])

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-case(c,h,l) : C(l)
by (erule list.induct, auto)

lemma list-0-triv: list(0) = {Nil}
apply (rule equalityI, auto)
apply (induct-tac x, auto)
done
```

lemma *tl-type*: *l*: *list*(*A*) ==> *tl*(*l*) : *list*(*A*)
apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*) *add*: *list.intros*)
done

lemma *drop-Nil* [*simp*]: *i*:*nat* ==> *drop*(*i*, *Nil*) = *Nil*
apply (*induct-tac i*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *drop-succ-Cons* [*simp*]: *i*:*nat* ==> *drop*(*succ*(*i*), *Cons*(*a*,*l*)) = *drop*(*i*,*l*)
apply (*rule sym*)
apply (*induct-tac i*)
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm-simp*))
done

lemma *drop-type* [*simp*,*TC*]: [| *i*:*nat*; *l*: *list*(*A*) |] ==> *drop*(*i*,*l*) : *list*(*A*)
apply (*induct-tac i*)
apply (*simp-all* (*no-asm-simp*) *add*: *tl-type*)
done

declare *drop-succ* [*simp del*]

lemma *list-rec-type* [*TC*]:
  [| *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*)
by (*induct-tac l*, *auto*)

lemma *map-type* [*TC*]:
  [| *l*: *list*(*A*); !!*x*. *x*: *A* ==> *h*(*x*): *B* |] ==> *map*(*h*,*l*) : *list*(*B*)
apply (*simp add*: *map-list-def*)
apply (*typecheck add*: *list.intros list-rec-type*, *blast*)
done

lemma *map-type2* [*TC*]: *l*: *list*(*A*) ==> *map*(*h*,*l*) : *list*({*h*(*u*). *u*:*A*})
apply (*erule map-type*)
apply (*erule RepFunI*)
done

lemma *length-type* [*TC*]: *l*: *list*(*A*) ==> *length*(*l*) : *nat*
by (*simp add*: *length-list-def*)

lemma *lt-length-in-nat*:
[|*x* < *length*(*xs*); *xs* ∈ *list*(*A*)|] ==> *x* ∈ *nat*
by (*frule lt-nat-in-nat*, *typecheck*)

lemma *app-type* [*TC*]: [| *xs*: *list*(*A*); *ys*: *list*(*A*) |] ==> *xs*@*ys* : *list*(*A*)
by (*simp add*: *app-list-def*)

lemma *rev-type* [*TC*]: *xs*: *list*(*A*) ==> *rev*(*xs*) : *list*(*A*)
by (*simp add*: *rev-list-def*)

lemma *flat-type* [*TC*]: *ls*: *list*(*list*(*A*)) ==> *flat*(*ls*) : *list*(*A*)
by (*simp add*: *flat-list-def*)

lemma *set-of-list-type* [*TC*]: *l*: *list*(*A*) ==> *set-of-list*(*l*) : *Pow*(*A*)
apply (*unfold set-of-list-list-def*)
apply (*erule list-rec-type*, *auto*)
done

lemma *set-of-list-append*:
xs: *list*(*A*) ==> *set-of-list* (*xs*@*ys*) = *set-of-list*(*xs*) *Un set-of-list*(*ys*)
apply (*erule list.induct*)
apply (*simp-all* (*no-asm-simp*) *add*: *Un-cons*)
done

lemma *list-add-type* [*TC*]: *xs*: *list*(*nat*) ==> *list-add*(*xs*) : *nat*
by (*simp add*: *list-add-list-def*)

lemma *map-ident* [*simp*]: *l*: *list*(*A*) ==> *map*(%*u*. *u*, *l*) = *l*

apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *map-compose*: *l*: *list*(*A*) ==> *map*(*h*, *map*(*j*,*l*)) = *map*(%*u*. *h*(*j*(*u*)), *l*)
apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *map-app-distrib*: *xs*: *list*(*A*) ==> *map*(*h*, *xs*@*ys*) = *map*(*h*,*xs*) @ *map*(*h*,*ys*)
apply (*induct-tac xs*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *map-flat*: *ls*: *list*(*list*(*A*)) ==> *map*(*h*, *flat*(*ls*)) = *flat*(*map*(*map*(*h*),*ls*))
apply (*induct-tac ls*)
apply (*simp-all* (*no-asm-simp*) *add*: *map-app-distrib*)
done

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*)
apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*))
done

lemmas *list-CollectD* = *Collect-subset* [*THEN list-mono*, *THEN subsetD*, *standard*]

lemma *map-list-Collect*: *l*: *list*({*x*:*A*. *h*(*x*)=*j*(*x*)}) ==> *map*(*h*,*l*) = *map*(*j*,*l*)
apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *length-map* [*simp*]: *xs*: *list*(*A*) ==> *length*(*map*(*h*,*xs*)) = *length*(*xs*)
by (*induct-tac xs*, *simp-all*)

lemma *length-app* [*simp*]:
  [| *xs*: *list*(*A*); *ys*: *list*(*A*) |]
   ==> *length*(*xs*@*ys*) = *length*(*xs*) #+ *length*(*ys*)
by (*induct-tac xs*, *simp-all*)

lemma *length-rev* [*simp*]: *xs*: *list*(*A*) ==> *length*(*rev*(*xs*)) = *length*(*xs*)

apply (*induct-tac xs*)
apply (*simp-all* (*no-asm-simp*) *add*: *length-app*)
done

lemma *length-flat*:
ls: *list*(*list*(*A*)) ==> *length*(*flat*(*ls*)) = *list-add*(*map*(*length*,*ls*))
apply (*induct-tac ls*)
apply (*simp-all* (*no-asm-simp*) *add*: *length-app*)
done

lemma *drop-length-Cons* [*rule-format*]:
xs: *list*(*A*) ==>
∀ *x*. *EX z zs*. *drop*(*length*(*xs*), *Cons*(*x*,*xs*)) = *Cons*(*z*,*zs*)
by (*erule list.induct*, *simp-all*)

lemma *drop-length* [*rule-format*]:
l: *list*(*A*) ==> ∀ *i* ∈ *length*(*l*). (*EX z zs*. *drop*(*i*,*l*) = *Cons*(*z*,*zs*))
apply (*erule list.induct*, *simp-all*, *safe*)
apply (*erule drop-length-Cons*)
apply (*rule natE*)
apply (*erule Ord-trans* [*OF asm-rl length-type Ord-nat*], *assumption*, *simp-all*)
apply (*blast intro*: *succ-in-naturalD length-type*)
done

lemma *app-right-Nil* [*simp*]: *xs*: *list*(*A*) ==> *xs*@*Nil*=*xs*
by (*erule list.induct*, *simp-all*)

lemma *app-assoc*: *xs*: *list*(*A*) ==> (*xs*@*ys*)@*zs* = *xs*@(*ys*@*zs*)
by (*induct-tac xs*, *simp-all*)

lemma *flat-app-distrib*: *ls*: *list*(*list*(*A*)) ==> *flat*(*ls*@*ms*) = *flat*(*ls*)@*flat*(*ms*)
apply (*induct-tac ls*)
apply (*simp-all* (*no-asm-simp*) *add*: *app-assoc*)
done

lemma *rev-map-distrib*: *l*: *list*(*A*) ==> *rev*(*map*(*h*,*l*)) = *map*(*h*,*rev*(*l*))
apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*) *add*: *map-app-distrib*)
done

lemma *rev-app-distrib*:
  [| *xs*: *list*(*A*); *ys*: *list*(*A*) |] ==> *rev*(*xs*@*ys*) = *rev*(*ys*)@*rev*(*xs*)
apply (*erule list.induct*)
apply (*simp-all add*: *app-assoc*)
done

lemma *rev-rev-ident* [*simp*]: *l*: *list*(*A*) ==> *rev*(*rev*(*l*))=*l*
apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*) *add*: *rev-app-distrib*)
done

lemma *rev-flat*: *ls*: *list*(*list*(*A*)) ==> *rev*(*flat*(*ls*)) = *flat*(*map*(*rev*,*rev*(*ls*)))
apply (*induct-tac ls*)
apply (*simp-all add*: *map-app-distrib flat-app-distrib rev-app-distrib*)
done

lemma *list-add-app*:
  [| *xs*: *list*(*nat*); *ys*: *list*(*nat*) |]
   ==> *list-add*(*xs*@*ys*) = *list-add*(*ys*) #+ *list-add*(*xs*)
apply (*induct-tac xs*, *simp-all*)
done

lemma *list-add-rev*: *l*: *list*(*nat*) ==> *list-add*(*rev*(*l*)) = *list-add*(*l*)
apply (*induct-tac l*)
apply (*simp-all* (*no-asm-simp*) *add*: *list-add-app*)
done

lemma *list-add-flat*:
  ls: *list*(*list*(*nat*)) ==> *list-add*(*flat*(*ls*)) = *list-add*(*map*(*list-add*,*ls*))
apply (*induct-tac ls*)
apply (*simp-all* (*no-asm-simp*) *add*: *list-add-app*)
done

lemma *list-append-induct* [*case-names Nil snoc*, *consumes 1*]:
  [| *l*: *list*(*A*);
    P(*Nil*);
    !!*x y*. [| *x*: *A*; *y*: *list*(*A*); *P*(*y*) |] ==> *P*(*y* @ [*x*])
  |] ==> *P*(*l*)
apply (*subgoal-tac P*(*rev*(*rev*(*l*))), *simp*)
apply (*erule rev-type* [*THEN list.induct*], *simp-all*)
done

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)
apply (induct-tac n, simp)
apply (blast intro: ih elim!: leE)
done

theorem list-complete-induct:
      [| l ∈ list(A);
          ⋀l. [| l ∈ list(A);
                  ∀l' ∈ list(A). length(l') < length(l) --> P(l')|]
              ==> P(l)
       |] ==> P(l)
apply (rule list-complete-induct-lemma [of A])
   prefer 4 apply (rule le-refl, simp)
  apply blast
 apply simp
apply assumption
done
```

```
lemma min-sym: [| i:nat; j:nat |] ==> min(i,j)=min(j,i)
apply (unfold min-def)
apply (auto dest!: not-lt-imp-le dest: lt-not-sym intro: le-anti-sym)
done

lemma min-type [simp,TC]: [| i:nat; j:nat |] ==> min(i,j):nat
by (unfold min-def, auto)

lemma min-0 [simp]: i:nat ==> min(0,i) = 0
apply (unfold min-def)
apply (auto dest: not-lt-imp-le)
done

lemma min-02 [simp]: i:nat ==> min(i, 0) = 0
apply (unfold min-def)
apply (auto dest: not-lt-imp-le)
done

lemma lt-min-iff: [| i:nat; j:nat; k:nat |] ==> i<min(j,k) <-> i<j & i<k
apply (unfold min-def)
apply (auto dest!: not-lt-imp-le intro: lt-trans2 lt-trans)
done
```

lemma *min-succ-succ* [*simp*]:
  [| *i*:*nat*; *j*:*nat* |] ==> *min*(*succ*(*i*), *succ*(*j*))= *succ*(*min*(*i*, *j*))
apply (*unfold min-def*, *auto*)
done

lemma *filter-append* [*simp*]:
  xs:*list*(*A*) ==> *filter*(*P*, *xs*@*ys*) = *filter*(*P*, *xs*) @ *filter*(*P*, *ys*)
by (*induct-tac xs*, *auto*)

lemma *filter-type* [*simp*,*TC*]: *xs*:*list*(*A*) ==> *filter*(*P*, *xs*):*list*(*A*)
by (*induct-tac xs*, *auto*)

lemma *length-filter*: *xs*:*list*(*A*) ==> *length*(*filter*(*P*, *xs*)) *le length*(*xs*)
apply (*induct-tac xs*, *auto*)
apply (*rule-tac j* = *length* (*l*) **in** *le-trans*)
apply (*auto simp add*: *le-iff*)
done

lemma *filter-is-subset*: *xs*:*list*(*A*) ==> *set-of-list*(*filter*(*P*,*xs*)) <= *set-of-list*(*xs*)
by (*induct-tac xs*, *auto*)

lemma *filter-False* [*simp*]: *xs*:*list*(*A*) ==> *filter*(%*p*. *False*, *xs*) = *Nil*
by (*induct-tac xs*, *auto*)

lemma *filter-True* [*simp*]: *xs*:*list*(*A*) ==> *filter*(%*p*. *True*, *xs*) = *xs*
by (*induct-tac xs*, *auto*)

lemma *length-is-0-iff* [*simp*]: *xs*:*list*(*A*) ==> *length*(*xs*)=*0* <-> *xs*=*Nil*
by (*erule list*.*induct*, *auto*)

lemma *length-is-0-iff2* [*simp*]: *xs*:*list*(*A*) ==> *0* = *length*(*xs*) <-> *xs*=*Nil*
by (*erule list*.*induct*, *auto*)

lemma *length-tl* [*simp*]: *xs*:*list*(*A*) ==> *length*(*tl*(*xs*)) = *length*(*xs*) #− *1*
by (*erule list*.*induct*, *auto*)

lemma *length-greater-0-iff*: *xs*:*list*(*A*) ==> *0*<*length*(*xs*) <-> *xs* ~= *Nil*
by (*erule list*.*induct*, *auto*)

lemma *length-succ-iff*: *xs*:*list*(*A*) ==> *length*(*xs*)=*succ*(*n*) <-> (*EX y ys*. *xs*=*Cons*(*y*, *ys*) & *length*(*ys*)=*n*)
by (*erule list*.*induct*, *auto*)

lemma append-is-Nil-iff [simp]:
xs:list(A) ==> (xs@ys = Nil) <-> (xs=Nil & ys = Nil)
by (erule list.induct, auto)

lemma append-is-Nil-iff2 [simp]:
xs:list(A) ==> (Nil = xs@ys) <-> (xs=Nil & ys = Nil)
by (erule list.induct, auto)

lemma append-left-is-self-iff [simp]:
xs:list(A) ==> (xs@ys = xs) <-> (ys = Nil)
by (erule list.induct, auto)

lemma append-left-is-self-iff2 [simp]:
xs:list(A) ==> (xs = xs@ys) <-> (ys = Nil)
by (erule list.induct, auto)

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))
apply (erule list.induct)
apply (auto simp add: length-app)
done

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))
apply (erule list.induct)
apply (auto simp add: length-app)
done

lemma append-eq-append-iff [rule-format,simp]:
xs:list(A) ==> ∀ ys ∈ list(A).
length(xs)=length(ys) --> (xs@us = ys@vs) <-> (xs=ys & us=vs)
apply (erule list.induct)
apply (simp (no-asm-simp))
apply clarify
apply (erule-tac a = ys **in** list.cases, auto)
done

lemma append-eq-append [rule-format]:
xs:list(A) ==>
∀ ys ∈ list(A). ∀ us ∈ list(A). ∀ vs ∈ list(A).
length(us) = length(vs) --> (xs@us = ys@vs) --> (xs=ys & us=vs)
apply (induct-tac xs)
apply (force simp add: length-app, clarify)

apply (*erule-tac a = ys* **in** *list.cases*, *simp*)
apply (*subgoal-tac Cons* (*a*, *l*) @ *us* =*vs*)
apply (*drule rev-iffD1* [*OF* - *append-left-is-Nil-iff*], *simp-all*, *blast*)
done

lemma *append-eq-append-iff2* [*simp*]:
[| *xs*:*list*(*A*); *ys*:*list*(*A*); *us*:*list*(*A*); *vs*:*list*(*A*); *length*(*us*)=*length*(*vs*) |]
==> *xs*@*us* = *ys*@*vs* <-> (*xs*=*ys* & *us*=*vs*)
apply (*rule iffI*)
apply (*rule append-eq-append*, *auto*)
done

lemma *append-self-iff* [*simp*]:
[| *xs*:*list*(*A*); *ys*:*list*(*A*); *zs*:*list*(*A*) |] ==> *xs*@*ys*=*xs*@*zs* <-> *ys*=*zs*
by *simp*

lemma *append-self-iff2* [*simp*]:
[| *xs*:*list*(*A*); *ys*:*list*(*A*); *zs*:*list*(*A*) |] ==> *ys*@*xs*=*zs*@*xs* <-> *ys*=*zs*
by *simp*

lemma *append1-eq-iff* [*rule-format*,*simp*]:
xs:*list*(*A*) ==> ∀ *ys* ∈ *list*(*A*). *xs*@[*x*] = *ys*@[*y*] <-> (*xs* = *ys* & *x*=*y*)
apply (*erule list.induct*)
apply *clarify*
apply (*erule list.cases*)
apply *simp-all*

Inductive step

apply *clarify*
apply (*erule-tac a*=*ys* **in** *list.cases*, *simp-all*)
done

lemma *append-right-is-self-iff* [*simp*]:
[| *xs*:*list*(*A*); *ys*:*list*(*A*) |] ==> (*xs*@*ys* = *ys*) <-> (*xs*=*Nil*)
by (*simp* (*no-asm-simp*) *add*: *append-left-is-Nil-iff*)

lemma *append-right-is-self-iff2* [*simp*]:
[| *xs*:*list*(*A*); *ys*:*list*(*A*) |] ==> (*ys* = *xs*@*ys*) <-> (*xs*=*Nil*)
apply (*rule iffI*)
apply (*drule sym*, *auto*)
done

lemma *hd-append* [*rule-format*,*simp*]:
xs:*list*(*A*) ==> *xs* ~= *Nil* --> *hd*(*xs* @ *ys*) = *hd*(*xs*)
by (*induct-tac xs*, *auto*)

lemma *tl-append* [*rule-format*,*simp*]:

xs:*list*(*A*) ==> *xs*~=*Nil* --> *tl*(*xs* @ *ys*) = *tl*(*xs*)@*ys*
by (*induct-tac xs*, *auto*)

lemma *rev-is-Nil-iff* [*simp*]: *xs*:*list*(*A*) ==> (*rev*(*xs*) = *Nil* <-> *xs* = *Nil*)
by (*erule list*.*induct*, *auto*)

lemma *Nil-is-rev-iff* [*simp*]: *xs*:*list*(*A*) ==> (*Nil* = *rev*(*xs*) <-> *xs* = *Nil*)
by (*erule list*.*induct*, *auto*)

lemma *rev-is-rev-iff* [*rule-format*,*simp*]:
xs:*list*(*A*) ==> ∀ *ys* ∈ *list*(*A*). *rev*(*xs*)=*rev*(*ys*) <-> *xs*=*ys*
apply (*erule list*.*induct*, *force*, *clarify*)
apply (*erule-tac a* = *ys* **in** *list*.*cases*, *auto*)
done

lemma *rev-list-elim* [*rule-format*]:
xs:*list*(*A*) ==>
(*xs*=*Nil* --> *P*) --> (∀ *ys* ∈ *list*(*A*). ∀ *y* ∈ *A*. *xs* =*ys*@[*y*] -->*P*)-->*P*
by (*erule list-append-induct*, *auto*)

lemma *length-drop* [*rule-format*,*simp*]:
n:*nat* ==> ∀ *xs* ∈ *list*(*A*). *length*(*drop*(*n*, *xs*)) = *length*(*xs*) #- *n*
apply (*erule nat-induct*)
apply (*auto elim*: *list*.*cases*)
done

lemma *drop-all* [*rule-format*,*simp*]:
n:*nat* ==> ∀ *xs* ∈ *list*(*A*). *length*(*xs*) *le n* --> *drop*(*n*, *xs*)=*Nil*
apply (*erule nat-induct*)
apply (*auto elim*: *list*.*cases*)
done

lemma *drop-append* [*rule-format*]:
n:*nat* ==>
∀ *xs* ∈ *list*(*A*). *drop*(*n*, *xs*@*ys*) = *drop*(*n*,*xs*) @ *drop*(*n* #- *length*(*xs*), *ys*)
apply (*induct-tac n*)
apply (*auto elim*: *list*.*cases*)
done

lemma *drop-drop*:
m:*nat* ==> ∀ *xs* ∈ *list*(*A*). ∀ *n* ∈ *nat*. *drop*(*n*, *drop*(*m*, *xs*))=*drop*(*n* #+ *m*, *xs*)
apply (*induct-tac m*)
apply (*auto elim*: *list*.*cases*)
done

```
lemma take-0 [simp]: xs:list(A) ==> take(0, xs) =  Nil
apply (unfold take-def)
apply (erule list.induct, auto)
done

lemma take-succ-Cons [simp]:
    n:nat ==> take(succ(n), Cons(a, xs)) = Cons(a, take(n, xs))
by (simp add: take-def)


lemma take-Nil [simp]: n:nat ==> take(n, Nil) = Nil
by (unfold take-def, auto)

lemma take-all [rule-format,simp]:
     n:nat ==> ∀ xs ∈ list(A). length(xs) le n  --> take(n, xs) = xs
apply (erule nat-induct)
apply (auto elim: list.cases)
done

lemma take-type [rule-format,simp,TC]:
     xs:list(A) ==> ∀ n ∈ nat. take(n, xs):list(A)
apply (erule list.induct, simp, clarify)
apply (erule natE, auto)
done

lemma take-append [rule-format,simp]:
 xs:list(A) ==>
  ∀ ys ∈ list(A). ∀ n ∈ nat. take(n, xs @ ys) =
                            take(n, xs) @ take(n #- length(xs), ys)
apply (erule list.induct, simp, clarify)
apply (erule natE, auto)
done

lemma take-take [rule-format]:
   m : nat ==>
    ∀ xs ∈ list(A). ∀ n ∈ nat. take(n, take(m,xs))= take(min(n, m), xs)
apply (induct-tac m, auto)
apply (erule-tac a = xs in list.cases)
apply (auto simp add: take-Nil)
apply (erule-tac n=n in natE)
apply (auto intro: take-0 take-type)
done



lemma nth-0 [simp]: nth(0, Cons(a, l)) = a
```

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

lemma *nth-Cons* [*simp*]: *n*:*nat* ==> *nth*(*succ*(*n*), *Cons*(*a*,*l*)) = *nth*(*n*,*l*)
by (*simp add*: *nth-def*)

lemma *nth-empty* [*simp*]: *nth*(*n*, *Nil*) = *0*
by (*simp add*: *nth-def*)

lemma *nth-type* [*rule-format*,*simp*,*TC*]:
    xs:*list*(*A*) ==> ∀ *n*. *n* < *length*(*xs*) −−> *nth*(*n*,*xs*) : *A*
apply (*erule list*.*induct*, *simp*, *clarify*)
apply (*subgoal-tac n* ∈ *nat*)
 apply (*erule natE*, *auto dest*!: *le-in-nat*)
done

lemma *nth-eq-0* [*rule-format*]:
    xs:*list*(*A*) ==> ∀ *n* ∈ *nat*. *length*(*xs*) *le n* −−> *nth*(*n*,*xs*) = *0*
apply (*erule list*.*induct*, *simp*, *clarify*)
apply (*erule natE*, *auto*)
done

lemma *nth-append* [*rule-format*]:
  xs:*list*(*A*) ==>
   ∀ *n* ∈ *nat*. *nth*(*n*, *xs* @ *ys*) = (*if n* < *length*(*xs*) *then nth*(*n*,*xs*)
                                  else nth(*n* #− *length*(*xs*), *ys*))
apply (*induct-tac xs*, *simp*, *clarify*)
apply (*erule natE*, *auto*)
done

lemma *set-of-list-conv-nth*:
   xs:*list*(*A*)
    ==> *set-of-list*(*xs*) = {*x*:*A*. *EX i*:*nat*. *i*<*length*(*xs*) & *x* = *nth*(*i*,*xs*)}
apply (*induct-tac xs*, *simp-all*)
apply (*rule equalityI*, *auto*)
apply (*rule-tac x* = *0* **in** *bexI*, *auto*)
apply (*erule natE*, *auto*)
done

lemma *nth-take-lemma* [*rule-format*]:
 k:*nat* ==>
  ∀ *xs* ∈ *list*(*A*). (∀ *ys* ∈ *list*(*A*). *k le length*(*xs*) −−> *k le length*(*ys*) −−>
     (∀ *i* ∈ *nat*. *i*<*k* −−> *nth*(*i*,*xs*) = *nth*(*i*,*ys*))−−> *take*(*k*,*xs*) = *take*(*k*,*ys*))
apply (*induct-tac k*)
apply (*simp-all* (*no-asm-simp*) *add*: *lt-succ-eq-0-disj all-conj-distrib*)
apply *clarify*

apply (*erule-tac a*=*xs* **in** *list*.*cases*, *simp*)

apply (*erule-tac a=ys* **in** *list.cases*, *clarify*)
apply (*simp* (*no-asm-use*))
apply *clarify*
apply (*simp* (*no-asm-simp*))
apply (*rule conjI*, *force*)
apply (*rename-tac y ys z zs*)
apply (*drule-tac x = zs* **and** *x1 = ys* **in** *bspec* [*THEN bspec*], *auto*)
done

lemma *nth-equalityI* [*rule-format*]:
[| *xs*:*list*(*A*); *ys*:*list*(*A*); *length*(*xs*) = *length*(*ys*);
∀ *i* ∈ *nat*. *i* < *length*(*xs*) −−> *nth*(*i*,*xs*) = *nth*(*i*,*ys*) |]
==> *xs* = *ys*
apply (*subgoal-tac length* (*xs*) *le length* (*ys*))
apply (*cut-tac k*=*length*(*xs*) **and** *xs*=*xs* **and** *ys*=*ys* **in** *nth-take-lemma*)
apply (*simp-all add*: *take-all*)
done

lemma *take-equalityI* [*rule-format*]:
[| *xs*:*list*(*A*); *ys*:*list*(*A*); (∀ *i* ∈ *nat*. *take*(*i*, *xs*) = *take*(*i*,*ys*)) |]
==> *xs* = *ys*
apply (*case-tac length* (*xs*) *le length* (*ys*))
apply (*drule-tac x* = *length* (*ys*) **in** *bspec*)
apply (*drule-tac* [*3*] *not-lt-imp-le*)
apply (*subgoal-tac* [*5*] *length* (*ys*) *le length* (*xs*))
apply (*rule-tac* [*6*] *j* = *succ* (*length* (*ys*)) **in** *le-trans*)
apply (*rule-tac* [*6*] *leI*)
apply (*drule-tac* [*5*] *x* = *length* (*xs*) **in** *bspec*)
apply (*simp-all add*: *take-all*)
done

lemma *nth-drop* [*rule-format*]:
n:*nat* ==> ∀ *i* ∈ *nat*. ∀ *xs* ∈ *list*(*A*). *nth*(*i*, *drop*(*n*, *xs*)) = *nth*(*n* #+ *i*, *xs*)
apply (*induct-tac n*, *simp-all*, *clarify*)
apply (*erule list.cases*, *auto*)
done

lemma *take-succ* [*rule-format*]:
xs∈*list*(*A*)
==> ∀ *i*. *i* < *length*(*xs*) −−> *take*(*succ*(*i*), *xs*) = *take*(*i*,*xs*) @ [*nth*(*i*, *xs*)]
apply (*induct-tac xs*, *auto*)
apply (*subgoal-tac i*∈*nat*)
apply (*erule natE*)
apply (*auto simp add*: *le-in-nat*)
done

lemma *take-add* [*rule-format*]:

[|xs∈list(A); j∈nat|]
==> ∀ i∈nat. take(i #+ j, xs) = take(i,xs) @ take(j, drop(i,xs))
apply (induct-tac xs, simp-all, clarify)
apply (erule-tac n = i **in** natE, simp-all)
done

lemma length-take:
l∈list(A) ==> ∀ n∈nat. length(take(n,l)) = min(n, length(l))
apply (induct-tac l, safe, simp-all)
apply (erule natE, simp-all)
done

28.1 The function zip

Crafty definition to eliminate a type argument

consts
zip-aux :: [i,i]=>i

primrec
zip-aux(B,[]) =
(λys ∈ list(B). list-case([], %y l. [], ys))

zip-aux(B,Cons(x,l)) =
(λys ∈ list(B).
list-case(Nil, %y zs. Cons(<x,y>, zip-aux(B,l)‘zs), ys))

definition
zip :: [i, i]=>i **where**
zip(xs, ys) == zip-aux(set-of-list(ys),xs)‘ys

lemma list-on-set-of-list: xs ∈ list(A) ==> xs ∈ list(set-of-list(xs))
apply (induct-tac xs, simp-all)
apply (blast intro: list-mono [THEN subsetD])
done

lemma zip-Nil [simp]: ys:list(A) ==> zip(Nil, ys)=Nil
apply (simp add: zip-def list-on-set-of-list [of - A])
apply (erule list.cases, simp-all)
done

lemma zip-Nil2 [simp]: xs:list(A) ==> zip(xs, Nil)=Nil
apply (simp add: zip-def list-on-set-of-list [of - A])
apply (erule list.cases, simp-all)
done

lemma zip-aux-unique [rule-format]:

[|B<=C; xs ∈ list(A)|]
==> ∀ ys ∈ list(B). zip-aux(C,xs) ‘ ys = zip-aux(B,xs) ‘ ys
apply (induct-tac xs)
apply simp-all
apply (blast intro: list-mono [THEN subsetD], clarify)
apply (erule-tac a=ys **in** list.cases, auto)
apply (blast intro: list-mono [THEN subsetD])
done

lemma zip-Cons-Cons [simp]:
[| xs:list(A); ys:list(B); x:A; y:B |] ==>
zip(Cons(x,xs), Cons(y, ys)) = Cons(<x,y>, zip(xs, ys))
apply (simp add: zip-def, auto)
apply (rule zip-aux-unique, auto)
apply (simp add: list-on-set-of-list [of - B])
apply (blast intro: list-on-set-of-list list-mono [THEN subsetD])
done

lemma zip-type [rule-format,simp,TC]:
xs:list(A) ==> ∀ ys ∈ list(B). zip(xs, ys):list(A*B)
apply (induct-tac xs)
apply (simp (no-asm))
apply clarify
apply (erule-tac a = ys **in** list.cases, auto)
done

lemma length-zip [rule-format,simp]:
xs:list(A) ==> ∀ ys ∈ list(B). length(zip(xs,ys)) =
min(length(xs), length(ys))
apply (unfold min-def)
apply (induct-tac xs, simp-all, clarify)
apply (erule-tac a = ys **in** list.cases, auto)
done

lemma zip-append1 [rule-format]:
[| ys:list(A); zs:list(B) |] ==>
∀ xs ∈ list(A). zip(xs @ ys, zs) =
zip(xs, take(length(xs), zs)) @ zip(ys, drop(length(xs),zs))
apply (induct-tac zs, force, clarify)
apply (erule-tac a = xs **in** list.cases, simp-all)
done

lemma zip-append2 [rule-format]:
[| xs:list(A); zs:list(B) |] ==> ∀ ys ∈ list(B). zip(xs, ys@zs) =
zip(take(length(ys), xs), ys) @ zip(drop(length(ys), xs), zs)
apply (induct-tac xs, force, clarify)
apply (erule-tac a = ys **in** list.cases, auto)
done

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*)
by (*simp* (*no-asm-simp*) *add*: *zip-append1 drop-append diff-self-eq-0*)

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*))
apply (*induct-tac ys*, *force*, *clarify*)
apply (*erule-tac a* = *xs* **in** *list.cases*)
apply (*auto simp add*: *length-rev*)
done

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*)>
apply (*induct-tac ys*, *force*, *clarify*)
apply (*erule-tac a* = *xs* **in** *list.cases*, *simp*)
apply (*auto elim*: *natE*)
done

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*)}
by (*force intro*!: *Collect-cong simp add*: *lt-min-iff set-of-list-conv-nth*)

lemma *list-update-Nil* [*simp*]: *i*:*nat* ==>*list-update*(*Nil*, *i*, *v*) = *Nil*
by (*unfold list-update-def*, *auto*)

lemma *list-update-Cons-0* [*simp*]: *list-update*(*Cons*(*x*, *xs*), *0*, *v*)= *Cons*(*v*, *xs*)
by (*unfold list-update-def*, *auto*)

lemma *list-update-Cons-succ* [*simp*]:
n:*nat* ==>
list-update(*Cons*(*x*, *xs*), *succ*(*n*), *v*)= *Cons*(*x*, *list-update*(*xs*, *n*, *v*))
apply (*unfold list-update-def*, *auto*)
done

lemma *list-update-type* [*rule-format*,*simp*,*TC*]:
[| *xs*:*list*(*A*); *v*:*A* |] ==> ∀ *n* ∈ *nat*. *list-update*(*xs*, *n*, *v*):*list*(*A*)
apply (*induct-tac xs*)

```
apply (simp (no-asm))
apply clarify
apply (erule natE, auto)
done

lemma length-list-update [rule-format,simp]:
    xs:list(A) ==> ∀ i ∈ nat. length(list-update(xs, i, v))=length(xs)
apply (induct-tac xs)
apply (simp (no-asm))
apply clarify
apply (erule natE, auto)
done

lemma nth-list-update [rule-format]:
    [| xs:list(A) |] ==> ∀ i ∈ nat. ∀ j ∈ nat. i < length(xs)  -->
        nth(j, list-update(xs, i, x)) = (if i=j then x else nth(j, xs))
apply (induct-tac xs)
 apply simp-all
apply clarify
apply (rename-tac i j)
apply (erule-tac n=i in natE)
apply (erule-tac [2] n=j in natE)
apply (erule-tac n=j in natE, simp-all, force)
done

lemma nth-list-update-eq [simp]:
    [| i < length(xs); xs:list(A) |] ==> nth(i, list-update(xs, i,x)) = x
by (simp (no-asm-simp) add: lt-length-in-nat nth-list-update)


lemma nth-list-update-neq [rule-format,simp]:
  xs:list(A) ==>
    ∀ i ∈ nat. ∀ j ∈ nat. i ~= j --> nth(j, list-update(xs,i,x)) = nth(j,xs)
apply (induct-tac xs)
 apply (simp (no-asm))
apply clarify
apply (erule natE)
apply (erule-tac [2] natE, simp-all)
apply (erule natE, simp-all)
done

lemma list-update-overwrite [rule-format,simp]:
    xs:list(A) ==> ∀ i ∈ nat. i < length(xs)
   --> list-update(list-update(xs, i, x), i, y) = list-update(xs, i,y)
apply (induct-tac xs)
 apply (simp (no-asm))
apply clarify
apply (erule natE, auto)
done
```

```
lemma list-update-same-conv [rule-format]:
     xs:list(A) ==>
      ∀ i ∈ nat. i < length(xs) -->
               (list-update(xs, i, x) = xs) <-> (nth(i, xs) = x)
apply (induct-tac xs)
 apply (simp (no-asm))
apply clarify
apply (erule natE, auto)
done

lemma update-zip [rule-format]:
     ys:list(B) ==>
      ∀ i ∈ nat. ∀ xy ∈ A*B. ∀ xs ∈ list(A).
        length(xs) = length(ys) -->
        list-update(zip(xs, ys), i, xy) = zip(list-update(xs, i, fst(xy)),
                                              list-update(ys, i, snd(xy)))
apply (induct-tac ys)
 apply auto
apply (erule-tac a = xs in list.cases)
apply (auto elim: natE)
done

lemma set-update-subset-cons [rule-format]:
  xs:list(A) ==>
   ∀ i ∈ nat. set-of-list(list-update(xs, i, x)) <= cons(x, set-of-list(xs))
apply (induct-tac xs)
 apply simp
apply (rule ballI)
apply (erule natE, simp-all, auto)
done

lemma set-of-list-update-subsetI:
     [| set-of-list(xs) <= A; xs:list(A); x:A; i:nat|]
   ==> set-of-list(list-update(xs, i,x)) <= A
apply (rule subset-trans)
apply (rule set-update-subset-cons, auto)
done


lemma upt-rec:
     j:nat ==> upt(i,j) = (if i<j then Cons(i, upt(succ(i), j)) else Nil)
apply (induct-tac j, auto)
apply (drule not-lt-imp-le)
apply (auto simp: lt-Ord intro: le-anti-sym)
done

lemma upt-conv-Nil [simp]: [| j le i; j:nat |] ==> upt(i,j) = Nil
```

apply (*subst upt-rec*, *auto*)
apply (*auto simp add*: *le-iff*)
apply (*drule lt-asym* [*THEN notE*], *auto*)
done

lemma *upt-succ-append*:
[| *i le j*; *j*:*nat* |] ==> *upt*(*i*,*succ*(*j*)) = *upt*(*i*, *j*)@[*j*]
by *simp*

lemma *upt-conv-Cons*:
[| *i*<*j*; *j*:*nat* |] ==> *upt*(*i*,*j*) = *Cons*(*i*,*upt*(*succ*(*i*),*j*))
apply (*rule trans*)
apply (*rule upt-rec*, *auto*)
done

lemma *upt-type* [*simp*,*TC*]: *j*:*nat* ==> *upt*(*i*,*j*):*list*(*nat*)
by (*induct-tac j*, *auto*)

lemma *upt-add-eq-append*:
[| *i le j*; *j*:*nat*; *k*:*nat* |] ==> *upt*(*i*, *j* #+*k*) = *upt*(*i*,*j*)@*upt*(*j*,*j*#+*k*)
apply (*induct-tac k*)
apply (*auto simp add*: *app-assoc app-type*)
apply (*rule-tac j* = *j* **in** *le-trans*, *auto*)
done

lemma *length-upt* [*simp*]: [| *i*:*nat*; *j*:*nat* |] ==>*length*(*upt*(*i*,*j*)) = *j* #− *i*
apply (*induct-tac j*)
apply (*rule-tac* [*2*] *sym*)
apply (*auto dest*!: *not-lt-imp-le simp add*: *diff-succ diff-is-0-iff*)
done

lemma *nth-upt* [*rule-format*,*simp*]:
[| *i*:*nat*; *j*:*nat*; *k*:*nat* |] ==> *i* #+ *k* < *j* −−> *nth*(*k*, *upt*(*i*,*j*)) = *i* #+ *k*
apply (*induct-tac j*, *simp*)
apply (*simp add*: *nth-append le-iff*)
apply (*auto dest*!: *not-lt-imp-le*
simp add: *nth-append less-diff-conv add-commute*)
done

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*)
apply (*induct-tac m*)
apply (*simp* (*no-asm-simp*) *add*: *take-0*)
apply *clarify*
apply (*subst upt-rec*, *simp*)
apply (*rule sym*)

```
apply (subst upt-rec, simp)
apply (simp-all del: upt.simps)
apply (rule-tac j = succ (i #+ x)  in lt-trans2)
apply auto
done

lemma map-succ-upt:
     [| m:nat; n:nat |] ==> map(succ, upt(m,n))= upt(succ(m), succ(n))
apply (induct-tac n)
apply (auto simp add: map-app-distrib)
done

lemma nth-map [rule-format,simp]:
     xs:list(A) ==>
      ∀ n ∈ nat. n < length(xs) --> nth(n, map(f, xs)) = f(nth(n, xs))
apply (induct-tac xs, simp)
apply (rule ballI)
apply (induct-tac n, auto)
done

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)
apply (rule-tac n = m and m = n in diff-induct, typecheck, simp, simp)
apply (subst map-succ-upt [symmetric], simp-all, clarify)
apply (subgoal-tac i < length (upt (0, x)))
 prefer 2
 apply (simp add: less-diff-conv)
 apply (rule-tac j = succ (i #+ y)  in lt-trans2)
  apply simp
 apply simp
apply (subgoal-tac i < length (upt (y, x)))
 apply (simp-all add: add-commute less-diff-conv)
done



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
by (unfold sublist-def, auto)

lemma sublist-Nil [simp]: sublist(Nil, A) = Nil
by (unfold sublist-def, auto)

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)))))
apply (erule list-append-induct)
apply (simp (no-asm-simp))
apply (auto simp add: add-commute length-app filter-append map-app-distrib)
done

lemma sublist-type [simp,TC]:
     xs:list(B) ==> sublist(xs, A):list(B)
apply (unfold sublist-def)
apply (induct-tac xs)
apply (auto simp add: filter-append map-app-distrib)
done

lemma upt-add-eq-append2:
     [| i:nat; j:nat |] ==> upt(0, i #+ j) = upt(0, i) @ upt(i, i #+ j)
by (simp add: upt-add-eq-append [of 0] nat-0-le)

lemma sublist-append:
 [| xs:list(B); ys:list(B)  |] ==>
  sublist(xs@ys, A) = sublist(xs, A) @ sublist(ys, {j:nat. j #+ length(xs): A})
apply (unfold sublist-def)
apply (erule-tac l = ys in list-append-induct, simp)
apply (simp (no-asm-simp) add: upt-add-eq-append2 app-assoc [symmetric])
apply (auto simp add: sublist-shift-lemma length-type map-app-distrib app-assoc)
apply (simp-all add: add-commute)
done


lemma sublist-Cons:
     [| xs:list(B); x:B |] ==>
      sublist(Cons(x, xs), A) =
      (if 0:A then [x] else []) @ sublist(xs, {j:nat. succ(j) : A})
apply (erule-tac l = xs in list-append-induct)
apply (simp (no-asm-simp) add: sublist-def)
apply (simp del: app-Cons add: app-Cons [symmetric] sublist-append, simp)
done

lemma sublist-singleton [simp]:
     sublist([x], A) = (if 0 : A then [x] else [])
by (simp add: sublist-Cons)

lemma sublist-upt-eq-take [rule-format, simp]:
    xs:list(A) ==> ALL n:nat. sublist(xs,n) = take(n,xs)
apply (erule list.induct, simp)
apply (clarify )
apply (erule natE)
apply (simp-all add: nat-eq-Collect-lt Ord-mem-iff-lt sublist-Cons)
```

done

lemma *sublist-Int-eq*:
xs : *list*(*B*) ==> *sublist*(*xs*, *A* ∩ *nat*) = *sublist*(*xs*, *A*)
apply (*erule list.induct*)
apply (*simp-all add*: *sublist-Cons*)
done

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*
by (*induct-tac n*, *auto*)

lemma *repeat-succ-app*: *n* ∈ *nat* ==> *repeat*(*a*,*succ*(*n*)) = *repeat*(*a*,*n*) @ [*a*]
apply (*induct-tac n*)
apply (*simp-all del*: *app-Cons add*: *app-Cons* [*symmetric*])
done

lemma *repeat-type* [*TC*]: [|*a* ∈ *A*; *n* ∈ *nat*|] ==> *repeat*(*a*,*n*) ∈ *list*(*A*)
by (*induct-tac n*, *auto*)

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*
by (*unfold trans-def sym-def*, *blast*)

lemma *refl-comp-subset*:
[| *refl*(*A*,*r*); *r* <= *A*∗*A* |] ==> *r* <= *converse*(*r*) *O r*
by (*unfold refl-def*, *blast*)

lemma *equiv-comp-eq*:
equiv(*A*,*r*) ==> *converse*(*r*) *O r* = *r*
apply (*unfold equiv-def*)
apply (*blast del*: *subsetI intro*!: *sym-trans-comp-subset refl-comp-subset*)
done

lemma *comp-equivI*:
[| *converse*(*r*) *O r* = *r*; *domain*(*r*) = *A* |] ==> *equiv*(*A*,*r*)
apply (*unfold equiv-def refl-def sym-def trans-def*)
apply (*erule equalityE*)
apply (*subgoal-tac ALL x y*. <*x*,*y*> : *r* −−> <*y*,*x*> : *r*, *blast*+)
done

lemma *equiv-class-subset*:
[| *sym*(*r*); *trans*(*r*); <*a*,*b*>: *r* |] ==> *r*‘‘{*a*} <= *r*‘‘{*b*}
by (*unfold trans-def sym-def*, *blast*)

lemma *equiv-class-eq*:
[| *equiv*(*A*,*r*); <*a*,*b*>: *r* |] ==> *r*‘‘{*a*} = *r*‘‘{*b*}
apply (*unfold equiv-def*)
apply (*safe del*: *subsetI intro*!: *equalityI equiv-class-subset*)
apply (*unfold sym-def*, *blast*)
done

lemma *equiv-class-self*:
[| *equiv*(*A*,*r*); *a*: *A* |] ==> *a*: *r*‘‘{*a*}
by (*unfold equiv-def refl-def*, *blast*)

```
lemma subset-equiv-class:
    [| equiv(A,r);  r``{b} <= r``{a};  b: A |] ==> <a,b>: r
by (unfold equiv-def refl-def, blast)

lemma eq-equiv-class: [| r``{a} = r``{b};  equiv(A,r);  b: A |] ==> <a,b>: r
by (assumption | rule equalityD2 subset-equiv-class)+


lemma equiv-class-nondisjoint:
    [| equiv(A,r);  x: (r``{a} Int r``{b}) |] ==> <a,b>: r
by (unfold equiv-def trans-def sym-def, blast)

lemma equiv-type: equiv(A,r) ==> r <= A*A
by (unfold equiv-def, blast)

lemma equiv-class-eq-iff:
     equiv(A,r) ==> <x,y>: r <-> r``{x} = r``{y} & x:A & y:A
by (blast intro: eq-equiv-class equiv-class-eq dest: equiv-type)

lemma eq-equiv-class-iff:
     [| equiv(A,r);  x: A;  y: A |] ==> r``{x} = r``{y} <-> <x,y>: r
by (blast intro: eq-equiv-class equiv-class-eq dest: equiv-type)


lemma quotientI [TC]: x:A ==> r``{x}: A//r
apply (unfold quotient-def)
apply (erule RepFunI)
done

lemma quotientE:
    [| X: A//r;  !!x. [| X = r``{x};  x:A |] ==> P |] ==> P
by (unfold quotient-def, blast)

lemma Union-quotient:
    equiv(A,r) ==> Union(A//r) = A
by (unfold equiv-def refl-def quotient-def, blast)

lemma quotient-disj:
    [| equiv(A,r);  X: A//r;  Y: A//r |] ==> X=Y | (X Int Y <= 0)
apply (unfold quotient-def)
apply (safe intro!: equiv-class-eq, assumption)
apply (unfold equiv-def trans-def sym-def, blast)
done
```

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*)
apply (*subgoal-tac* ∀ *x* ∈ *r*``{*a*}. *b*(*x*) = *b*(*a*))
 apply *simp*
 apply (*blast intro*: *equiv-class-self*)
apply (*unfold equiv-def sym-def congruent-def*, *blast*)
done

lemma *UN-equiv-class-type*:
  [| *equiv*(*A*,*r*); *b respects r*; *X*: *A*//*r*; !!*x*. *x* : *A* ==> *b*(*x*) : *B* |]
   ==> (*UN x*:*X*. *b*(*x*)) : *B*
apply (*unfold quotient-def*, *safe*)
apply (*simp* (*no-asm-simp*) *add*: *UN-equiv-class*)
done

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*
apply (*unfold quotient-def*, *safe*)
apply (*rule equiv-class-eq*, *assumption*)
apply (*simp add*: *UN-equiv-class* [*of A r b*])
done

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*))
by (*unfold congruent-def congruent2-def equiv-def refl-def*, *blast*)

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*))
apply (*unfold congruent-def*, *safe*)
apply (*frule equiv-type* [*THEN subsetD*], *assumption*)
apply *clarify*
apply (*simp add*: *UN-equiv-class congruent2-implies-congruent*)
apply (*unfold congruent2-def equiv-def refl-def*, *blast*)
done

lemma *UN-equiv-class2*:
  [| *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*)
by (*simp add*: *UN-equiv-class congruent2-implies-congruent*
      congruent2-implies-congruent-UN)

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*
apply (*unfold quotient-def*, *safe*)
apply (*blast intro*: *UN-equiv-class-type congruent2-implies-congruent-UN*
      congruent2-implies-congruent quotientI)
done

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*)
apply (*unfold congruent2-def equiv-def refl-def*, *safe*)
apply (*blast intro*: *trans*)
done

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*
apply (*insert equivA* [*THEN equiv-type*, *THEN subsetD*])
apply (*rule congruent2I* [*OF equivA equivA*])
apply (*rule commute* [*THEN trans*])
apply (*rule-tac* [*3*] *commute* [*THEN trans*, *symmetric*])
apply (*rule-tac* [*5*] *sym*)
apply (*blast intro*: *congt*)+
done

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*))
apply (*simp* (*no-asm*) *add*: *congruent-def*)
apply (*safe elim*!: *quotientE*)
apply (*frule equiv-type* [*THEN subsetD*], *assumption*)
apply (*simp add*: *UN-equiv-class* [*of A r*])
apply (*simp add*: *congruent-def*)
done

end

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) ==
⋃*p1∈z1*. ⋃*p2∈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*) ==
⋃*z1∈z1*. ⋃*z2∈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∈*nat* & *y1*∈*nat* & *x2*∈*nat* & *y2*∈*nat* & *x1*#+*y2* = *x2*#+*y1*
by (*simp add*: *intrel-def*)

lemma *intrelI* [*intro*!]:
[| *x1*#+*y2* = *x2*#+*y1*; *x1*∈*nat*; *y1*∈*nat*; *x2*∈*nat*; *y2*∈*nat* |]
==> <<*x1*,*y1*>,<*x2*,*y2*>>: *intrel*
by (*simp add*: *intrel-def*)

lemma *intrelE* [*elim*!]:
[| *p*: *intrel*;
!!*x1 y1 x2 y2*. [| *p* = <<*x1*,*y1*>,<*x2*,*y2*>>; *x1*#+*y2* = *x2*#+*y1*;
x1∈*nat*; *y1*∈*nat*; *x2*∈*nat*; *y2*∈*nat* |] ==> *Q* |]
==> *Q*
by (*simp add*: *intrel-def*, *blast*)

lemma *int-trans-lemma*:
[| *x1* #+ *y2* = *x2* #+ *y1*; *x2* #+ *y3* = *x3* #+ *y2* |] ==> *x1* #+ *y3* = *x3* #+ *y1*
apply (*rule sym*)
apply (*erule add-left-cancel*)+
apply (*simp-all* (*no-asm-simp*))
done

lemma *equiv-intrel*: *equiv*(*nat*∗*nat*, *intrel*)
apply (*simp add*: *equiv-def refl-def sym-def trans-def*)
apply (*fast elim*!: *sym int-trans-lemma*)
done

lemma *image-intrel-int*: [| *m*∈*nat*; *n*∈*nat* |] ==> *intrel* `` {<*m*,*n*>} : *int*
by (*simp add*: *int-def*)

declare *equiv-intrel* [*THEN eq-equiv-class-iff*, *simp*]
declare *conj-cong* [*cong*]

lemmas *eq-intrelD* = *eq-equiv-class* [*OF - equiv-intrel*]

lemma *int-of-type* [*simp*,*TC*]: $#*m* : *int*
by (*simp add*: *int-def quotient-def int-of-def*, *auto*)

lemma *int-of-eq* [*iff*]: ($# *m* = $# *n*) <−> *natify*(*m*)=*natify*(*n*)
by (*simp add*: *int-of-def*)

lemma *int-of-inject*: [| $#*m* = $#*n*; *m*∈*nat*; *n*∈*nat* |] ==> *m*=*n*
by (*drule int-of-eq* [*THEN iffD1*], *auto*)

lemma *intify-in-int* [*iff*,*TC*]: *intify*(*x*) : *int*
by (*simp add*: *intify-def*)

lemma *intify-ident* [*simp*]: *n* : *int* ==> *intify*(*n*) = *n*
by (*simp add*: *intify-def*)

30.2 Collapsing rules: to remove *intify* from arithmetic expressions

lemma *intify-idem* [*simp*]: *intify*(*intify*(*x*)) = *intify*(*x*)
by *simp*

lemma *int-of-natify* [*simp*]: $# (*natify*(*m*)) = $# *m*
by (*simp add*: *int-of-def*)

lemma *zminus-intify* [*simp*]: $− (*intify*(*m*)) = $− *m*
by (*simp add*: *zminus-def*)

lemma *zadd-intify1* [*simp*]: *intify*(*x*) $+ *y* = *x* $+ *y*
by (*simp add*: *zadd-def*)

lemma *zadd-intify2* [*simp*]: *x* $+ *intify*(*y*) = *x* $+ *y*
by (*simp add*: *zadd-def*)

lemma *zdiff-intify1* [*simp*]:*intify*(*x*) $− *y* = *x* $− *y*
by (*simp add*: *zdiff-def*)

lemma *zdiff-intify2* [*simp*]:*x* $− *intify*(*y*) = *x* $− *y*
by (*simp add*: *zdiff-def*)

lemma *zmult-intify1* [*simp*]:*intify*(*x*) $∗ *y* = *x* $∗ *y*
by (*simp add*: *zmult-def*)

lemma *zmult-intify2* [*simp*]:*x* $∗ *intify*(*y*) = *x* $∗ *y*

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

lemma *zless-intify1* [*simp*]:*intify*(*x*) $< *y* <−> *x* $< *y*
by (*simp add*: *zless-def*)

lemma *zless-intify2* [*simp*]:*x* $< *intify*(*y*) <−> *x* $< *y*
by (*simp add*: *zless-def*)

lemma *zle-intify1* [*simp*]:*intify*(*x*) $<= *y* <−> *x* $<= *y*
by (*simp add*: *zle-def*)

lemma *zle-intify2* [*simp*]:*x* $<= *intify*(*y*) <−> *x* $<= *y*
by (*simp add*: *zle-def*)

30.3 *zminus*: unary negation on *int*

lemma *zminus-congruent*: (%<*x*,*y*>. *intrel*‘‘{<*y*,*x*>}) *respects intrel*
by (*auto simp add*: *congruent-def add-ac*)

lemma *raw-zminus-type*: *z* : *int* ==> *raw-zminus*(*z*) : *int*
apply (*simp add*: *int-def raw-zminus-def*)
apply (*typecheck add*: *UN-equiv-class-type* [*OF equiv-intrel zminus-congruent*])
done

lemma *zminus-type* [*TC*,*iff*]: $−*z* : *int*
by (*simp add*: *zminus-def raw-zminus-type*)

lemma *raw-zminus-inject*:
  [| *raw-zminus*(*z*) = *raw-zminus*(*w*); *z*: *int*; *w*: *int* |] ==> *z*=*w*
apply (*simp add*: *int-def raw-zminus-def*)
apply (*erule UN-equiv-class-inject* [*OF equiv-intrel zminus-congruent*], *safe*)
apply (*auto dest*: *eq-intrelD simp add*: *add-ac*)
done

lemma *zminus-inject-intify* [*dest*!]: $−*z* = $−*w* ==> *intify*(*z*) = *intify*(*w*)
apply (*simp add*: *zminus-def*)
apply (*blast dest*!: *raw-zminus-inject*)
done

lemma *zminus-inject*: [| $−*z* = $−*w*; *z*: *int*; *w*: *int* |] ==> *z*=*w*
by *auto*

lemma *raw-zminus*:
  [| *x*∈*nat*; *y*∈*nat* |] ==> *raw-zminus*(*intrel*‘‘{<*x*,*y*>}) = *intrel* ‘‘ {<*y*,*x*>}
apply (*simp add*: *raw-zminus-def UN-equiv-class* [*OF equiv-intrel zminus-congruent*])
done

lemma zminus:
  [| x∈nat; y∈nat |]
  ==> $− (intrel``{<x,y>}) = intrel `` {<y,x>}
by (simp add: zminus-def raw-zminus image-intrel-int)

lemma raw-zminus-zminus: z : int ==> raw-zminus (raw-zminus(z)) = z
by (auto simp add: int-def raw-zminus)

lemma zminus-zminus-intify [simp]: $− ($− z) = intify(z)
by (simp add: zminus-def raw-zminus-type raw-zminus-zminus)

lemma zminus-int0 [simp]: $− ($#0) = $#0
by (simp add: int-of-def zminus)

lemma zminus-zminus: z : int ==> $− ($− z) = z
by simp

30.4 *znegative*: the test for negative integers

lemma znegative: [| x∈nat; y∈nat |] ==> znegative(intrel``{<x,y>}) <−> x<y
apply (cases x<y)
apply (auto simp add: znegative-def not-lt-iff-le)
apply (subgoal-tac y #+ x2 < x #+ y2, force)
apply (rule add-le-lt-mono, auto)
done

lemma not-znegative-int-of [iff]: ~ znegative($# n)
by (simp add: znegative int-of-def)

lemma znegative-zminus-int-of [simp]: znegative($− $# succ(n))
by (simp add: znegative int-of-def zminus natify-succ)

lemma not-znegative-imp-zero: ~ znegative($− $# n) ==> natify(n)=0
by (simp add: znegative int-of-def zminus Ord-0-lt-iff [THEN iff-sym])

30.5 *nat-of*: Coercion of an Integer to a Natural Number

lemma nat-of-intify [simp]: nat-of(intify(z)) = nat-of(z)
by (simp add: nat-of-def)

lemma nat-of-congruent: (λx. (λ⟨x,y⟩. x #− y)(x)) respects intrel
by (auto simp add: congruent-def split add: nat-diff-split)

lemma raw-nat-of:
  [| x∈nat; y∈nat |] ==> raw-nat-of(intrel``{<x,y>}) = x#−y
by (simp add: raw-nat-of-def UN-equiv-class [OF equiv-intrel nat-of-congruent])

lemma raw-nat-of-int-of: raw-nat-of($# n) = natify(n)
by (simp add: int-of-def raw-nat-of)

lemma *nat-of-int-of* [*simp*]: *nat-of*(\$# *n*) = *natify*(*n*)
by (*simp add*: *raw-nat-of-int-of nat-of-def*)

lemma *raw-nat-of-type*: *raw-nat-of*(*z*) ∈ *nat*
by (*simp add*: *raw-nat-of-def*)

lemma *nat-of-type* [*iff*,*TC*]: *nat-of*(*z*) ∈ *nat*
by (*simp add*: *nat-of-def raw-nat-of-type*)

30.6 zmagnitude: magnitide of an integer, as a natural number

lemma *zmagnitude-int-of* [*simp*]: *zmagnitude*(\$# *n*) = *natify*(*n*)
by (*auto simp add*: *zmagnitude-def int-of-eq*)

lemma *natify-int-of-eq*: *natify*(*x*)=*n* ==> \$#*x* = \$# *n*
apply (*drule sym*)
apply (*simp* (*no-asm-simp*) *add*: *int-of-eq*)
done

lemma *zmagnitude-zminus-int-of* [*simp*]: *zmagnitude*(\$− \$# *n*) = *natify*(*n*)
apply (*simp add*: *zmagnitude-def*)
apply (*rule the-equality*)
apply (*auto dest*!: *not-znegative-imp-zero natify-int-of-eq*
iff del: *int-of-eq*, *auto*)
done

lemma *zmagnitude-type* [*iff*,*TC*]: *zmagnitude*(*z*)∈*nat*
apply (*simp add*: *zmagnitude-def*)
apply (*rule theI2*, *auto*)
done

lemma *not-zneg-int-of*:
[| *z*: *int*; ~ *znegative*(*z*) |] ==> ∃ *n*∈*nat*. *z* = \$# *n*
apply (*auto simp add*: *int-def znegative int-of-def not-lt-iff-le*)
apply (*rename-tac x y*)
apply (*rule-tac x*=*x*#−*y* **in** *bexI*)
apply (*auto simp add*: *add-diff-inverse2*)
done

lemma *not-zneg-mag* [*simp*]:
[| *z*: *int*; ~ *znegative*(*z*) |] ==> \$# (*zmagnitude*(*z*)) = *z*
by (*drule not-zneg-int-of*, *auto*)

lemma *zneg-int-of*:
[| *znegative*(*z*); *z*: *int* |] ==> ∃ *n*∈*nat*. *z* = \$− (\$# *succ*(*n*))
by (*auto simp add*: *int-def znegative zminus int-of-def dest*!: *less-imp-succ-add*)

lemma *zneg-mag* [*simp*]:
  [| *znegative*(*z*); *z*: *int* |] ==> $# (*zmagnitude*(*z*)) = $− *z*
by (*drule zneg-int-of* , *auto*)

lemma *int-cases*: *z* : *int* ==> ∃ *n*∈*nat*. *z* = $# *n* | *z* = $− ($# *succ*(*n*))
apply (*case-tac znegative* (*z*))
prefer *2* **apply** (*blast dest*: *not-zneg-mag sym*)
apply (*blast dest*: *zneg-int-of*)
done

lemma *not-zneg-raw-nat-of* :
  [| ~ *znegative*(*z*); *z*: *int* |] ==> $# (*raw-nat-of* (*z*)) = *z*
apply (*drule not-zneg-int-of*)
apply (*auto simp add*: *raw-nat-of-type raw-nat-of-int-of*)
done

lemma *not-zneg-nat-of-intify*:
  ~ *znegative*(*intify*(*z*)) ==> $# (*nat-of* (*z*)) = *intify*(*z*)
by (*simp* (*no-asm-simp*) *add*: *nat-of-def not-zneg-raw-nat-of*)

lemma *not-zneg-nat-of* : [| ~ *znegative*(*z*); *z*: *int* |] ==> $# (*nat-of* (*z*)) = *z*
apply (*simp* (*no-asm-simp*) *add*: *not-zneg-nat-of-intify*)
done

lemma *zneg-nat-of* [*simp*]: *znegative*(*intify*(*z*)) ==> *nat-of* (*z*) = *0*
apply (*subgoal-tac intify*(*z*) ∈ *int*)
apply (*simp add*: *int-def*)
apply (*auto simp add*: *znegative nat-of-def raw-nat-of*
    split add: *nat-diff-split*)
done

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
apply (*simp add*: *congruent2-def*)

apply *safe*
apply (*simp* (*no-asm-simp*) *add*: *add-assoc Let-def*)

apply (*rule-tac m1* = *x1a* **in** *add-left-commute* [*THEN ssubst*])
apply (*rule-tac m1* = *x2a* **in** *add-left-commute* [*THEN ssubst*])
apply (*simp* (*no-asm-simp*) *add*: *add-assoc* [*symmetric*])
done

lemma *raw-zadd-type*: [| *z*: *int*; *w*: *int* |] ==> *raw-zadd*(*z*,*w*) : *int*

```
apply (simp add: int-def raw-zadd-def)
apply (rule UN-equiv-class-type2 [OF equiv-intrel zadd-congruent2], assumption+)
apply (simp add: Let-def)
done

lemma zadd-type [iff,TC]: z $+ w : int
by (simp add: zadd-def raw-zadd-type)

lemma raw-zadd:
  [| x1∈nat; y1∈nat;  x2∈nat; y2∈nat |]
   ==> raw-zadd (intrel``{<x1,y1>}, intrel``{<x2,y2>}) =
       intrel `` {<x1#+x2, y1#+y2>}
apply (simp add: raw-zadd-def
            UN-equiv-class2 [OF equiv-intrel equiv-intrel zadd-congruent2])
apply (simp add: Let-def)
done

lemma zadd:
  [| x1∈nat; y1∈nat;  x2∈nat; y2∈nat |]
   ==> (intrel``{<x1,y1>}) $+ (intrel``{<x2,y2>}) =
       intrel `` {<x1#+x2, y1#+y2>}
by (simp add: zadd-def raw-zadd image-intrel-int)

lemma raw-zadd-int0: z : int ==> raw-zadd ($#0,z) = z
by (auto simp add: int-def int-of-def raw-zadd)

lemma zadd-int0-intify [simp]: $#0 $+ z = intify(z)
by (simp add: zadd-def raw-zadd-int0)

lemma zadd-int0: z: int ==> $#0 $+ z = z
by simp

lemma raw-zminus-zadd-distrib:
     [| z: int;  w: int |] ==> $- raw-zadd(z,w) = raw-zadd($- z, $- w)
by (auto simp add: zminus raw-zadd int-def)

lemma zminus-zadd-distrib [simp]: $- (z $+ w) = $- z $+ $- w
by (simp add: zadd-def raw-zminus-zadd-distrib)

lemma raw-zadd-commute:
     [| z: int;  w: int |] ==> raw-zadd(z,w) = raw-zadd(w,z)
by (auto simp add: raw-zadd add-ac int-def)

lemma zadd-commute: z $+ w = w $+ z
by (simp add: zadd-def raw-zadd-commute)

lemma raw-zadd-assoc:
    [| z1: int;  z2: int;  z3: int |]
     ==> raw-zadd (raw-zadd(z1,z2),z3) = raw-zadd(z1,raw-zadd(z2,z3))
```

by (*auto simp add*: *int-def raw-zadd add-assoc*)

lemma *zadd-assoc*: (*z1* $+ *z2*) $+ *z3* = *z1* $+ (*z2* $+ *z3*)
by (*simp add*: *zadd-def raw-zadd-type raw-zadd-assoc*)

lemma *zadd-left-commute*: *z1*$+(*z2*$+*z3*) = *z2*$+(*z1*$+*z3*)
apply (*simp add*: *zadd-assoc* [*symmetric*])
apply (*simp add*: *zadd-commute*)
done

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

lemma *int-of-add*: $# (*m* #+ *n*) = ($#*m*) $+ ($#*n*)
by (*simp add*: *int-of-def zadd*)

lemma *int-succ-int-1*: $# *succ*(*m*) = $# *1* $+ ($# *m*)
by (*simp add*: *int-of-add* [*symmetric*] *natify-succ*)

lemma *int-of-diff*:
  [| *m*∈*nat*; *n le m* |] ==> $# (*m* #− *n*) = ($#*m*) $− ($#*n*)
apply (*simp add*: *int-of-def zdiff-def*)
apply (*frule lt-nat-in-nat*)
apply (*simp-all add*: *zadd zminus add-diff-inverse2*)
done

lemma *raw-zadd-zminus-inverse*: *z* : *int* ==> *raw-zadd* (*z*, $− *z*) = $#*0*
by (*auto simp add*: *int-def int-of-def zminus raw-zadd add-commute*)

lemma *zadd-zminus-inverse* [*simp*]: *z* $+ ($− *z*) = $#*0*
apply (*simp add*: *zadd-def*)
apply (*subst zminus-intify* [*symmetric*])
apply (*rule intify-in-int* [*THEN raw-zadd-zminus-inverse*])
done

lemma *zadd-zminus-inverse2* [*simp*]: ($− *z*) $+ *z* = $#*0*
by (*simp add*: *zadd-commute zadd-zminus-inverse*)

lemma *zadd-int0-right-intify* [*simp*]: *z* $+ $#*0* = *intify*(*z*)
by (*rule trans* [*OF zadd-commute zadd-int0-intify*])

lemma *zadd-int0-right*: *z*:*int* ==> *z* $+ $#*0* = *z*
by *simp*

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
apply (*rule equiv-intrel* [*THEN congruent2-commuteI*], *auto*)

apply (*rename-tac x y*)
apply (*frule-tac t* = %*u*. *x*#***u* **in** *sym* [*THEN subst-context*])
apply (*drule-tac t* = %*u*. *y*#***u* **in** *subst-context*)
apply (*erule add-left-cancel*)+
apply (*simp-all add*: *add-mult-distrib-left*)
done

lemma *raw-zmult-type*: [| *z*: *int*; *w*: *int* |] ==> *raw-zmult*(*z*,*w*) : *int*
apply (*simp add*: *int-def raw-zmult-def*)
apply (*rule UN-equiv-class-type2* [*OF equiv-intrel zmult-congruent2*], *assumption*+)
apply (*simp add*: *Let-def*)
done

lemma *zmult-type* [*iff*,*TC*]: *z* $* *w* : *int*
by (*simp add*: *zmult-def raw-zmult-type*)

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*>}
by (*simp add*: *raw-zmult-def*
UN-equiv-class2 [*OF equiv-intrel equiv-intrel zmult-congruent2*])

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*>}
by (*simp add*: *zmult-def raw-zmult image-intrel-int*)

lemma *raw-zmult-int0*: *z* : *int* ==> *raw-zmult* ($#*0*,*z*) = $#*0*
by (*auto simp add*: *int-def int-of-def raw-zmult*)

lemma *zmult-int0* [*simp*]: $#*0* $* *z* = $#*0*
by (*simp add*: *zmult-def raw-zmult-int0*)

lemma *raw-zmult-int1*: *z* : *int* ==> *raw-zmult* ($#*1*,*z*) = *z*
by (*auto simp add*: *int-def int-of-def raw-zmult*)

lemma *zmult-int1-intify* [*simp*]: $#*1* $* *z* = *intify*(*z*)
by (*simp add*: *zmult-def raw-zmult-int1*)

lemma *zmult-int1*: *z* : *int* ==> $#*1* $* *z* = *z*
by *simp*

lemma *raw-zmult-commute*:
[| *z*: *int*; *w*: *int* |] ==> *raw-zmult*(*z*,*w*) = *raw-zmult*(*w*,*z*)
by (*auto simp add*: *int-def raw-zmult add-ac mult-ac*)

lemma *zmult-commute*: *z* $* *w* = *w* $* *z*
by (*simp add*: *zmult-def raw-zmult-commute*)

lemma *raw-zmult-zminus*:
[| *z*: *int*; *w*: *int* |] ==> *raw-zmult*($− *z*, *w*) = $− *raw-zmult*(*z*, *w*)
by (*auto simp add*: *int-def zminus raw-zmult add-ac*)

lemma *zmult-zminus* [*simp*]: ($− *z*) $* *w* = $− (*z* $* *w*)
apply (*simp add*: *zmult-def raw-zmult-zminus*)
apply (*subst zminus-intify* [*symmetric*], *rule raw-zmult-zminus*, *auto*)
done

lemma *zmult-zminus-right* [*simp*]: *w* $* ($− *z*) = $− (*w* $* *z*)
by (*simp add*: *zmult-commute* [*of w*])

lemma *raw-zmult-assoc*:
[| *z1*: *int*; *z2*: *int*; *z3*: *int* |]
==> *raw-zmult* (*raw-zmult*(*z1*,*z2*),*z3*) = *raw-zmult*(*z1*,*raw-zmult*(*z2*,*z3*))
by (*auto simp add*: *int-def raw-zmult add-mult-distrib-left add-ac mult-ac*)

lemma *zmult-assoc*: (*z1* $* *z2*) $* *z3* = *z1* $* (*z2* $* *z3*)
by (*simp add*: *zmult-def raw-zmult-type raw-zmult-assoc*)

lemma *zmult-left-commute*: *z1*$*(*z2*$**z3*) = *z2*$*(*z1*$**z3*)
apply (*simp add*: *zmult-assoc* [*symmetric*])
apply (*simp add*: *zmult-commute*)
done

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

lemma *raw-zadd-zmult-distrib*:
[| *z1*: *int*; *z2*: *int*; *w*: *int* |]
==> *raw-zmult*(*raw-zadd*(*z1*,*z2*), *w*) =
raw-zadd (*raw-zmult*(*z1*,*w*), *raw-zmult*(*z2*,*w*))
by (*auto simp add*: *int-def raw-zadd raw-zmult add-mult-distrib-left add-ac mult-ac*)

lemma *zadd-zmult-distrib*: (*z1* $+ *z2*) $* *w* = (*z1* $* *w*) $+ (*z2* $* *w*)
by (*simp add*: *zmult-def zadd-def raw-zadd-type raw-zmult-type*
raw-zadd-zmult-distrib)

lemma *zadd-zmult-distrib2*: *w* $* (*z1* $+ *z2*) = (*w* $* *z1*) $+ (*w* $* *z2*)
by (*simp add*: *zmult-commute* [*of w*] *zadd-zmult-distrib*)

```
lemmas int-typechecks =
  int-of-type zminus-type zmagnitude-type zadd-type zmult-type
```

```
lemma zdiff-type [iff,TC]: z $- w : int
by (simp add: zdiff-def)

lemma zminus-zdiff-eq [simp]: $- (z $- y) = y $- z
by (simp add: zdiff-def zadd-commute)

lemma zdiff-zmult-distrib: (z1 $- z2) $* w = (z1 $* w) $- (z2 $* w)
apply (simp add: zdiff-def)
apply (subst zadd-zmult-distrib)
apply (simp add: zmult-zminus)
done

lemma zdiff-zmult-distrib2: w $* (z1 $- z2) = (w $* z1) $- (w $* z2)
by (simp add: zmult-commute [of w] zdiff-zmult-distrib)

lemma zadd-zdiff-eq: x $+ (y $- z) = (x $+ y) $- z
by (simp add: zdiff-def zadd-ac)

lemma zdiff-zadd-eq: (x $- y) $+ z = (x $+ z) $- y
by (simp add: zdiff-def zadd-ac)
```

30.9 The "Less Than" Relation

```
lemma zless-linear-lemma:
    [| z: int; w: int |] ==> z$<w | z=w | w$<z
apply (simp add: int-def zless-def znegative-def zdiff-def, auto)
apply (simp add: zadd zminus image-iff Bex-def)
apply (rule-tac i = xb#+ya and j = xc #+ y in Ord-linear-lt)
apply (force dest!: spec simp add: add-ac)+
done

lemma zless-linear: z$<w | intify(z)=intify(w) | w$<z
apply (cut-tac z = intify (z) and w = intify (w) in zless-linear-lemma)
apply auto
done

lemma zless-not-refl [iff]: ~ (z$<z)
by (auto simp add: zless-def znegative-def int-of-def zdiff-def)

lemma neq-iff-zless: [| x: int; y: int |] ==> (x ~= y) <-> (x $< y | y $< x)
by (cut-tac z = x and w = y in zless-linear, auto)
```

```
lemma zless-imp-intify-neq: w $< z ==> intify(w) ~= intify(z)
apply auto
apply (subgoal-tac ~ (intify (w) $< intify (z)))
apply (erule-tac [2] ssubst)
apply (simp (no-asm-use))
apply auto
done


lemma zless-imp-succ-zadd-lemma:
    [| w $< z; w: int; z: int |] ==> (∃ n∈nat. z = w $+ $#(succ(n)))
apply (simp add: zless-def znegative-def zdiff-def int-def)
apply (auto dest!: less-imp-succ-add simp add: zadd zminus int-of-def)
apply (rule-tac x = k in bexI)
apply (erule add-left-cancel, auto)
done

lemma zless-imp-succ-zadd:
     w $< z ==> (∃ n∈nat. w $+ $#(succ(n)) = intify(z))
apply (subgoal-tac intify (w) $< intify (z) )
apply (drule-tac w = intify (w)  in zless-imp-succ-zadd-lemma)
apply auto
done

lemma zless-succ-zadd-lemma:
    w : int ==> w $< w $+ $# succ(n)
apply (simp add: zless-def znegative-def zdiff-def int-def)
apply (auto simp add: zadd zminus int-of-def image-iff)
apply (rule-tac x = 0 in exI, auto)
done

lemma zless-succ-zadd: w $< w $+ $# succ(n)
by (cut-tac intify-in-int [THEN zless-succ-zadd-lemma], auto)

lemma zless-iff-succ-zadd:
     w $< z <-> (∃ n∈nat. w $+ $#(succ(n)) = intify(z))
apply (rule iffI)
apply (erule zless-imp-succ-zadd, auto)
apply (rename-tac n)
apply (cut-tac w = w and n = n in zless-succ-zadd, auto)
done

lemma zless-int-of [simp]: [| m∈nat; n∈nat |] ==> ($#m $< $#n) <-> (m<n)
apply (simp add: less-iff-succ-add zless-iff-succ-zadd int-of-add [symmetric])
apply (blast intro: sym)
done

lemma zless-trans-lemma:
    [| x $< y; y $< z; x: int; y : int; z: int |] ==> x $< z
```

apply (*simp add*: *zless-def znegative-def zdiff-def int-def*)
apply (*auto simp add*: *zadd zminus image-iff*)
apply (*rename-tac x1 x2 y1 y2*)
apply (*rule-tac x = x1*#+*x2* **in** *exI*)
apply (*rule-tac x = y1*#+*y2* **in** *exI*)
apply (*auto simp add*: *add-lt-mono*)
apply (*rule sym*)
apply (*erule add-left-cancel*)+
apply *auto*
done

lemma *zless-trans*: [| *x* $< *y*; *y* $< *z* |] ==> *x* $< *z*
apply (*subgoal-tac intify* (*x*) $< *intify* (*z*))
apply (*rule-tac* [*2*] *y* = *intify* (*y*) **in** *zless-trans-lemma*)
apply *auto*
done

lemma *zless-not-sym*: *z* $< *w* ==> ~ (*w* $< *z*)
by (*blast dest*: *zless-trans*)

lemmas *zless-asym* = *zless-not-sym* [*THEN swap*, *standard*]

lemma *zless-imp-zle*: *z* $< *w* ==> *z* $<= *w*
by (*simp add*: *zle-def*)

lemma *zle-linear*: *z* $<= *w* | *w* $<= *z*
apply (*simp add*: *zle-def*)
apply (*cut-tac zless-linear*, *blast*)
done

30.10 Less Than or Equals

lemma *zle-refl*: *z* $<= *z*
by (*simp add*: *zle-def*)

lemma *zle-eq-refl*: *x*=*y* ==> *x* $<= *y*
by (*simp add*: *zle-refl*)

lemma *zle-anti-sym-intify*: [| *x* $<= *y*; *y* $<= *x* |] ==> *intify*(*x*) = *intify*(*y*)
apply (*simp add*: *zle-def*, *auto*)
apply (*blast dest*: *zless-trans*)
done

lemma *zle-anti-sym*: [| *x* $<= *y*; *y* $<= *x*; *x*: *int*; *y*: *int* |] ==> *x*=*y*
by (*drule zle-anti-sym-intify*, *auto*)

lemma *zle-trans-lemma*:
[| *x*: *int*; *y*: *int*; *z*: *int*; *x* $<= *y*; *y* $<= *z* |] ==> *x* $<= *z*

apply (*simp add*: *zle-def*, *auto*)
apply (*blast intro*: *zless-trans*)
done

lemma *zle-trans*: [| *x* $<= *y*; *y* $<= *z* |] ==> *x* $<= *z*
apply (*subgoal-tac intify* (*x*) $<= *intify* (*z*))
apply (*rule-tac* [*2*] *y* = *intify* (*y*) **in** *zle-trans-lemma*)
apply *auto*
done

lemma *zle-zless-trans*: [| *i* $<= *j*; *j* $< *k* |] ==> *i* $< *k*
apply (*auto simp add*: *zle-def*)
apply (*blast intro*: *zless-trans*)
apply (*simp add*: *zless-def zdiff-def zadd-def*)
done

lemma *zless-zle-trans*: [| *i* $< *j*; *j* $<= *k* |] ==> *i* $< *k*
apply (*auto simp add*: *zle-def*)
apply (*blast intro*: *zless-trans*)
apply (*simp add*: *zless-def zdiff-def zminus-def*)
done

lemma *not-zless-iff-zle*: ~ (*z* $< *w*) <-> (*w* $<= *z*)
apply (*cut-tac z* = *z* **and** *w* = *w* **in** *zless-linear*)
apply (*auto dest*: *zless-trans simp add*: *zle-def*)
apply (*auto dest*!: *zless-imp-intify-neq*)
done

lemma *not-zle-iff-zless*: ~ (*z* $<= *w*) <-> (*w* $< *z*)
by (*simp add*: *not-zless-iff-zle* [*THEN iff-sym*])

30.11 More subtraction laws (for *zcompare-rls*)

lemma *zdiff-zdiff-eq*: (*x* $− *y*) $− *z* = *x* $− (*y* $+ *z*)
by (*simp add*: *zdiff-def zadd-ac*)

lemma *zdiff-zdiff-eq2*: *x* $− (*y* $− *z*) = (*x* $+ *z*) $− *y*
by (*simp add*: *zdiff-def zadd-ac*)

lemma *zdiff-zless-iff*: (*x*$−*y* $< *z*) <-> (*x* $< *z* $+ *y*)
by (*simp add*: *zless-def zdiff-def zadd-ac*)

lemma *zless-zdiff-iff*: (*x* $< *z*$−*y*) <-> (*x* $+ *y* $< *z*)
by (*simp add*: *zless-def zdiff-def zadd-ac*)

lemma *zdiff-eq-iff*: [| *x*: *int*; *z*: *int* |] ==> (*x*$−*y* = *z*) <-> (*x* = *z* $+ *y*)
by (*auto simp add*: *zdiff-def zadd-assoc*)

lemma *eq-zdiff-iff*: [| *x*: *int*; *z*: *int* |] ==> (*x* = *z*$−*y*) <-> (*x* $+ *y* = *z*)

```
by (auto simp add: zdiff-def zadd-assoc)

lemma zdiff-zle-iff-lemma:
    [| x: int; z: int |] ==> (x$−y $<= z) <−> (x $<= z $+ y)
by (auto simp add: zle-def zdiff-eq-iff zdiff-zless-iff)

lemma zdiff-zle-iff: (x$−y $<= z) <−> (x $<= z $+ y)
by (cut-tac zdiff-zle-iff-lemma [OF intify-in-int intify-in-int], simp)

lemma zle-zdiff-iff-lemma:
    [| x: int; z: int |] ==>(x $<= z$−y) <−> (x $+ y $<= z)
apply (auto simp add: zle-def zdiff-eq-iff zless-zdiff-iff)
apply (auto simp add: zdiff-def zadd-assoc)
done

lemma zle-zdiff-iff: (x $<= z$−y) <−> (x $+ y $<= z)
by (cut-tac zle-zdiff-iff-lemma [ OF intify-in-int intify-in-int], simp)
```

This list of rewrites simplifies (in)equalities by bringing subtractions to the top and then moving negative terms to the other side. Use with *zadd-ac*

```
lemmas zcompare-rls =
    zdiff-def [symmetric]
    zadd-zdiff-eq zdiff-zadd-eq zdiff-zdiff-eq zdiff-zdiff-eq2
    zdiff-zless-iff zless-zdiff-iff zdiff-zle-iff zle-zdiff-iff
    zdiff-eq-iff eq-zdiff-iff
```

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)
apply safe
apply (drule-tac t = %x. x $+ ($−z) in subst-context)
apply (simp add: zadd-ac)
done

lemma zadd-left-cancel-intify [simp]:
    (z $+ w' = z $+ w) <−> intify(w') = intify(w)
apply (rule iff-trans)
apply (rule-tac [2] zadd-left-cancel, auto)
done

lemma zadd-right-cancel:
    [| w: int; w': int |] ==> (w' $+ z = w $+ z) <−> (w' = w)
apply safe
apply (drule-tac t = %x. x $+ ($−z) in subst-context)
apply (simp add: zadd-ac)
done
```

lemma *zadd-right-cancel-intify* [*simp*]:
  (*w'* $+ *z* = *w* $+ *z*) <−> *intify*(*w'*) = *intify*(*w*)
apply (*rule iff-trans*)
apply (*rule-tac* [*2*] *zadd-right-cancel*, *auto*)
done

lemma *zadd-right-cancel-zless* [*simp*]: (*w'* $+ *z* $< *w* $+ *z*) <−> (*w'* $< *w*)
by (*simp add*: *zdiff-zless-iff* [*THEN iff-sym*] *zdiff-def zadd-assoc*)

lemma *zadd-left-cancel-zless* [*simp*]: (*z* $+ *w'* $< *z* $+ *w*) <−> (*w'* $< *w*)
by (*simp add*: *zadd-commute* [*of z*] *zadd-right-cancel-zless*)

lemma *zadd-right-cancel-zle* [*simp*]: (*w'* $+ *z* $<= *w* $+ *z*) <−> *w'* $<= *w*
by (*simp add*: *zle-def*)

lemma *zadd-left-cancel-zle* [*simp*]: (*z* $+ *w'* $<= *z* $+ *w*) <−> *w'* $<= *w*
by (*simp add*: *zadd-commute* [*of z*] *zadd-right-cancel-zle*)

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*
by (*erule zadd-zle-mono1* [*THEN zle-trans*], *simp*)

lemma *zadd-zless-mono*: [| *w'* $< *w*; *z'* $<= *z* |] ==> *w'* $+ *z'* $< *w* $+ *z*
by (*erule zadd-zless-mono1* [*THEN zless-zle-trans*], *simp*)

30.13 Comparison laws

lemma *zminus-zless-zminus* [*simp*]: ($− *x* $< $− *y*) <−> (*y* $< *x*)
by (*simp add*: *zless-def zdiff-def zadd-ac*)

lemma *zminus-zle-zminus* [*simp*]: ($− *x* $<= $− *y*) <−> (*y* $<= *x*)
by (*simp add*: *not-zless-iff-zle* [*THEN iff-sym*])

30.13.1 More inequality lemmas

lemma *equation-zminus*: [| *x*: *int*; *y*: *int* |] ==> (*x* = $− *y*) <−> (*y* = $− *x*)
by *auto*

lemma *zminus-equation*: [| *x*: *int*; *y*: *int* |] ==> ($− *x* = *y*) <−> ($− *y* = *x*)
by *auto*

lemma *equation-zminus-intify*: (*intify*(*x*) = $− *y*) <−> (*intify*(*y*) = $− *x*)
apply (*cut-tac x* = *intify* (*x*) **and** *y* = *intify* (*y*) **in** *equation-zminus*)
apply *auto*
done

lemma *zminus-equation-intify*: ($− *x* = *intify*(*y*)) <−> ($− *y* = *intify*(*x*))
apply (*cut-tac x* = *intify* (*x*) **and** *y* = *intify* (*y*) **in** *zminus-equation*)
apply *auto*
done

30.13.2 The next several equations are permutative: watch out!

lemma *zless-zminus*: (*x* $< $− *y*) <−> (*y* $< $− *x*)
by (*simp add*: *zless-def zdiff-def zadd-ac*)

lemma *zminus-zless*: ($− *x* $< *y*) <−> ($− *y* $< *x*)
by (*simp add*: *zless-def zdiff-def zadd-ac*)

lemma *zle-zminus*: (*x* $<= $− *y*) <−> (*y* $<= $− *x*)
by (*simp add*: *not-zless-iff-zle* [*THEN iff-sym*] *zminus-zless*)

lemma *zminus-zle*: ($− *x* $<= *y*) <−> ($− *y* $<= *x*)
by (*simp add*: *not-zless-iff-zle* [*THEN iff-sym*] *zless-zminus*)

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*)

use *Tools/numeral-syntax.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*))

setup *NumeralSyntax.setup*

declare *bin.intros* [*simp*,*TC*]

lemma *NCons-Pls-0*: *NCons*(*Pls*,*0*) = *Pls*
by *simp*

lemma *NCons-Pls-1*: *NCons*(*Pls*,*1*) = *Pls BIT 1*
by *simp*

lemma *NCons-Min-0*: *NCons*(*Min*,*0*) = *Min BIT 0*
by *simp*

lemma *NCons-Min-1*: *NCons*(*Min*,*1*) = *Min*
by *simp*

lemma *NCons-BIT*: *NCons*(*w BIT x*,*b*) = *w BIT x BIT b*
by (*simp add*: *bin.case-eqns*)

lemmas *NCons-simps* [*simp*] =
NCons-Pls-0 NCons-Pls-1 NCons-Min-0 NCons-Min-1 NCons-BIT

lemma *integ-of-type* [*TC*]: *w*: *bin* ==> *integ-of*(*w*) : *int*
apply (*induct-tac w*)
apply (*simp-all add*: *bool-into-nat*)

done

lemma *NCons-type* [*TC*]: [| *w*: *bin*; *b*: *bool* |] ==> *NCons*(*w*,*b*) : *bin*
by (*induct-tac w*, *auto*)

lemma *bin-succ-type* [*TC*]: *w*: *bin* ==> *bin-succ*(*w*) : *bin*
by (*induct-tac w*, *auto*)

lemma *bin-pred-type* [*TC*]: *w*: *bin* ==> *bin-pred*(*w*) : *bin*
by (*induct-tac w*, *auto*)

lemma *bin-minus-type* [*TC*]: *w*: *bin* ==> *bin-minus*(*w*) : *bin*
by (*induct-tac w*, *auto*)

lemma *bin-add-type* [*rule-format*,*TC*]:
  v: *bin* ==> *ALL w*: *bin*. *bin-add*(*v*,*w*) : *bin*
apply (*unfold bin-add-def*)
apply (*induct-tac v*)
apply (*rule-tac* [*3*] *ballI*)
apply (*rename-tac* [*3*] *w'*)
apply (*induct-tac* [*3*] *w'*)
apply (*simp-all add*: *NCons-type*)
done

lemma *bin-mult-type* [*TC*]: [| *v*: *bin*; *w*: *bin* |] ==> *bin-mult*(*v*,*w*) : *bin*
by (*induct-tac v*, *auto*)

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*)
apply (*erule bin.cases*)
apply (*auto elim*!: *boolE*)
done

lemma *integ-of-succ* [*simp*]:
  w: *bin* ==> *integ-of*(*bin-succ*(*w*)) = $#*1* $+ *integ-of*(*w*)
apply (*erule bin.induct*)
apply (*auto simp add*: *zadd-ac elim*!: *boolE*)
done

lemma *integ-of-pred* [*simp*]:
  w: *bin* ==> *integ-of*(*bin-pred*(*w*)) = $− ($#*1*) $+ *integ-of*(*w*)
apply (*erule bin.induct*)
apply (*auto simp add*: *zadd-ac elim*!: *boolE*)
done

31.0.4 *bin-minus*: Unary Negation of Binary Integers

lemma *integ-of-minus*: *w*: *bin* ==> *integ-of*(*bin-minus*(*w*)) = $− *integ-of*(*w*)
apply (*erule bin.induct*)
apply (*auto simp add*: *zadd-ac zminus-zadd-distrib elim*!: *boolE*)
done

31.0.5 *bin-add*: Binary Addition

lemma *bin-add-Pls* [*simp*]: *w*: *bin* ==> *bin-add*(*Pls*,*w*) = *w*
by (*unfold bin-add-def*, *simp*)

lemma *bin-add-Pls-right*: *w*: *bin* ==> *bin-add*(*w*,*Pls*) = *w*
apply (*unfold bin-add-def*)
apply (*erule bin.induct*, *auto*)
done

lemma *bin-add-Min* [*simp*]: *w*: *bin* ==> *bin-add*(*Min*,*w*) = *bin-pred*(*w*)
by (*unfold bin-add-def*, *simp*)

lemma *bin-add-Min-right*: *w*: *bin* ==> *bin-add*(*w*,*Min*) = *bin-pred*(*w*)
apply (*unfold bin-add-def*)
apply (*erule bin.induct*, *auto*)
done

lemma *bin-add-BIT-Pls* [*simp*]: *bin-add*(*v BIT x*,*Pls*) = *v BIT x*
by (*unfold bin-add-def*, *simp*)

lemma *bin-add-BIT-Min* [*simp*]: *bin-add*(*v BIT x*,*Min*) = *bin-pred*(*v BIT x*)
by (*unfold bin-add-def*, *simp*)

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*)
by (*unfold bin-add-def*, *simp*)

lemma *integ-of-add* [*rule-format*]:
  v: *bin* ==>
    ALL w: *bin*. *integ-of*(*bin-add*(*v*,*w*)) = *integ-of*(*v*) $+ *integ-of*(*w*)
apply (*erule bin.induct*, *simp*, *simp*)
apply (*rule ballI*)
apply (*induct-tac wa*)
apply (*auto simp add*: *zadd-ac elim*!: *boolE*)
done

lemma *diff-integ-of-eq*:
  [| *v*: *bin*; *w*: *bin* |]
  ==> *integ-of*(*v*) $− *integ-of*(*w*) = *integ-of*(*bin-add* (*v*, *bin-minus*(*w*)))

apply (*unfold zdiff-def*)
apply (*simp add*: *integ-of-add integ-of-minus*)
done

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*)
apply (*induct-tac v*, *simp*)
apply (*simp add*: *integ-of-minus*)
apply (*auto simp add*: *zadd-ac integ-of-add zadd-zmult-distrib elim*!: *boolE*)
done

31.1 Computations

lemma *bin-succ-1*: *bin-succ*(*w BIT 1*) = *bin-succ*(*w*) *BIT 0*
by *simp*

lemma *bin-succ-0*: *bin-succ*(*w BIT 0*) = *NCons*(*w*,*1*)
by *simp*

lemma *bin-pred-1*: *bin-pred*(*w BIT 1*) = *NCons*(*w*,*0*)
by *simp*

lemma *bin-pred-0*: *bin-pred*(*w BIT 0*) = *bin-pred*(*w*) *BIT 1*
by *simp*

lemma *bin-minus-1*: *bin-minus*(*w BIT 1*) = *bin-pred*(*NCons*(*bin-minus*(*w*), *0*))
by *simp*

lemma *bin-minus-0*: *bin-minus*(*w BIT 0*) = *bin-minus*(*w*) *BIT 0*
by *simp*

lemma *bin-add-BIT-11*: *w*: *bin* ==> *bin-add*(*v BIT 1*, *w BIT 1*) =
    NCons(*bin-add*(*v*, *bin-succ*(*w*)), *0*)
by *simp*

lemma *bin-add-BIT-10*: *w*: *bin* ==> *bin-add*(*v BIT 1*, *w BIT 0*) =
    NCons(*bin-add*(*v*,*w*), *1*)
by *simp*

lemma *bin-add-BIT-0*: [| *w*: *bin*; *y*: *bool* |]
  ==> *bin-add*(*v BIT 0*, *w BIT y*) = *NCons*(*bin-add*(*v*,*w*), *y*)
by *simp*

lemma *bin-mult-1*: *bin-mult*(*v BIT 1*, *w*) = *bin-add*(*NCons*(*bin-mult*(*v*,*w*),*0*), *w*)
by *simp*

lemma *bin-mult-0*: *bin-mult*(*v BIT 0*, *w*) = *NCons*(*bin-mult*(*v*,*w*),*0*)
by *simp*

lemma *int-of-0*: $#*0* = #*0*
by *simp*

lemma *int-of-succ*: $# *succ*(*n*) = #*1* $+ $#*n*
by (*simp add*: *int-of-add* [*symmetric*] *natify-succ*)

lemma *zminus-0* [*simp*]: $− #*0* = #*0*
by *simp*

lemma *zadd-0-intify* [*simp*]: #*0* $+ *z* = *intify*(*z*)
by *simp*

lemma *zadd-0-right-intify* [*simp*]: *z* $+ #*0* = *intify*(*z*)
by *simp*

lemma *zmult-1-intify* [*simp*]: #*1* $∗ *z* = *intify*(*z*)
by *simp*

lemma *zmult-1-right-intify* [*simp*]: *z* $∗ #*1* = *intify*(*z*)
by (*subst zmult-commute*, *simp*)

lemma *zmult-0* [*simp*]: #*0* $∗ *z* = #*0*
by *simp*

lemma *zmult-0-right* [*simp*]: *z* $∗ #*0* = #*0*
by (*subst zmult-commute*, *simp*)

lemma *zmult-minus1* [*simp*]: #−*1* $∗ *z* = $−*z*
by (*simp add*: *zcompare-rls*)

lemma *zmult-minus1-right* [*simp*]: *z* $∗ #−*1* = $−*z*
apply (*subst zmult-commute*)
apply (*rule zmult-minus1*)
done

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))))
apply (unfold iszero-def)
apply (simp add: zcompare-rls integ-of-add integ-of-minus)
done

lemma iszero-integ-of-Pls: iszero (integ-of(Pls))
by (unfold iszero-def, simp)

lemma nonzero-integ-of-Min: ~ iszero (integ-of(Min))
apply (unfold iszero-def)
apply (simp add: zminus-equation)
done

lemma iszero-integ-of-BIT:
[| w: bin; x: bool |]
==> iszero (integ-of (w BIT x)) <-> (x=0 & iszero (integ-of(w)))
apply (unfold iszero-def, simp)
apply (subgoal-tac integ-of (w) : int)
apply typecheck
apply (drule int-cases)
apply (safe elim!: boolE)
apply (simp-all (asm-lr) add: zcompare-rls zminus-zadd-distrib [symmetric]
int-of-add [symmetric])
done

lemma iszero-integ-of-0:
w: bin ==> iszero (integ-of (w BIT 0)) <-> iszero (integ-of(w))
by (simp only: iszero-integ-of-BIT, blast)

lemma iszero-integ-of-1: w: bin ==> ~ iszero (integ-of (w BIT 1))
by (simp only: iszero-integ-of-BIT, blast)

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))))
apply (unfold zless-def zdiff-def)

apply (*simp add*: *integ-of-minus integ-of-add*)
done

lemma *not-neg-integ-of-Pls*: ~ *znegative* (*integ-of*(*Pls*))
by *simp*

lemma *neg-integ-of-Min*: *znegative* (*integ-of*(*Min*))
by *simp*

lemma *neg-integ-of-BIT*:
[| *w*: *bin*; *x*: *bool* |]
==> *znegative* (*integ-of* (*w BIT x*)) <-> *znegative* (*integ-of*(*w*))
apply *simp*
apply (*subgoal-tac integ-of* (*w*) : *int*)
apply *typecheck*
apply (*drule int-cases*)
apply (*auto elim*!: *boolE simp add*: *int-of-add* [*symmetric*] *zcompare-rls*)
apply (*simp-all add*: *zminus-zadd-distrib* [*symmetric*] *zdiff-def*
int-of-add [*symmetric*])
apply (*subgoal-tac* $#*1* $− $# *succ* (*succ* (*n* #+ *n*)) = $− $# *succ* (*n* #+ *n*))
apply (*simp add*: *zdiff-def*)
apply (*simp add*: *equation-zminus int-of-diff* [*symmetric*])
done

lemma *le-integ-of-eq-not-less*:
(*integ-of*(*x*) $<= (*integ-of*(*w*))) <-> ~ (*integ-of*(*w*) $< (*integ-of*(*x*)))
by (*simp add*: *not-zless-iff-zle* [*THEN iff-sym*])

declare *bin-succ-BIT* [*simp del*]
bin-pred-BIT [*simp del*]
bin-minus-BIT [*simp del*]
NCons-Pls [*simp del*]
NCons-Min [*simp del*]
bin-adder-BIT [*simp del*]
bin-mult-BIT [*simp del*]

declare *integ-of-Pls* [*simp del*] *integ-of-Min* [*simp del*] *integ-of-BIT* [*simp del*]

lemmas *bin-arith-extra-simps* =
integ-of-add [*symmetric*]
integ-of-minus [*symmetric*]
integ-of-mult [*symmetric*]
bin-succ-1 bin-succ-0

bin-pred-1 bin-pred-0
bin-minus-1 bin-minus-0
bin-add-Pls-right bin-add-Min-right
bin-add-BIT-0 bin-add-BIT-10 bin-add-BIT-11
diff-integ-of-eq
bin-mult-1 bin-mult-0 NCons-simps

lemmas *bin-arith-simps* =
bin-pred-Pls bin-pred-Min
bin-succ-Pls bin-succ-Min
bin-add-Pls bin-add-Min
bin-minus-Pls bin-minus-Min
bin-mult-Pls bin-mult-Min
bin-arith-extra-simps

lemmas *bin-rel-simps* =
eq-integ-of-eq iszero-integ-of-Pls nonzero-integ-of-Min
iszero-integ-of-0 iszero-integ-of-1
less-integ-of-eq-neg
not-neg-integ-of-Pls neg-integ-of-Min neg-integ-of-BIT
le-integ-of-eq-not-less

declare *bin-arith-simps* [*simp*]
declare *bin-rel-simps* [*simp*]

lemma *add-integ-of-left* [*simp*]:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $+ (*integ-of*(*w*) $+ *z*) = (*integ-of*(*bin-add*(*v*,*w*)) $+ *z*)
by (*simp add*: *zadd-assoc* [*symmetric*])

lemma *mult-integ-of-left* [*simp*]:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $∗ (*integ-of*(*w*) $∗ *z*) = (*integ-of*(*bin-mult*(*v*,*w*)) $∗ *z*)
by (*simp add*: *zmult-assoc* [*symmetric*])

lemma *add-integ-of-diff1* [*simp*]:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $+ (*integ-of*(*w*) $− *c*) = *integ-of*(*bin-add*(*v*,*w*)) $− (*c*)
apply (*unfold zdiff-def*)
apply (*rule add-integ-of-left*, *auto*)
done

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*)
apply (*subst diff-integ-of-eq* [*symmetric*])
apply (*simp-all add*: *zdiff-def zadd-ac*)
done

declare *int-of-0* [*simp*] *int-of-succ* [*simp*]

lemma *zdiff0* [*simp*]: *#0* $− *x* = $−*x*
by (*simp add*: *zdiff-def*)

lemma *zdiff0-right* [*simp*]: *x* $− *#0* = *intify*(*x*)
by (*simp add*: *zdiff-def*)

lemma *zdiff-self* [*simp*]: *x* $− *x* = *#0*
by (*simp add*: *zdiff-def*)

lemma *znegative-iff-zless-0*: *k*: *int* ==> *znegative*(*k*) <−> *k* $< *#0*
by (*simp add*: *zless-def*)

lemma *zero-zless-imp-znegative-zminus*: [|*#0* $< *k*; *k*: *int*|] ==> *znegative*($−*k*)
by (*simp add*: *zless-def*)

lemma *zero-zle-int-of* [*simp*]: *#0* $<= $# *n*
by (*simp add*: *not-zless-iff-zle* [*THEN iff-sym*] *znegative-iff-zless-0* [*THEN iff-sym*])

lemma *nat-of-0* [*simp*]: *nat-of*(*#0*) = *0*
by (*simp only*: *natify-0 int-of-0* [*symmetric*] *nat-of-int-of*)

lemma *nat-le-int0-lemma*: [| *z* $<= $*#0*; *z*: *int* |] ==> *nat-of*(*z*) = *0*
by (*auto simp add*: *znegative-iff-zless-0* [*THEN iff-sym*] *zle-def zneg-nat-of*)

lemma *nat-le-int0*: *z* $<= $*#0* ==> *nat-of*(*z*) = *0*
apply (*subgoal-tac nat-of* (*intify* (*z*)) = *0*)
apply (*rule-tac* [*2*] *nat-le-int0-lemma*, *auto*)
done

lemma *int-of-eq-0-imp-natify-eq-0*: $# *n* = *#0* ==> *natify*(*n*) = *0*
by (*rule not-znegative-imp-zero*, *auto*)

lemma *nat-of-zminus-int-of*: *nat-of*($− $# *n*) = *0*
by (*simp add*: *nat-of-def int-of-def raw-nat-of zminus image-intrel-int*)

lemma *int-of-nat-of*: *#0* $<= *z* ==> $# *nat-of*(*z*) = *intify*(*z*)
apply (*rule not-zneg-nat-of-intify*)

apply (*simp add*: *znegative-iff-zless-0 not-zless-iff-zle*)
done

declare *int-of-nat-of* [*simp*] *nat-of-zminus-int-of* [*simp*]

lemma *int-of-nat-of-if*: $# *nat-of*(*z*) = (*if* *#0* $<= *z then intify*(*z*) *else* *#0*)
by (*simp add*: *int-of-nat-of znegative-iff-zless-0 not-zle-iff-zless*)

lemma *zless-nat-iff-int-zless*: [| *m*: *nat*; *z*: *int* |] ==> (*m* < *nat-of*(*z*)) <-> ($#*m* $< *z*)
apply (*case-tac znegative* (*z*))
apply (*erule-tac* [*2*] *not-zneg-nat-of* [*THEN subst*])
apply (*auto dest*: *zless-trans dest*!: *zero-zle-int-of* [*THEN zle-zless-trans*]
    simp add: *znegative-iff-zless-0*)
done

lemma *zless-nat-conj-lemma*: $#*0* $< *z* ==> (*nat-of*(*w*) < *nat-of*(*z*)) <-> (*w* $< *z*)
apply (*rule iff-trans*)
apply (*rule zless-int-of* [*THEN iff-sym*])
apply (*auto simp add*: *int-of-nat-of-if simp del*: *zless-int-of*)
apply (*auto elim*: *zless-asym simp add*: *not-zle-iff-zless*)
apply (*blast intro*: *zless-zle-trans*)
done

lemma *zless-nat-conj*: (*nat-of*(*w*) < *nat-of*(*z*)) <-> ($#*0* $< *z* & *w* $< *z*)
apply (*case-tac* $#*0* $< *z*)
apply (*auto simp add*: *zless-nat-conj-lemma nat-le-int0 not-zless-iff-zle*)
done

lemma *integ-of-minus-reorient* [*simp*]:
  (*integ-of*(*w*) = $- *x*) <-> ($- *x* = *integ-of*(*w*))
by *auto*

lemma *integ-of-add-reorient* [*simp*]:
  (*integ-of*(*w*) = *x* $+ *y*) <-> (*x* $+ *y* = *integ-of*(*w*))
by *auto*

lemma *integ-of-diff-reorient* [*simp*]:
  (*integ-of*(*w*) = *x* $- *y*) <-> (*x* $- *y* = *integ-of*(*w*))
by *auto*

lemma *integ-of-mult-reorient* [*simp*]:

(*integ-of*(w) = x $* y) <-> (x $* y = *integ-of*(w))
by *auto*

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)
by (*simp add*: *zcompare-rls*)

lemma *eq-iff-zdiff-eq-0*: [| x: *int*; y: *int* |] ==> (x = y) <-> (x$-y = #0)
by (*simp add*: *zcompare-rls*)

lemma *zle-iff-zdiff-zle-0*: (x $<= y) <-> (x$-y $<= #0)
by (*simp add*: *zcompare-rls*)

lemma *left-zadd-zmult-distrib*: i$*u $+ (j$*u $+ k) = (i$+j)$*u $+ k
by (*simp add*: *zadd-zmult-distrib zadd-ac*)

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))
  apply (simp add: zdiff-def zadd-zmult-distrib)
  apply (simp add: zcompare-rls)
  apply (simp add: zadd-ac)
  done

lemma eq-add-iff2: (i$*u $+ m = j$*u $+ n) <−> (intify(m) = (j$−i)$*u $+ n)
  apply (simp add: zdiff-def zadd-zmult-distrib)
  apply (simp add: zcompare-rls)
  apply (simp add: zadd-ac)
  done

lemma less-add-iff1: (i$*u $+ m $< j$*u $+ n) <−> ((i$−j)$*u $+ m $< n)
  apply (simp add: zdiff-def zadd-zmult-distrib zadd-ac rel-iff-rel-0-rls)
  done

lemma less-add-iff2: (i$*u $+ m $< j$*u $+ n) <−> (m $< (j$−i)$*u $+ n)
  apply (simp add: zdiff-def zadd-zmult-distrib zadd-ac rel-iff-rel-0-rls)
  done

lemma le-add-iff1: (i$*u $+ m $<= j$*u $+ n) <−> ((i$−j)$*u $+ m $<= n)
  apply (simp add: zdiff-def zadd-zmult-distrib)
  apply (simp add: zcompare-rls)
  apply (simp add: zadd-ac)
  done

lemma le-add-iff2: (i$*u $+ m $<= j$*u $+ n) <−> (m $<= (j$−i)$*u $+ n)
  apply (simp add: zdiff-def zadd-zmult-distrib)
  apply (simp add: zcompare-rls)
  apply (simp add: zadd-ac)
  done

use int-arith.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
apply (rule-tac y = y in zless-trans)
apply (rule-tac [2] zdiff-zless-iff [THEN iffD1])
apply auto
done

lemma zpos-add-zpos-imp-zpos: [| #0 $<= x;  #0 $<= y |] ==> #0 $<= x $+
y
apply (rule-tac y = y in zle-trans)
apply (rule-tac [2] zdiff-zle-iff [THEN iffD1])
apply auto
done

lemma zneg-add-zneg-imp-zneg: [| x $< #0;  y $< #0 |] ==> x $+ y $< #0
apply (rule-tac y = y in zless-trans)
apply (rule zless-zdiff-iff [THEN iffD1])
apply auto
done


lemma zneg-or-0-add-zneg-or-0-imp-zneg-or-0:
     [| x $<= #0;  y $<= #0 |] ==> x $+ y $<= #0
apply (rule-tac y = y in zle-trans)
apply (rule zle-zdiff-iff [THEN iffD1])
apply auto
done

lemma zero-lt-zmagnitude: [| #0 $< k; k ∈ int |] ==> 0 < zmagnitude(k)
apply (drule zero-zless-imp-znegative-zminus)
```

apply (*drule-tac* [*2*] *zneg-int-of*)
apply (*auto simp add*: *zminus-equation* [*of k*])
apply (*subgoal-tac 0* < *zmagnitude* ($# *succ* (*n*)))
apply *simp*
apply (*simp only*: *zmagnitude-int-of*)
apply *simp*
done

lemma *zless-add-succ-iff*:
  (*w* $< *z* $+ $# *succ*(*m*)) <−> (*w* $< *z* $+ $#*m* | *intify*(*w*) = *z* $+ $#*m*)
apply (*auto simp add*: *zless-iff-succ-zadd zadd-assoc int-of-add* [*symmetric*])
apply (*rule-tac* [*3*] *x* = *0* **in** *bexI*)
apply (*cut-tac m* = *m* **in** *int-succ-int-1*)
apply (*cut-tac m* = *n* **in** *int-succ-int-1*)
apply *simp*
apply (*erule natE*)
apply *auto*
apply (*rule-tac x* = *succ* (*n*) **in** *bexI*)
apply *auto*
done

lemma *zadd-succ-lemma*:
  z ∈ *int* ==> (*w* $+ $# *succ*(*m*) $<= *z*) <−> (*w* $+ $#*m* $< *z*)
apply (*simp only*: *not-zless-iff-zle* [*THEN iff-sym*] *zless-add-succ-iff*)
apply (*auto intro*: *zle-anti-sym elim*: *zless-asym*
    simp add: *zless-imp-zle not-zless-iff-zle*)
done

lemma *zadd-succ-zle-iff*: (*w* $+ $# *succ*(*m*) $<= *z*) <−> (*w* $+ $#*m* $< *z*)
apply (*cut-tac z* = *intify* (*z*) **in** *zadd-succ-lemma*)
apply *auto*
done

lemma *zless-add1-iff-zle*: (*w* $< *z* $+ #*1*) <−> (*w*$<=*z*)
apply (*subgoal-tac* #*1* = $# *1*)
apply (*simp only*: *zless-add-succ-iff zle-def*)
apply *auto*
done

lemma *add1-zle-iff*: (*w* $+ #*1* $<= *z*) <−> (*w* $< *z*)
apply (*subgoal-tac* #*1* = $# *1*)
apply (*simp only*: *zadd-succ-zle-iff*)
apply *auto*
done

lemma *add1-left-zle-iff*: (*#1* $+ *w* $<= *z*) <−> (*w* $< *z*)
apply (*subst zadd-commute*)
apply (*rule add1-zle-iff*)
done

lemma *zmult-mono-lemma*: *k* ∈ *nat* ==> *i* $<= *j* ==> *i* $∗ $#*k* $<= *j* $∗ $#*k*
apply (*induct-tac k*)
prefer *2* **apply** (*subst int-succ-int-1*)
apply (*simp-all* (*no-asm-simp*) *add*: *zadd-zmult-distrib2 zadd-zle-mono*)
done

lemma *zmult-zle-mono1*: [| *i* $<= *j*; *#0* $<= *k* |] ==> *i*$∗*k* $<= *j*$∗*k*
apply (*subgoal-tac i* $∗ *intify* (*k*) $<= *j* $∗ *intify* (*k*))
apply (*simp* (*no-asm-use*))
apply (*rule-tac b* = *intify* (*k*) **in** *not-zneg-mag* [*THEN subst*])
apply (*rule-tac* [*3*] *zmult-mono-lemma*)
apply *auto*
apply (*simp add*: *znegative-iff-zless-0 not-zless-iff-zle* [*THEN iff-sym*])
done

lemma *zmult-zle-mono1-neg*: [| *i* $<= *j*; *k* $<= *#0* |] ==> *j*$∗*k* $<= *i*$∗*k*
apply (*rule zminus-zle-zminus* [*THEN iffD1*])
apply (*simp del*: *zmult-zminus-right*
　　　　add: *zmult-zminus-right* [*symmetric*] *zmult-zle-mono1 zle-zminus*)
done

lemma *zmult-zle-mono2*: [| *i* $<= *j*; *#0* $<= *k* |] ==> *k*$∗*i* $<= *k*$∗*j*
apply (*drule zmult-zle-mono1*)
apply (*simp-all add*: *zmult-commute*)
done

lemma *zmult-zle-mono2-neg*: [| *i* $<= *j*; *k* $<= *#0* |] ==> *k*$∗*j* $<= *k*$∗*i*
apply (*drule zmult-zle-mono1-neg*)
apply (*simp-all add*: *zmult-commute*)
done

lemma *zmult-zle-mono*:
　[| *i* $<= *j*; *k* $<= *l*; *#0* $<= *j*; *#0* $<= *k* |] ==> *i*$∗*k* $<= *j*$∗*l*
apply (*erule zmult-zle-mono1* [*THEN zle-trans*])
apply *assumption*
apply (*erule zmult-zle-mono2*)
apply *assumption*
done

lemma *zmult-zless-mono2-lemma* [*rule-format*]:
  [| *i*$<*j*; *k* ∈ *nat* |] ==> *0*<*k* −−> $#*k* $∗ *i* $< $#*k* $∗ *j*
apply (*induct-tac k*)
 prefer *2*
 apply (*subst int-succ-int-1*)
 apply (*erule natE*)
apply (*simp-all add*: *zadd-zmult-distrib zadd-zless-mono zle-def*)
apply (*frule nat-0-le*)
apply (*subgoal-tac i* $+ (*i* $+ $# *xa* $∗ *i*) $< *j* $+ (*j* $+ $# *xa* $∗ *j*))
apply (*simp* (*no-asm-use*))
apply (*rule zadd-zless-mono*)
apply (*simp-all* (*no-asm-simp*) *add*: *zle-def*)
done

lemma *zmult-zless-mono2*: [| *i*$<*j*; #*0* $< *k* |] ==> *k*$∗*i* $< *k*$∗*j*
apply (*subgoal-tac intify* (*k*) $∗ *i* $< *intify* (*k*) $∗ *j*)
apply (*simp* (*no-asm-use*))
apply (*rule-tac b* = *intify* (*k*) **in** *not-zneg-mag* [*THEN subst*])
apply (*rule-tac* [*3*] *zmult-zless-mono2-lemma*)
apply *auto*
apply (*simp add*: *znegative-iff-zless-0*)
apply (*drule zless-trans*, *assumption*)
apply (*auto simp add*: *zero-lt-zmagnitude*)
done

lemma *zmult-zless-mono1*: [| *i*$<*j*; #*0* $< *k* |] ==> *i*$∗*k* $< *j*$∗*k*
apply (*drule zmult-zless-mono2*)
apply (*simp-all add*: *zmult-commute*)
done

lemma *zmult-zless-mono*:
  [| *i* $< *j*; *k* $< *l*; #*0* $< *j*; #*0* $< *k* |] ==> *i*$∗*k* $< *j*$∗*l*
apply (*erule zmult-zless-mono1* [*THEN zless-trans*])
apply *assumption*
apply (*erule zmult-zless-mono2*)
apply *assumption*
done

lemma *zmult-zless-mono1-neg*: [| *i* $< *j*; *k* $< #*0* |] ==> *j*$∗*k* $< *i*$∗*k*
apply (*rule zminus-zless-zminus* [*THEN iffD1*])
apply (*simp del*: *zmult-zminus-right*
      add: *zmult-zminus-right* [*symmetric*] *zmult-zless-mono1 zless-zminus*)
done

lemma *zmult-zless-mono2-neg*: [| *i* $< *j*; *k* $< #*0* |] ==> *k*$∗*j* $< *k*$∗*i*

apply (*rule zminus-zless-zminus* [*THEN iffD1*])
apply (*simp del*: *zmult-zminus*
add: *zmult-zminus* [*symmetric*] *zmult-zless-mono2 zless-zminus*)
done

lemma *zmult-eq-lemma*:
[| *m* ∈ *int*; *n* ∈ *int* |] ==> (*m* = *#0* | *n* = *#0*) <−> (*m*$∗*n* = *#0*)
apply (*case-tac m* $< *#0*)
apply (*auto simp add*: *not-zless-iff-zle zle-def neq-iff-zless*)
apply (*force dest*: *zmult-zless-mono1-neg zmult-zless-mono1*)+
done

lemma *zmult-eq-0-iff* [*iff*]: (*m*$∗*n* = *#0*) <−> (*intify*(*m*) = *#0* | *intify*(*n*) = *#0*)
apply (*simp add*: *zmult-eq-lemma*)
done

lemma *zmult-zless-lemma*:
[| *k* ∈ *int*; *m* ∈ *int*; *n* ∈ *int* |]
==> (*m*$∗*k* $< *n*$∗*k*) <−> ((*#0* $< *k* & *m*$<*n*) | (*k* $< *#0* & *n*$<*m*))
apply (*case-tac k* = *#0*)
apply (*auto simp add*: *neq-iff-zless zmult-zless-mono1 zmult-zless-mono1-neg*)
apply (*auto simp add*: *not-zless-iff-zle*
not-zle-iff-zless [*THEN iff-sym*, *of m*$∗*k*]
not-zle-iff-zless [*THEN iff-sym*, *of m*])
apply (*auto elim*: *notE*
simp add: *zless-imp-zle zmult-zle-mono1 zmult-zle-mono1-neg*)
done

lemma *zmult-zless-cancel2*:
(*m*$∗*k* $< *n*$∗*k*) <−> ((*#0* $< *k* & *m*$<*n*) | (*k* $< *#0* & *n*$<*m*))
apply (*cut-tac k* = *intify* (*k*) **and** *m* = *intify* (*m*) **and** *n* = *intify* (*n*)
in *zmult-zless-lemma*)
apply *auto*
done

lemma *zmult-zless-cancel1*:
(*k*$∗*m* $< *k*$∗*n*) <−> ((*#0* $< *k* & *m*$<*n*) | (*k* $< *#0* & *n*$<*m*))
by (*simp add*: *zmult-commute* [*of k*] *zmult-zless-cancel2*)

lemma *zmult-zle-cancel2*:
(*m*$∗*k* $<= *n*$∗*k*) <−> ((*#0* $< *k* −−> *m*$<=*n*) & (*k* $< *#0* −−> *n*$<=*m*))

by (*auto simp add*: *not-zless-iff-zle* [*THEN iff-sym*] *zmult-zless-cancel2*)

lemma *zmult-zle-cancel1*:
(k$*m $<= k$*n) <-> ((#0 $< k --> m$<=n) & (k $< #0 --> n$<=m))
by (*auto simp add*: *not-zless-iff-zle* [*THEN iff-sym*] *zmult-zless-cancel1*)

lemma *int-eq-iff-zle*: [| m ∈ int; n ∈ int |] ==> m=n <-> (m $<= n & n $<= m)
apply (*blast intro*: *zle-refl zle-anti-sym*)
done

lemma *zmult-cancel2-lemma*:
[| k ∈ int; m ∈ int; n ∈ int |] ==> (m$*k = n$*k) <-> (k=#0 | m=n)
apply (*simp add*: *int-eq-iff-zle* [*of* m$*k] *int-eq-iff-zle* [*of* m])
apply (*auto simp add*: *zmult-zle-cancel2 neq-iff-zless*)
done

lemma *zmult-cancel2* [*simp*]:
(m$*k = n$*k) <-> (intify(k) = #0 | intify(m) = intify(n))
apply (*rule iff-trans*)
apply (*rule-tac* [2] *zmult-cancel2-lemma*)
apply *auto*
done

lemma *zmult-cancel1* [*simp*]:
(k$*m = k$*n) <-> (intify(k) = #0 | intify(m) = intify(n))
by (*simp add*: *zmult-commute* [*of* k] *zmult-cancel2*)

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
apply (*subgoal-tac* r' $+ b $* (q'$-q) $<= r)
prefer *2* **apply** (*simp add*: *zdiff-zmult-distrib2 zadd-ac zcompare-rls*)
apply (*subgoal-tac* #0 $< b $* (#1 $+ q $- q'))
prefer *2*
apply (*erule zle-zless-trans*)
apply (*simp add*: *zdiff-zmult-distrib2 zadd-zmult-distrib2 zadd-ac zcompare-rls*)
apply (*erule zle-zless-trans*)
apply (*simp add*:)
apply (*subgoal-tac* b $* q' $< b $* (#1 $+ q))
prefer *2*
apply (*simp add*: *zdiff-zmult-distrib2 zadd-zmult-distrib2 zadd-ac zcompare-rls*)
apply (*auto elim*: *zless-asym*
simp add: *zmult-zless-cancel1 zless-add1-iff-zle zadd-ac zcompare-rls*)
done

lemma *unique-quotient-lemma-neg*:
[| b$*q' $+ r' $<= b$*q $+ r; r $<= #0; b $< #0; b $< r' |]
==> q $<= q'
apply (*rule-tac* b = $-b **and** r = $-r' **and** r' = $-r
in *unique-quotient-lemma*)
apply (*auto simp del*: *zminus-zadd-distrib*
simp add: *zminus-zadd-distrib* [*symmetric*] *zle-zminus zless-zminus*)
done

lemma *unique-quotient*:
[| *quorem* (<a,b>, <q,r>); *quorem* (<a,b>, <q',r'>); b ∈ *int*; b ~= #0;
q ∈ *int*; q' ∈ *int* |] ==> q = q'
apply (*simp add*: *split-ifs quorem-def neq-iff-zless*)
apply *safe*
apply *simp-all*
apply (*blast intro*: *zle-anti-sym*
dest: *zle-eq-refl* [*THEN unique-quotient-lemma*]
zle-eq-refl [*THEN unique-quotient-lemma-neg*] *sym*)+
done

lemma *unique-remainder*:
[| *quorem* (<a,b>, <q,r>); *quorem* (<a,b>, <q',r'>); b ∈ *int*; b ~= #0;
q ∈ *int*; q' ∈ *int*;
r ∈ *int*; r' ∈ *int* |] ==> r = r'
apply (*subgoal-tac* q = q')
prefer *2* **apply** (*blast intro*: *unique-quotient*)
apply (*simp add*: *quorem-def*)
done

32.2 Correctness of posDivAlg, the Division Algorithm for *a≥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>)
by (*simp add*: *Let-def adjust-def*)

lemma *posDivAlg-termination*:
[| #0 $< b; ~ a $< b |]
==> *nat-of*(a $- #2 $× b $+ #1) < *nat-of*(a $- b $+ #1)
apply (*simp* (*no-asm*) *add*: *zless-nat-conj*)
apply (*simp add*: *not-zless-iff-zle zless-add1-iff-zle zcompare-rls*)
done

lemmas *posDivAlg-unfold* = *def-wfrec* [*OF posDivAlg-def wf-measure*]

```
lemma posDivAlg-eqn:
    [| #0 $< b; a ∈ int; b ∈ int |] ==>
     posDivAlg(<a,b>) =
      (if a$<b then <#0,a> else adjust(b, posDivAlg (<a, #2$*b>)))
apply (rule posDivAlg-unfold [THEN trans])
apply (simp add: vimage-iff not-zless-iff-zle [THEN iff-sym])
apply (blast intro: posDivAlg-termination)
done

lemma posDivAlg-induct-lemma [rule-format]:
  assumes prem:
      !!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>)
apply (rule-tac a = <u,v> in wf-induct)
apply (rule-tac A = int*int and f = %<a,b>.nat-of (a $- b $+ #1)
      in wf-measure)
apply clarify
apply (rule prem)
apply (drule-tac [3] x = <xa, #2 $× y> in spec)
apply auto
apply (simp add: not-zle-iff-zless posDivAlg-termination)
done

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)
apply (subgoal-tac (%<x,y>. P (x,y)) (<u,v>))
apply simp
apply (rule posDivAlg-induct-lemma)
apply (simp (no-asm-use))
apply (rule ih)
apply (auto simp add: u-int v-int)
done

lemma intify-eq-0-iff-zle: intify(m) = #0 <-> (m $<= #0 & #0 $<= m)
apply (simp (no-asm) add: int-eq-iff-zle)
done
```

32.3 Some convenient biconditionals for products of signs

```
lemma zmult-pos: [| #0 $< i; #0 $< j |] ==> #0 $< i $* j
apply (drule zmult-zless-mono1)
```

apply *auto*
done

lemma *zmult-neg*: [| *i* $< *#0*; *j* $< *#0* |] ==> *#0* $< *i* $* *j*
apply (*drule zmult-zless-mono1-neg*)
apply *auto*
done

lemma *zmult-pos-neg*: [| *#0* $< *i*; *j* $< *#0* |] ==> *i* $* *j* $< *#0*
apply (*drule zmult-zless-mono1-neg*)
apply *auto*
done

lemma *int-0-less-lemma*:
  [| *x* ∈ *int*; *y* ∈ *int* |]
   ==> (*#0* $< *x* $* *y*) <-> (*#0* $< *x* & *#0* $< *y* | *x* $< *#0* & *y* $< *#0*)
apply (*auto simp add*: *zle-def not-zless-iff-zle zmult-pos zmult-neg*)
apply (*rule ccontr*)
apply (*rule-tac* [*2*] *ccontr*)
apply (*auto simp add*: *zle-def not-zless-iff-zle*)
apply (*erule-tac P* = *#0*$< *x*$* *y* **in** *rev-mp*)
apply (*erule-tac* [*2*] *P* = *#0*$< *x*$* *y* **in** *rev-mp*)
apply (*drule zmult-pos-neg*, *assumption*)
 prefer *2*
 apply (*drule zmult-pos-neg*, *assumption*)
apply (*auto dest*: *zless-not-sym simp add*: *zmult-commute*)
done

lemma *int-0-less-mult-iff*:
  (*#0* $< *x* $* *y*) <-> (*#0* $< *x* & *#0* $< *y* | *x* $< *#0* & *y* $< *#0*)
apply (*cut-tac x* = *intify* (*x*) **and** *y* = *intify* (*y*) **in** *int-0-less-lemma*)
apply *auto*
done

lemma *int-0-le-lemma*:
  [| *x* ∈ *int*; *y* ∈ *int* |]
   ==> (*#0* $<= *x* $* *y*) <-> (*#0* $<= *x* & *#0* $<= *y* | *x* $<= *#0* & *y* $<= *#0*)
by (*auto simp add*: *zle-def not-zless-iff-zle int-0-less-mult-iff*)

lemma *int-0-le-mult-iff*:
  (*#0* $<= *x* $* *y*) <-> ((*#0* $<= *x* & *#0* $<= *y*) | (*x* $<= *#0* & *y* $<= *#0*))
apply (*cut-tac x* = *intify* (*x*) **and** *y* = *intify* (*y*) **in** *int-0-le-lemma*)
apply *auto*
done

```
lemma zmult-less-0-iff:
    (x $* y $< #0) <-> (#0 $< x & y $< #0 | x $< #0 & #0 $< y)
apply (auto simp add: int-0-le-mult-iff not-zle-iff-zless [THEN iff-sym])
apply (auto dest: zless-not-sym simp add: not-zle-iff-zless)
done

lemma zmult-le-0-iff:
    (x $* y $<= #0) <-> (#0 $<= x & y $<= #0 | x $<= #0 & #0 $<= y)
by (auto dest: zless-not-sym
        simp add: int-0-less-mult-iff not-zless-iff-zle [THEN iff-sym])

lemma posDivAlg-type [rule-format]:
    [| a ∈ int; b ∈ int |] ==> posDivAlg(<a,b>) ∈ int * int
apply (rule-tac u = a and v = b in posDivAlg-induct)
apply assumption+
apply (case-tac #0 $< ba)
 apply (simp add: posDivAlg-eqn adjust-def integ-of-type
            split add: split-if-asm)
 apply clarify
 apply (simp add: int-0-less-mult-iff not-zle-iff-zless)
apply (simp add: not-zless-iff-zle)
apply (subst posDivAlg-unfold)
apply simp
done

lemma posDivAlg-correct [rule-format]:
    [| a ∈ int; b ∈ int |]
     ==> #0 $<= a --> #0 $< b --> quorem (<a,b>, posDivAlg(<a,b>))
apply (rule-tac u = a and v = b in posDivAlg-induct)
apply auto
   apply (simp-all add: quorem-def)
```

base case: a¡b

```
   apply (simp add: posDivAlg-eqn)
  apply (simp add: not-zless-iff-zle [THEN iff-sym])
 apply (simp add: int-0-less-mult-iff)
```

main argument

```
apply (subst posDivAlg-eqn)
apply (simp-all (no-asm-simp))
apply (erule splitE)
apply (rule posDivAlg-type)
apply (simp-all add: int-0-less-mult-iff)
apply (auto simp add: zadd-zmult-distrib2 Let-def)
```

now just linear arithmetic

apply (*simp add*: *not-zle-iff-zless zdiff-zless-iff*)
done

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)
apply (*simp* (*no-asm*) *add*: *zless-nat-conj*)
apply (*simp add*: *zcompare-rls not-zle-iff-zless zless-zdiff-iff* [*THEN iff-sym*]
zless-zminus)
done

lemmas *negDivAlg-unfold* = *def-wfrec* [*OF negDivAlg-def wf-measure*]

lemma *negDivAlg-eqn*:
[| #0 $< b; a : int; b : int |] ==>
negDivAlg(<a,b>) =
(if #0 $<= a$+b then <#−1,a$+b>
else adjust(b, negDivAlg (<a, #2$∗b>)))
apply (*rule negDivAlg-unfold* [*THEN trans*])
apply (*simp* (*no-asm-simp*) *add*: *vimage-iff not-zless-iff-zle* [*THEN iff-sym*])
apply (*blast intro*: *negDivAlg-termination*)
done

lemma *negDivAlg-induct-lemma* [*rule-format*]:
assumes *prem*:
!!a b. [| a ∈ int; b ∈ int;
~ (#0 $<= a $+ b | b $<= #0) −−> P(<a, #2 $∗ b>) |]
==> P(<a,b>)
shows <u,v> ∈ int∗int −−> P(<u,v>)
apply (*rule-tac a* = <u,v> **in** *wf-induct*)
apply (*rule-tac A* = *int∗int* **and** *f* = %<a,b>.nat-of ($− a $− b)
in *wf-measure*)
apply *clarify*
apply (*rule prem*)
apply (*drule-tac* [3] *x* = <xa, #2 $× y> **in** *spec*)
apply *auto*
apply (*simp add*: *not-zle-iff-zless negDivAlg-termination*)
done

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)

apply (*subgoal-tac* (%<*x*,*y*>. *P* (*x*,*y*)) (<*u*,*v*>))
apply *simp*
apply (*rule negDivAlg-induct-lemma*)
apply (*simp* (*no-asm-use*))
apply (*rule ih*)
apply (*auto simp add*: *u-int v-int*)
done

lemma *negDivAlg-type*:
[| *a* ∈ *int*; *b* ∈ *int* |] ==> *negDivAlg*(<*a*,*b*>) ∈ *int* ∗ *int*
apply (*rule-tac u* = *a* **and** *v* = *b* **in** *negDivAlg-induct*)
apply *assumption*+
apply (*case-tac #0* $< *ba*)
apply (*simp add*: *negDivAlg-eqn adjust-def integ-of-type split add*: *split-if-asm*)
apply *clarify*
apply (*simp add*: *int-0-less-mult-iff not-zle-iff-zless*)
apply (*simp add*: *not-zless-iff-zle*)
apply (*subst negDivAlg-unfold*)
apply *simp*
done

lemma *negDivAlg-correct* [*rule-format*]:
[| *a* ∈ *int*; *b* ∈ *int* |]
==> *a* $< *#0* −−> *#0* $< *b* −−> *quorem* (<*a*,*b*>, *negDivAlg*(<*a*,*b*>))
apply (*rule-tac u* = *a* **and** *v* = *b* **in** *negDivAlg-induct*)
apply *auto*
apply (*simp-all add*: *quorem-def*)

base case: *0* $≤ *a* $+ *b*

apply (*simp add*: *negDivAlg-eqn*)
apply (*simp add*: *not-zless-iff-zle* [*THEN iff-sym*])
apply (*simp add*: *int-0-less-mult-iff*)

main argument

apply (*subst negDivAlg-eqn*)
apply (*simp-all* (*no-asm-simp*))
apply (*erule splitE*)
apply (*rule negDivAlg-type*)
apply (*simp-all add*: *int-0-less-mult-iff*)
apply (*auto simp add*: *zadd-zmult-distrib2 Let-def*)

now just linear arithmetic

apply (*simp add*: *not-zle-iff-zless zdiff-zless-iff*)
done

32.5 Existence shown by proving the division algorithm to be correct

lemma *quorem-0*: [|b ≠ #0; b ∈ int|] ==> quorem (<#0,b>, <#0,#0>)
by (*force simp add*: *quorem-def neq-iff-zless*)

lemma *posDivAlg-zero-divisor*: posDivAlg(<a,#0>) = <#0,a>
apply (*subst posDivAlg-unfold*)
apply *simp*
done

lemma *posDivAlg-0* [*simp*]: posDivAlg (<#0,b>) = <#0,#0>
apply (*subst posDivAlg-unfold*)
apply (*simp add*: *not-zle-iff-zless*)
done

lemma *linear-arith-lemma*: ~ (#0 $<= #−1 $+ b) ==> (b $<= #0)
apply (*simp add*: *not-zle-iff-zless*)
apply (*drule zminus-zless-zminus* [*THEN iffD2*])
apply (*simp add*: *zadd-commute zless-add1-iff-zle zle-zminus*)
done

lemma *negDivAlg-minus1* [*simp*]: negDivAlg (<#−1,b>) = <#−1, b$−#1>
apply (*subst negDivAlg-unfold*)
apply (*simp add*: *linear-arith-lemma integ-of-type vimage-iff*)
done

lemma *negateSnd-eq* [*simp*]: negateSnd (<q,r>) = <q, $−r>
apply (*unfold negateSnd-def*)
apply *auto*
done

lemma *negateSnd-type*: qr ∈ int ∗ int ==> negateSnd (qr) ∈ int ∗ int
apply (*unfold negateSnd-def*)
apply *auto*
done

lemma *quorem-neg*:
  [|quorem (<$−a,$−b>, qr); a ∈ int; b ∈ int; qr ∈ int ∗ int|]
  ==> quorem (<a,b>, negateSnd(qr))
apply *clarify*
apply (*auto elim*: *zless-asym simp add*: *quorem-def zless-zminus*)

linear arithmetic from here on

apply (*simp-all add*: *zminus-equation* [*of a*] *zminus-zless*)
apply (*cut-tac* [*2*] z = b **and** w = #0 **in** *zless-linear*)
apply (*cut-tac* [*1*] z = b **and** w = #0 **in** *zless-linear*)
apply *auto*

apply (*blast dest*: *zle-zless-trans*)+
done

lemma *divAlg-correct*:
[|$b \neq$ #*0*; $a \in$ *int*; $b \in$ *int*|] ==> *quorem* (<*a*,*b*>, *divAlg*(<*a*,*b*>))
apply (*auto simp add*: *quorem-0 divAlg-def*)
apply (*safe intro*!: *quorem-neg posDivAlg-correct negDivAlg-correct posDivAlg-type negDivAlg-type*)
apply (*auto simp add*: *quorem-def neq-iff-zless*)

linear arithmetic from here on

apply (*auto simp add*: *zle-def*)
done

lemma *divAlg-type*: [|$a \in$ *int*; $b \in$ *int*|] ==> *divAlg*(<*a*,*b*>) $\in$ *int* $*$ *int*
apply (*auto simp add*: *divAlg-def*)
apply (*auto simp add*: *posDivAlg-type negDivAlg-type negateSnd-type*)
done

lemma *zdiv-intify1* [*simp*]: *intify*(*x*) *zdiv y* = *x zdiv y*
apply (*simp* (*no-asm*) *add*: *zdiv-def*)
done

lemma *zdiv-intify2* [*simp*]: *x zdiv intify*(*y*) = *x zdiv y*
apply (*simp* (*no-asm*) *add*: *zdiv-def*)
done

lemma *zdiv-type* [*iff*,*TC*]: *z zdiv w* $\in$ *int*
apply (*unfold zdiv-def*)
apply (*blast intro*: *fst-type divAlg-type*)
done

lemma *zmod-intify1* [*simp*]: *intify*(*x*) *zmod y* = *x zmod y*
apply (*simp* (*no-asm*) *add*: *zmod-def*)
done

lemma *zmod-intify2* [*simp*]: *x zmod intify*(*y*) = *x zmod y*
apply (*simp* (*no-asm*) *add*: *zmod-def*)
done

lemma *zmod-type* [*iff*,*TC*]: *z zmod w* $\in$ *int*
apply (*unfold zmod-def*)
apply (*rule snd-type*)
apply (*blast intro*: *divAlg-type*)
done

lemma *DIVISION-BY-ZERO-ZDIV*: *a zdiv #0* = *#0*
apply (*simp* (*no-asm*) *add*: *zdiv-def divAlg-def posDivAlg-zero-divisor*)
done

lemma *DIVISION-BY-ZERO-ZMOD*: *a zmod #0* = *intify*(*a*)
apply (*simp* (*no-asm*) *add*: *zmod-def divAlg-def posDivAlg-zero-divisor*)
done

lemma *raw-zmod-zdiv-equality*:
  [| $a \in int$; $b \in int$ |] ==> *a* = *b* $* (*a zdiv b*) $+ (*a zmod b*)
apply (*case-tac b* = *#0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*cut-tac a* = *a* **and** *b* = *b* **in** *divAlg-correct*)
apply (*auto simp add*: *quorem-def zdiv-def zmod-def split-def*)
done

lemma *zmod-zdiv-equality*: *intify*(*a*) = *b* $* (*a zdiv b*) $+ (*a zmod b*)
apply (*rule trans*)
apply (*rule-tac b* = *intify* (*b*) **in** *raw-zmod-zdiv-equality*)
apply *auto*
done

lemma *pos-mod*: *#0* $< *b* ==> *#0* $<= *a zmod b* & *a zmod b* $< *b*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **in** *divAlg-correct*)
apply (*auto simp add*: *intify-eq-0-iff-zle quorem-def zmod-def split-def*)
apply (*blast dest*: *zle-zless-trans*)+
done

lemmas *pos-mod-sign* = *pos-mod* [*THEN conjunct1*, *standard*]
and *pos-mod-bound* = *pos-mod* [*THEN conjunct2*, *standard*]

lemma *neg-mod*: *b* $< *#0* ==> *a zmod b* $<= *#0* & *b* $< *a zmod b*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **in** *divAlg-correct*)
apply (*auto simp add*: *intify-eq-0-iff-zle quorem-def zmod-def split-def*)
apply (*blast dest*: *zle-zless-trans*)
apply (*blast dest*: *zless-trans*)+
done

lemmas *neg-mod-sign* = *neg-mod* [*THEN conjunct1*, *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>)
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (auto simp add: quorem-def neq-iff-zless pos-mod-sign pos-mod-bound
                 neg-mod-sign neg-mod-bound)
done


lemma quorem-div:
    [| quorem(<a,b>,<q,r>);  b ≠ #0;  a ∈ int;  b ∈ int;  q ∈ int |]
     ==> a zdiv b = q
by (blast intro: quorem-div-mod [THEN unique-quotient])

lemma quorem-mod:
    [| quorem(<a,b>,<q,r>); b ≠ #0; a ∈ int; b ∈ int; q ∈ int; r ∈ int |]
     ==> a zmod b = r
by (blast intro: quorem-div-mod [THEN unique-remainder])

lemma zdiv-pos-pos-trivial-raw:
    [| a ∈ int;  b ∈ int;  #0 $<= a;  a $< b |] ==> a zdiv b = #0
apply (rule quorem-div)
apply (auto simp add: quorem-def)

apply (blast dest: zle-zless-trans)+
done

lemma zdiv-pos-pos-trivial: [| #0 $<= a;  a $< b |] ==> a zdiv b = #0
apply (cut-tac a = intify (a) and b = intify (b)
      in zdiv-pos-pos-trivial-raw)
apply auto
done

lemma zdiv-neg-neg-trivial-raw:
    [| a ∈ int;  b ∈ int;  a $<= #0;  b $< a |] ==> a zdiv b = #0
apply (rule-tac r = a in quorem-div)
apply (auto simp add: quorem-def)

apply (blast dest: zle-zless-trans zless-trans)+
done

lemma zdiv-neg-neg-trivial: [| a $<= #0;  b $< a |] ==> a zdiv b = #0
apply (cut-tac a = intify (a) and b = intify (b)
      in zdiv-neg-neg-trivial-raw)
apply auto
done
```

lemma *zadd-le-0-lemma*: [| a$+b $<= #0; #0 $< a; #0 $< b |] ==> *False*
apply (*drule-tac z' = #0* **and** *z = b* **in** *zadd-zless-mono*)
apply (*auto simp add*: *zle-def*)
apply (*blast dest*: *zless-trans*)
done

lemma *zdiv-pos-neg-trivial-raw*:
  [| a ∈ int; b ∈ int; #0 $< a; a$+b $<= #0 |] ==> a zdiv b = #−1
apply (*rule-tac r = a $+ b* **in** *quorem-div*)
apply (*auto simp add*: *quorem-def*)

apply (*blast dest*: *zadd-le-0-lemma zle-zless-trans*)+
done

lemma *zdiv-pos-neg-trivial*: [| #0 $< a; a$+b $<= #0 |] ==> a zdiv b = #−1
apply (*cut-tac a = intify (a)* **and** *b = intify (b)*
  in *zdiv-pos-neg-trivial-raw*)
apply *auto*
done

lemma *zmod-pos-pos-trivial-raw*:
  [| a ∈ int; b ∈ int; #0 $<= a; a $< b |] ==> a zmod b = a
apply (*rule-tac q = #0* **in** *quorem-mod*)
apply (*auto simp add*: *quorem-def*)

apply (*blast dest*: *zle-zless-trans*)+
done

lemma *zmod-pos-pos-trivial*: [| #0 $<= a; a $< b |] ==> a zmod b = intify(a)
apply (*cut-tac a = intify (a)* **and** *b = intify (b)*
  in *zmod-pos-pos-trivial-raw*)
apply *auto*
done

lemma *zmod-neg-neg-trivial-raw*:
  [| a ∈ int; b ∈ int; a $<= #0; b $< a |] ==> a zmod b = a
apply (*rule-tac q = #0* **in** *quorem-mod*)
apply (*auto simp add*: *quorem-def*)

apply (*blast dest*: *zle-zless-trans zless-trans*)+
done

lemma *zmod-neg-neg-trivial*: [| a $<= #0; b $< a |] ==> a zmod b = intify(a)
apply (*cut-tac a = intify (a)* **and** *b = intify (b)*
  in *zmod-neg-neg-trivial-raw*)
apply *auto*

done

lemma *zmod-pos-neg-trivial-raw*:
[| $a \in int$; $b \in int$; *#0* \$< *a*; *a*\$+*b* \$<= *#0* |] ==> *a zmod b* = *a*\$+*b*
apply (*rule-tac q* = *#−1* **in** *quorem-mod*)
apply (*auto simp add*: *quorem-def*)

apply (*blast dest*: *zadd-le-0-lemma zle-zless-trans*)+
done

lemma *zmod-pos-neg-trivial*: [| *#0* \$< *a*; *a*\$+*b* \$<= *#0* |] ==> *a zmod b* = *a*\$+*b*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*)
in *zmod-pos-neg-trivial-raw*)
apply *auto*
done

lemma *zdiv-zminus-zminus-raw*:
[|$a \in int$; $b \in int$|] ==> (\$−*a*) *zdiv* (\$−*b*) = *a zdiv b*
apply (*case-tac b* = *#0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*subst quorem-div-mod* [*THEN quorem-neg*, *simplified*, *THEN quorem-div*])
apply *auto*
done

lemma *zdiv-zminus-zminus* [*simp*]: (\$−*a*) *zdiv* (\$−*b*) = *a zdiv b*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **in** *zdiv-zminus-zminus-raw*)
apply *auto*
done

lemma *zmod-zminus-zminus-raw*:
[|$a \in int$; $b \in int$|] ==> (\$−*a*) *zmod* (\$−*b*) = \$− (*a zmod b*)
apply (*case-tac b* = *#0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*subst quorem-div-mod* [*THEN quorem-neg*, *simplified*, *THEN quorem-mod*])
apply *auto*
done

lemma *zmod-zminus-zminus* [*simp*]: (\$−*a*) *zmod* (\$−*b*) = \$− (*a zmod b*)
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **in** *zmod-zminus-zminus-raw*)
apply *auto*
done

32.6 division of a number by itself

lemma *self-quotient-aux1*: [| *#0* $< *a*; *a* = *r* $+ *a*$**q*; *r* $< *a* |] ==> *#1* $<= *q*
apply (*subgoal-tac #0* $< *a*$**q*)
apply (*cut-tac w* = *#0* **and** *z* = *q* **in** *add1-zle-iff*)
apply (*simp add*: *int-0-less-mult-iff*)
apply (*blast dest*: *zless-trans*)

apply (*drule-tac t* = %*x*. *x* $− *r* **in** *subst-context*)
apply (*drule sym*)
apply (*simp add*: *zcompare-rls*)
done

lemma *self-quotient-aux2*: [| *#0* $< *a*; *a* = *r* $+ *a*$**q*; *#0* $<= *r* |] ==> *q* $<= *#1*
apply (*subgoal-tac #0* $<= *a*$* (*#1*$−*q*))
 apply (*simp add*: *int-0-le-mult-iff zcompare-rls*)
 apply (*blast dest*: *zle-zless-trans*)
apply (*simp add*: *zdiff-zmult-distrib2*)
apply (*drule-tac t* = %*x*. *x* $− *a* $* *q* **in** *subst-context*)
apply (*simp add*: *zcompare-rls*)
done

lemma *self-quotient*:
  [| *quorem*(<*a*,*a*>,<*q*,*r*>); *a* ∈ *int*; *q* ∈ *int*; *a* ≠ *#0*|] ==> *q* = *#1*
apply (*simp add*: *split-ifs quorem-def neq-iff-zless*)
apply (*rule zle-anti-sym*)
apply *safe*
apply *auto*
prefer *4* **apply** (*blast dest*: *zless-trans*)
apply (*blast dest*: *zless-trans*)
apply (*rule-tac* [*3*] *a* = $−*a* **and** *r* = $−*r* **in** *self-quotient-aux1*)
apply (*rule-tac a* = $−*a* **and** *r* = $−*r* **in** *self-quotient-aux2*)
apply (*rule-tac* [*6*] *zminus-equation* [*THEN iffD1*])
apply (*rule-tac* [*2*] *zminus-equation* [*THEN iffD1*])
apply (*force intro*: *self-quotient-aux1 self-quotient-aux2*
 simp add: *zadd-commute zmult-zminus*)+
done

lemma *self-remainder*:
  [|*quorem*(<*a*,*a*>,<*q*,*r*>); *a* ∈ *int*; *q* ∈ *int*; *r* ∈ *int*; *a* ≠ *#0*|] ==> *r* = *#0*
apply (*frule self-quotient*)
apply (*auto simp add*: *quorem-def*)
done

lemma *zdiv-self-raw*: [|*a* ≠ *#0*; *a* ∈ *int*|] ==> *a zdiv a* = *#1*
apply (*blast intro*: *quorem-div-mod* [*THEN self-quotient*])
done

lemma *zdiv-self* [*simp*]: *intify*(*a*) ≠ #*0* ==> *a zdiv a* = #*1*
apply (*drule zdiv-self-raw*)
apply *auto*
done

lemma *zmod-self-raw*: *a* ∈ *int* ==> *a zmod a* = #*0*
apply (*case-tac a* = #*0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*blast intro*: *quorem-div-mod* [*THEN self-remainder*])
done

lemma *zmod-self* [*simp*]: *a zmod a* = #*0*
apply (*cut-tac a* = *intify* (*a*) **in** *zmod-self-raw*)
apply *auto*
done

32.7 Computation of division and remainder

lemma *zdiv-zero* [*simp*]: #*0 zdiv b* = #*0*
apply (*simp* (*no-asm*) *add*: *zdiv-def divAlg-def*)
done

lemma *zdiv-eq-minus1*: #*0* $< *b* ==> #−*1 zdiv b* = #−*1*
apply (*simp* (*no-asm-simp*) *add*: *zdiv-def divAlg-def*)
done

lemma *zmod-zero* [*simp*]: #*0 zmod b* = #*0*
apply (*simp* (*no-asm*) *add*: *zmod-def divAlg-def*)
done

lemma *zdiv-minus1*: #*0* $< *b* ==> #−*1 zdiv b* = #−*1*
apply (*simp* (*no-asm-simp*) *add*: *zdiv-def divAlg-def*)
done

lemma *zmod-minus1*: #*0* $< *b* ==> #−*1 zmod b* = *b* $− #*1*
apply (*simp* (*no-asm-simp*) *add*: *zmod-def divAlg-def*)
done

lemma *zdiv-pos-pos*: [| #*0* $< *a*; #*0* $<= *b* |]
    ==> *a zdiv b* = *fst* (*posDivAlg*(<*intify*(*a*), *intify*(*b*)>))
apply (*simp* (*no-asm-simp*) *add*: *zdiv-def divAlg-def*)
apply (*auto simp add*: *zle-def*)
done

lemma *zmod-pos-pos*:
    [| #*0* $< *a*; #*0* $<= *b* |]

==> *a zmod b = snd (posDivAlg(<intify(a), intify(b)>))*
apply (*simp* (*no-asm-simp*) *add*: *zmod-def divAlg-def*)
apply (*auto simp add*: *zle-def*)
done

lemma *zdiv-neg-pos*:
[| *a* $< #0; #0 $< *b* |]
==> *a zdiv b = fst (negDivAlg(<intify(a), intify(b)>))*
apply (*simp* (*no-asm-simp*) *add*: *zdiv-def divAlg-def*)
apply (*blast dest*: *zle-zless-trans*)
done

lemma *zmod-neg-pos*:
[| *a* $< #0; #0 $< *b* |]
==> *a zmod b = snd (negDivAlg(<intify(a), intify(b)>))*
apply (*simp* (*no-asm-simp*) *add*: *zmod-def divAlg-def*)
apply (*blast dest*: *zle-zless-trans*)
done

lemma *zdiv-pos-neg*:
[| #0 $< *a*; *b* $< #0 |]
==> *a zdiv b = fst (negateSnd(negDivAlg (<$−a, $−b>)))*
apply (*simp* (*no-asm-simp*) *add*: *zdiv-def divAlg-def intify-eq-0-iff-zle*)
apply *auto*
apply (*blast dest*: *zle-zless-trans*)+
apply (*blast dest*: *zless-trans*)
apply (*blast intro*: *zless-imp-zle*)
done

lemma *zmod-pos-neg*:
[| #0 $< *a*; *b* $< #0 |]
==> *a zmod b = snd (negateSnd(negDivAlg (<$−a, $−b>)))*
apply (*simp* (*no-asm-simp*) *add*: *zmod-def divAlg-def intify-eq-0-iff-zle*)
apply *auto*
apply (*blast dest*: *zle-zless-trans*)+
apply (*blast dest*: *zless-trans*)
apply (*blast intro*: *zless-imp-zle*)
done

lemma *zdiv-neg-neg*:
[| *a* $< #0; *b* $<= #0 |]
==> *a zdiv b = fst (negateSnd(posDivAlg(<$−a, $−b>)))*
apply (*simp* (*no-asm-simp*) *add*: *zdiv-def divAlg-def*)

apply *auto*
apply (*blast dest*!: *zle-zless-trans*)+
done

lemma *zmod-neg-neg*:
[| *a* $< *#0*; *b* $<= *#0* |]
==> *a zmod b* = *snd* (*negateSnd*(*posDivAlg*(<$-*a*, $-*b*>)))
apply (*simp* (*no-asm-simp*) *add*: *zmod-def divAlg-def*)
apply *auto*
apply (*blast dest*!: *zle-zless-trans*)+
done

declare *zdiv-pos-pos* [*of integ-of* (*v*) *integ-of* (*w*), *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*
apply (*cut-tac a* = *a* **and** *b* = *#1* **in** *pos-mod-sign*)
apply (*cut-tac* [*2*] *a* = *a* **and** *b* = *#1* **in** *pos-mod-bound*)
apply *auto*

apply (*drule add1-zle-iff* [*THEN iffD2*])
apply (*rule zle-anti-sym*)
apply *auto*
done

lemma *zdiv-1* [*simp*]: *a zdiv #1* = *intify*(*a*)
apply (*cut-tac a* = *a* **and** *b* = *#1* **in** *zmod-zdiv-equality*)
apply *auto*
done

lemma *zmod-minus1-right* [*simp*]: *a zmod #-1* = *#0*
apply (*cut-tac a* = *a* **and** *b* = *#-1* **in** *neg-mod-sign*)
apply (*cut-tac* [*2*] *a* = *a* **and** *b* = *#-1* **in** *neg-mod-bound*)
apply *auto*

apply (*drule add1-zle-iff* [*THEN iffD2*])
apply (*rule zle-anti-sym*)
apply *auto*

done

lemma *zdiv-minus1-right-raw*: $a \in int ==> a\ zdiv\ \#{-1} = \$-a$
apply (*cut-tac* $a = a$ **and** $b = \#{-1}$ **in** *zmod-zdiv-equality*)
apply *auto*
apply (*rule equation-zminus* [*THEN iffD2*])
apply *auto*
done

lemma *zdiv-minus1-right*: $a\ zdiv\ \#{-1} = \$-a$
apply (*cut-tac* $a = intify\ (a)$ **in** *zdiv-minus1-right-raw*)
apply *auto*
done
declare *zdiv-minus1-right* [*simp*]

32.8 Monotonicity in the first argument (divisor)

lemma *zdiv-mono1*: $[|\ a\ \$<= a';\ \ \#0\ \$< b\ |] ==> a\ zdiv\ b\ \$<= a'\ zdiv\ b$
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*cut-tac* $a = a'$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*rule unique-quotient-lemma*)
apply (*erule subst*)
apply (*erule subst*)
apply (*simp-all* (*no-asm-simp*) *add*: *pos-mod-sign pos-mod-bound*)
done

lemma *zdiv-mono1-neg*: $[|\ a\ \$<= a';\ \ b\ \$< \#0\ |] ==> a'\ zdiv\ b\ \$<= a\ zdiv\ b$
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*cut-tac* $a = a'$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*rule unique-quotient-lemma-neg*)
apply (*erule subst*)
apply (*erule subst*)
apply (*simp-all* (*no-asm-simp*) *add*: *neg-mod-sign neg-mod-bound*)
done

32.9 Monotonicity in the second argument (dividend)

lemma *q-pos-lemma*:
  $[|\ \#0\ \$<= b'\$*q'\ \$+ r';\ r'\ \$< b';\ \ \#0\ \$< b'\ |] ==> \#0\ \$<= q'$
apply (*subgoal-tac* $\#0\ \$< b'\$*\ (q'\ \$+ \#1)$)
 apply (*simp add*: *int-0-less-mult-iff*)
 apply (*blast dest*: *zless-trans intro*: *zless-add1-iff-zle* [*THEN iffD1*])
apply (*simp add*: *zadd-zmult-distrib2*)
apply (*erule zle-zless-trans*)
apply (*erule zadd-zless-mono2*)
done

lemma *zdiv-mono2-lemma*:
  $[|\ b\$*q\ \$+ r = b'\$*q'\ \$+ r';\ \ \#0\ \$<= b'\$*q'\ \$+ r';$
    $r'\ \$< b';\ \ \#0\ \$<= r;\ \ \#0\ \$< b';\ \ b'\ \$<= b\ |]$

```
    ==> q $<= q′
apply (frule q-pos-lemma, assumption+)
apply (subgoal-tac b$*q $< b$* (q′ $+ #1))
 apply (simp add: zmult-zless-cancel1)
 apply (force dest: zless-add1-iff-zle [THEN iffD1] zless-trans zless-zle-trans)
apply (subgoal-tac b$*q = r′ $- r $+ b′$*q′)
 prefer 2 apply (simp add: zcompare-rls)
apply (simp (no-asm-simp) add: zadd-zmult-distrib2)
apply (subst zadd-commute [of b $× q′], rule zadd-zless-mono)
 prefer 2 apply (blast intro: zmult-zle-mono1)
apply (subgoal-tac r′ $+ #0 $< b $+ r)
 apply (simp add: zcompare-rls)
apply (rule zadd-zless-mono)
 apply auto
apply (blast dest: zless-zle-trans)
done


lemma zdiv-mono2-raw:
    [| #0 $<= a;  #0 $< b′;  b′ $<= b;  a ∈ int |]
     ==> a zdiv b $<= a zdiv b′
apply (subgoal-tac #0 $< b)
 prefer 2 apply (blast dest: zless-zle-trans)
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (cut-tac a = a and b = b′ in zmod-zdiv-equality)
apply (rule zdiv-mono2-lemma)
apply (erule subst)
apply (erule subst)
apply (simp-all add: pos-mod-sign pos-mod-bound)
done

lemma zdiv-mono2:
    [| #0 $<= a;  #0 $< b′;  b′ $<= b |]
     ==> a zdiv b $<= a zdiv b′
apply (cut-tac a = intify (a) in zdiv-mono2-raw)
apply auto
done

lemma q-neg-lemma:
    [| b′$*q′ $+ r′ $< #0;  #0 $<= r′;  #0 $< b′ |] ==> q′ $< #0
apply (subgoal-tac b′$*q′ $< #0)
 prefer 2 apply (force intro: zle-zless-trans)
apply (simp add: zmult-less-0-iff)
apply (blast dest: zless-trans)
done



lemma zdiv-mono2-neg-lemma:
```

[| b$*q $+ r = b'$*q' $+ r'; b'$*q' $+ r' $< #0;
r $< b; #0 $<= r'; #0 $< b'; b' $<= b |]
==> q' $<= q
apply (*subgoal-tac #0 $< b*)
prefer *2* **apply** (*blast dest*: *zless-zle-trans*)
apply (*frule q-neg-lemma*, *assumption+*)
apply (*subgoal-tac b$*q' $< b$* (q $+ #1)*)
apply (*simp add*: *zmult-zless-cancel1*)
apply (*blast dest*: *zless-trans zless-add1-iff-zle* [*THEN iffD1*])
apply (*simp* (*no-asm-simp*) *add*: *zadd-zmult-distrib2*)
apply (*subgoal-tac b$*q' $<= b'$*q'*)
prefer *2*
apply (*simp add*: *zmult-zle-cancel2*)
apply (*blast dest*: *zless-trans*)
apply (*subgoal-tac b'$*q' $+ r $< b $+ (b$*q $+ r)*)
prefer *2*
apply (*erule ssubst*)
apply *simp*
apply (*drule-tac w' = r* **and** *z' = #0* **in** *zadd-zless-mono*)
apply (*assumption*)
apply *simp*
apply (*simp* (*no-asm-use*) *add*: *zadd-commute*)
apply (*rule zle-zless-trans*)
prefer *2* **apply** (*assumption*)
apply (*simp* (*no-asm-simp*) *add*: *zmult-zle-cancel2*)
apply (*blast dest*: *zless-trans*)
done

lemma *zdiv-mono2-neg-raw*:
[| a $< #0; #0 $< b'; b' $<= b; a ∈ int |]
==> a zdiv b' $<= a zdiv b
apply (*subgoal-tac #0 $< b*)
prefer *2* **apply** (*blast dest*: *zless-zle-trans*)
apply (*cut-tac a = a* **and** *b = b* **in** *zmod-zdiv-equality*)
apply (*cut-tac a = a* **and** *b = b'* **in** *zmod-zdiv-equality*)
apply (*rule zdiv-mono2-neg-lemma*)
apply (*erule subst*)
apply (*erule subst*)
apply (*simp-all add*: *pos-mod-sign pos-mod-bound*)
done

lemma *zdiv-mono2-neg*: [| a $< #0; #0 $< b'; b' $<= b |]
==> a zdiv b' $<= a zdiv b
apply (*cut-tac a = intify (a)* **in** *zdiv-mono2-neg-raw*)
apply *auto*
done

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>)
apply (auto simp add: split-ifs quorem-def neq-iff-zless zadd-zmult-distrib2
pos-mod-sign pos-mod-bound neg-mod-sign neg-mod-bound)
apply (auto intro: raw-zmod-zdiv-equality)
done

lemma *zdiv-zmult1-eq-raw*:
[|b ∈ int; c ∈ int|]
==> (a$*b) zdiv c = a$*(b zdiv c) $+ a$*(b zmod c) zdiv c
apply (case-tac c = #0)
apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
apply (rule quorem-div-mod [THEN zmult1-lemma, THEN quorem-div])
apply auto
done

lemma *zdiv-zmult1-eq*: (a$*b) zdiv c = a$*(b zdiv c) $+ a$*(b zmod c) zdiv c
apply (cut-tac b = intify (b) **and** c = intify (c) **in** zdiv-zmult1-eq-raw)
apply auto
done

lemma *zmod-zmult1-eq-raw*:
[|b ∈ int; c ∈ int|] ==> (a$*b) zmod c = a$*(b zmod c) zmod c
apply (case-tac c = #0)
apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
apply (rule quorem-div-mod [THEN zmult1-lemma, THEN quorem-mod])
apply auto
done

lemma *zmod-zmult1-eq*: (a$*b) zmod c = a$*(b zmod c) zmod c
apply (cut-tac b = intify (b) **and** c = intify (c) **in** zmod-zmult1-eq-raw)
apply auto
done

lemma *zmod-zmult1-eq'*: (a$*b) zmod c = ((a zmod c) $* b) zmod c
apply (rule trans)
apply (rule-tac b = (b $* a) zmod c **in** trans)
apply (rule-tac [2] zmod-zmult1-eq)
apply (simp-all (no-asm) add: zmult-commute)
done

lemma *zmod-zmult-distrib*: (a$*b) zmod c = ((a zmod c) $* (b zmod c)) zmod c
apply (rule zmod-zmult1-eq' [THEN trans])
apply (rule zmod-zmult1-eq)
done

lemma *zdiv-zmult-self1* [simp]: intify(b) ≠ #0 ==> (a$*b) zdiv b = intify(a)

apply (*simp* (*no-asm-simp*) *add*: *zdiv-zmult1-eq*)
done

lemma *zdiv-zmult-self2* [*simp*]: *intify*(*b*) ≠ *#0* ==> (*b*$∗*a*) *zdiv* *b* = *intify*(*a*)
apply (*subst zmult-commute* , *erule zdiv-zmult-self1*)
done

lemma *zmod-zmult-self1* [*simp*]: (*a*$∗*b*) *zmod* *b* = *#0*
apply (*simp* (*no-asm*) *add*: *zmod-zmult1-eq*)
done

lemma *zmod-zmult-self2* [*simp*]: (*b*$∗*a*) *zmod* *b* = *#0*
apply (*simp* (*no-asm*) *add*: *zmult-commute zmod-zmult1-eq*)
done

lemma *zadd1-lemma*:
  [| *quorem*(<*a*,*c*>, <*aq*,*ar*>); *quorem*(<*b*,*c*>, <*bq*,*br*>);
   c ∈ *int*; *c* ≠ *#0* |]
  ==> *quorem* (<*a*$+*b*, *c*>, <*aq* $+ *bq* $+ (*ar*$+*br*) *zdiv* *c*, (*ar*$+*br*) *zmod* *c*>)
apply (*auto simp add*: *split-ifs quorem-def neq-iff-zless zadd-zmult-distrib2*
  pos-mod-sign pos-mod-bound neg-mod-sign neg-mod-bound)
apply (*auto intro*: *raw-zmod-zdiv-equality*)
done

lemma *zdiv-zadd1-eq-raw*:
  [|*a* ∈ *int*; *b* ∈ *int*; *c* ∈ *int*|] ==>
  (*a*$+*b*) *zdiv* *c* = *a zdiv c* $+ *b zdiv c* $+ ((*a zmod c* $+ *b zmod c*) *zdiv c*)
apply (*case-tac c* = *#0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*blast intro*: *zadd1-lemma* [*OF quorem-div-mod quorem-div-mod*,
  THEN quorem-div])
done

lemma *zdiv-zadd1-eq*:
  (*a*$+*b*) *zdiv* *c* = *a zdiv c* $+ *b zdiv c* $+ ((*a zmod c* $+ *b zmod c*) *zdiv c*)
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **and** *c* = *intify* (*c*)
  in *zdiv-zadd1-eq-raw*)
apply *auto*
done

lemma *zmod-zadd1-eq-raw*:
  [|*a* ∈ *int*; *b* ∈ *int*; *c* ∈ *int*|]
  ==> (*a*$+*b*) *zmod* *c* = (*a zmod c* $+ *b zmod c*) *zmod c*
apply (*case-tac c* = *#0*)

apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*blast intro*: *zadd1-lemma* [*OF quorem-div-mod quorem-div-mod*,
THEN quorem-mod])
done

lemma *zmod-zadd1-eq*: (*a*$+*b*) *zmod c* = (*a zmod c* $+ *b zmod c*) *zmod c*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **and** *c* = *intify* (*c*)
in *zmod-zadd1-eq-raw*)
apply *auto*
done

lemma *zmod-div-trivial-raw*:
[|*a* ∈ *int*; *b* ∈ *int*|] ==> (*a zmod b*) *zdiv b* = #*0*
apply (*case-tac b* = #*0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*auto simp add*: *neq-iff-zless pos-mod-sign pos-mod-bound*
zdiv-pos-pos-trivial neg-mod-sign neg-mod-bound zdiv-neg-neg-trivial)
done

lemma *zmod-div-trivial* [*simp*]: (*a zmod b*) *zdiv b* = #*0*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **in** *zmod-div-trivial-raw*)
apply *auto*
done

lemma *zmod-mod-trivial-raw*:
[|*a* ∈ *int*; *b* ∈ *int*|] ==> (*a zmod b*) *zmod b* = *a zmod b*
apply (*case-tac b* = #*0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*auto simp add*: *neq-iff-zless pos-mod-sign pos-mod-bound*
zmod-pos-pos-trivial neg-mod-sign neg-mod-bound zmod-neg-neg-trivial)
done

lemma *zmod-mod-trivial* [*simp*]: (*a zmod b*) *zmod b* = *a zmod b*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **in** *zmod-mod-trivial-raw*)
apply *auto*
done

lemma *zmod-zadd-left-eq*: (*a*$+*b*) *zmod c* = ((*a zmod c*) $+ *b*) *zmod c*
apply (*rule trans* [*symmetric*])
apply (*rule zmod-zadd1-eq*)
apply (*simp* (*no-asm*))
apply (*rule zmod-zadd1-eq* [*symmetric*])
done

lemma *zmod-zadd-right-eq*: (*a*$+*b*) *zmod c* = (*a* $+ (*b zmod c*)) *zmod c*
apply (*rule trans* [*symmetric*])
apply (*rule zmod-zadd1-eq*)
apply (*simp* (*no-asm*))
apply (*rule zmod-zadd1-eq* [*symmetric*])

done

lemma *zdiv-zadd-self1* [*simp*]:
 intify(*a*) ≠ *#0* ==> (*a*$+*b*) *zdiv a* = *b zdiv a* $+ *#1*
by (*simp* (*no-asm-simp*) *add*: *zdiv-zadd1-eq*)

lemma *zdiv-zadd-self2* [*simp*]:
 intify(*a*) ≠ *#0* ==> (*b*$+*a*) *zdiv a* = *b zdiv a* $+ *#1*
by (*simp* (*no-asm-simp*) *add*: *zdiv-zadd1-eq*)

lemma *zmod-zadd-self1* [*simp*]: (*a*$+*b*) *zmod a* = *b zmod a*
apply (*case-tac a* = *#0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*simp* (*no-asm-simp*) *add*: *zmod-zadd1-eq*)
done

lemma *zmod-zadd-self2* [*simp*]: (*b*$+*a*) *zmod a* = *b zmod a*
apply (*case-tac a* = *#0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*simp* (*no-asm-simp*) *add*: *zmod-zadd1-eq*)
done

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*
apply (*subgoal-tac b* $* (*c* $− *q zmod c*) $< *r* $* *#1*)
apply (*simp add*: *zdiff-zmult-distrib2 zadd-commute zcompare-rls*)
apply (*rule zle-zless-trans*)
apply (*erule-tac* [*2*] *zmult-zless-mono1*)
apply (*rule zmult-zle-mono2-neg*)
apply (*auto simp add*: *zcompare-rls zadd-commute add1-zle-iff pos-mod-bound*)
apply (*blast intro*: *zless-imp-zle dest*: *zless-zle-trans*)
done

lemma *zdiv-zmult2-aux2*:
 [| *#0* $< *c*; *b* $< *r*; *r* $<= *#0* |] ==> *b* $* (*q zmod c*) $+ *r* $<= *#0*
apply (*subgoal-tac b* $* (*q zmod c*) $<= *#0*)
prefer *2*
apply (*simp add*: *zmult-le-0-iff pos-mod-sign*)
apply (*blast intro*: *zless-imp-zle dest*: *zless-zle-trans*)

apply (*drule zadd-zle-mono*)
apply *assumption*
apply (*simp add*: *zadd-commute*)
done

lemma *zdiv-zmult2-aux3*:

```
  [| #0 $< c;  #0 $<= r;  r $< b |] ==> #0 $<= b $* (q zmod c) $+ r
apply (subgoal-tac #0 $<= b $* (q zmod c))
 prefer 2
 apply (simp add: int-0-le-mult-iff pos-mod-sign)
 apply (blast intro: zless-imp-zle dest: zle-zless-trans)

apply (drule zadd-zle-mono)
apply assumption
apply (simp add: zadd-commute)
done

lemma zdiv-zmult2-aux4:
  [| #0 $< c; #0 $<= r; r $< b |] ==> b $* (q zmod c) $+ r $< b $* c
apply (subgoal-tac r $* #1 $< b $* (c $- q zmod c))
apply (simp add: zdiff-zmult-distrib2 zadd-commute zcompare-rls)
apply (rule zless-zle-trans)
apply (erule zmult-zless-mono1)
apply (rule-tac [2] zmult-zle-mono2)
apply (auto simp add: zcompare-rls zadd-commute add1-zle-iff pos-mod-bound)
apply (blast intro: zless-imp-zle dest: zle-zless-trans)
done

lemma zdiv-zmult2-lemma:
  [| 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>)
apply (auto simp add: zmult-ac zmod-zdiv-equality [symmetric] quorem-def
          neq-iff-zless int-0-less-mult-iff
          zadd-zmult-distrib2 [symmetric] zdiv-zmult2-aux1 zdiv-zmult2-aux2
          zdiv-zmult2-aux3 zdiv-zmult2-aux4)
apply (blast dest: zless-trans)+
done

lemma zdiv-zmult2-eq-raw:
  [|#0 $< c;  a ∈ int;  b ∈ int|] ==> a zdiv (b$*c) = (a zdiv b) zdiv c
apply (case-tac b = #0)
 apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
apply (rule quorem-div-mod [THEN zdiv-zmult2-lemma, THEN quorem-div])
apply (auto simp add: intify-eq-0-iff-zle)
apply (blast dest: zle-zless-trans)
done

lemma zdiv-zmult2-eq: #0 $< c ==> a zdiv (b$*c) = (a zdiv b) zdiv c
apply (cut-tac a = intify (a) and b = intify (b) in zdiv-zmult2-eq-raw)
apply auto
done

lemma zmod-zmult2-eq-raw:
  [|#0 $< c;  a ∈ int;  b ∈ int|]
   ==> a zmod (b$*c) = b$*(a zdiv b zmod c) $+ a zmod b
```

apply (*case-tac b = #0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*rule quorem-div-mod* [*THEN zdiv-zmult2-lemma*, *THEN quorem-mod*])
apply (*auto simp add*: *intify-eq-0-iff-zle*)
apply (*blast dest*: *zle-zless-trans*)
done

lemma *zmod-zmult2-eq*:
#0 \$< *c* ==> *a zmod* (*b*\$**c*) = *b*\$*(*a zdiv b zmod c*) \$+ *a zmod b*
apply (*cut-tac a* = *intify* (*a*) **and** *b* = *intify* (*b*) **in** *zmod-zmult2-eq-raw*)
apply *auto*
done

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*
apply (*subst zdiv-zmult2-eq*)
apply *auto*
done

lemma *zdiv-zmult-zmult1-aux2*:
[| *b* \$< *#0*; *intify*(*c*) ≠ *#0* |] ==> (*c*\$**a*) *zdiv* (*c*\$**b*) = *a zdiv b*
apply (*subgoal-tac* (*c* \$* (\$−*a*)) *zdiv* (*c* \$* (\$−*b*)) = (\$−*a*) *zdiv* (\$−*b*))
apply (*rule-tac* [*2*] *zdiv-zmult-zmult1-aux1*)
apply *auto*
done

lemma *zdiv-zmult-zmult1-raw*:
[|*intify*(*c*) ≠ *#0*; *b* ∈ *int*|] ==> (*c*\$**a*) *zdiv* (*c*\$**b*) = *a zdiv b*
apply (*case-tac b* = *#0*)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*auto simp add*: *neq-iff-zless* [*of b*]
zdiv-zmult-zmult1-aux1 zdiv-zmult-zmult1-aux2)
done

lemma *zdiv-zmult-zmult1*: *intify*(*c*) ≠ *#0* ==> (*c*\$**a*) *zdiv* (*c*\$**b*) = *a zdiv b*
apply (*cut-tac b* = *intify* (*b*) **in** *zdiv-zmult-zmult1-raw*)
apply *auto*
done

lemma *zdiv-zmult-zmult2*: *intify*(*c*) ≠ *#0* ==> (*a*\$**c*) *zdiv* (*b*\$**c*) = *a zdiv b*
apply (*drule zdiv-zmult-zmult1*)
apply (*auto simp add*: *zmult-commute*)
done

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)
apply (*subst zmod-zmult2-eq*)
apply *auto*
done

lemma *zmod-zmult-zmult1-aux2*:
[| b $< #0; intify(c) ≠ #0 |]
==> (c$*a) zmod (c$*b) = c $* (a zmod b)
apply (*subgoal-tac* (c $* ($−a)) zmod (c $* ($−b)) = c $* (($−a) zmod ($−b)))
apply (*rule-tac* [2] *zmod-zmult-zmult1-aux1*)
apply *auto*
done

lemma *zmod-zmult-zmult1-raw*:
[|b ∈ int; c ∈ int|] ==> (c$*a) zmod (c$*b) = c $* (a zmod b)
apply (*case-tac* b = #0)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*case-tac* c = #0)
apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*auto simp add*: *neq-iff-zless* [*of b*]
zmod-zmult-zmult1-aux1 zmod-zmult-zmult1-aux2)
done

lemma *zmod-zmult-zmult1*: (c$*a) zmod (c$*b) = c $* (a zmod b)
apply (*cut-tac* b = *intify* (b) **and** c = *intify* (c) **in** *zmod-zmult-zmult1-raw*)
apply *auto*
done

lemma *zmod-zmult-zmult2*: (a$*c) zmod (b$*c) = (a zmod b) $* c
apply (*cut-tac* c = c **in** *zmod-zmult-zmult1*)
apply (*auto simp add*: *zmult-commute*)
done

lemma *zdiv-neg-pos-less0*: [| a $< #0; #0 $< b |] ==> a zdiv b $< #0
apply (*subgoal-tac* a zdiv b $<= #−1)
apply (*erule zle-zless-trans*)
apply (*simp* (*no-asm*))
apply (*rule zle-trans*)
apply (*rule-tac* a′ = #−1 **in** *zdiv-mono1*)
apply (*rule zless-add1-iff-zle* [*THEN iffD1*])
apply (*simp* (*no-asm*))
apply (*auto simp add*: *zdiv-minus1*)
done

lemma *zdiv-nonneg-neg-le0*: [| #0 $<= a; b $< #0 |] ==> a zdiv b $<= #0
apply (*drule zdiv-mono1-neg*)

apply *auto*
done

lemma *pos-imp-zdiv-nonneg-iff*: *#0 $< b ==> (#0 $<= a zdiv b) <-> (#0 $<= a)*
apply *auto*
apply (*drule-tac* [*2*] *zdiv-mono1*)
apply (*auto simp add*: *neq-iff-zless*)
apply (*simp* (*no-asm-use*) *add*: *not-zless-iff-zle* [*THEN iff-sym*])
apply (*blast intro*: *zdiv-neg-pos-less0*)
done

lemma *neg-imp-zdiv-nonneg-iff*: *b $< #0 ==> (#0 $<= a zdiv b) <-> (a $<= #0)*
apply (*subst zdiv-zminus-zminus* [*symmetric*])
apply (*rule iff-trans*)
apply (*rule pos-imp-zdiv-nonneg-iff*)
apply *auto*
done

lemma *pos-imp-zdiv-neg-iff*: *#0 $< b ==> (a zdiv b $< #0) <-> (a $< #0)*
apply (*simp* (*no-asm-simp*) *add*: *not-zle-iff-zless* [*THEN iff-sym*])
apply (*erule pos-imp-zdiv-nonneg-iff*)
done

lemma *neg-imp-zdiv-neg-iff*: *b $< #0 ==> (a zdiv b $< #0) <-> (#0 $< a)*
apply (*simp* (*no-asm-simp*) *add*: *not-zle-iff-zless* [*THEN iff-sym*])
apply (*erule neg-imp-zdiv-nonneg-iff*)
done

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-rel(*K*) ==
rvimage(*K*∗*K*,
lam <*x*,*y*>:*K*∗*K*. <*x Un y*, *x*, *y*>,
rmult(*K*,*Memrel*(*K*), *K*∗*K*, *rmult*(*K*,*Memrel*(*K*), *K*,*Memrel*(*K*))))

definition
jump-cardinal :: *i*=>*i* **where**
— This def is more complex than Kunen's but it more easily proved to be a cardinal
jump-cardinal(*K*) ==
$\bigcup$*X*∈*Pow*(*K*). {*z*. *r*: *Pow*(*K*∗*K*), *well-ord*(*X*,*r*) & *z* = *ordertype*(*X*,*r*)}

definition
csucc :: *i*=>*i* **where**
— needed because *jump-cardinal*(*K*) might not be the successor of *K*
csucc(*K*) == *LEAST L*. *Card*(*L*) & *K*<*L*

notation (*xsymbols* **output**)
cadd (**infixl** ⊕ *65*) **and**
cmult (**infixl** ⊗ *70*)

notation (*HTML* **output**)
cadd (**infixl** ⊕ *65*) **and**
cmult (**infixl** ⊗ *70*)

lemma *Card-Union* [*simp*,*intro*,*TC*]: (*ALL x*:*A*. *Card*(*x*)) ==> *Card*(*Union*(*A*))
apply (*rule CardI*)
apply (*simp add*: *Card-is-Ord*)
apply (*clarify dest*!: *ltD*)
apply (*drule bspec*, *assumption*)
apply (*frule lt-Card-imp-lesspoll*, *blast intro*: *ltI Card-is-Ord*)
apply (*drule eqpoll-sym* [*THEN eqpoll-imp-lepoll*])
apply (*drule lesspoll-trans1*, *assumption*)
apply (*subgoal-tac B* $\lesssim$ $\bigcup$*A*)
apply (*drule lesspoll-trans1*, *assumption*, *blast*)
apply (*blast intro*: *subset-imp-lepoll*)
done

lemma *Card-UN*: (!!*x*. *x*:*A* ==> *Card*(*K*(*x*))) ==> *Card*($\bigcup$*x*∈*A*. *K*(*x*))

by (*blast intro*: *Card-Union*)

lemma *Card-OUN* [*simp*,*intro*,*TC*]:
  (!!*x*. *x*:*A* ==> *Card*(*K*(*x*))) ==> *Card*(⋃*x*<*A*. *K*(*x*))
by (*simp add*: *OUnion-def Card-0*)

lemma *n-lesspoll-nat*: *n* ∈ *nat* ==> *n* ≺ *nat*
apply (*unfold lesspoll-def*)
apply (*rule conjI*)
apply (*erule OrdmemD* [*THEN subset-imp-lepoll*], *rule Ord-nat*)
apply (*rule notI*)
apply (*erule eqpollE*)
apply (*rule succ-lepoll-natE*)
apply (*blast intro*: *nat-succI* [*THEN OrdmemD*, *THEN subset-imp-lepoll*]
  lepoll-trans, *assumption*)
done

lemma *in-Card-imp-lesspoll*: [| *Card*(*K*); *b* ∈ *K* |] ==> *b* ≺ *K*
apply (*unfold lesspoll-def*)
apply (*simp add*: *Card-iff-initial*)
apply (*fast intro*!: *le-imp-lepoll ltI leI*)
done

lemma *lesspoll-lemma*: [| ~ *A* ≺ *B*; *C* ≺ *B* |] ==> *A* − *C* ≠ *0*
apply (*unfold lesspoll-def*)
apply (*fast dest*!: *Diff-eq-0-iff* [*THEN iffD1*, *THEN subset-imp-lepoll*]
  intro!: *eqpollI elim*: *notE*
  elim!: *eqpollE lepoll-trans*)
done

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* ≈ *B*+*A*
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule-tac c* = *case*(*Inr*,*Inl*) **and** *d* = *case*(*Inr*,*Inl*) **in** *lam-bijective*)
apply *auto*
done

lemma *cadd-commute*: *i* |+| *j* = *j* |+| *i*
apply (*unfold cadd-def*)
apply (*rule sum-commute-eqpoll* [*THEN cardinal-cong*])

done

33.1.2 Cardinal addition is associative

lemma *sum-assoc-eqpoll*: $(A+B)+C \approx A+(B+C)$
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule sum-assoc-bij*)
done

lemma *well-ord-cadd-assoc*:
  [| *well-ord*(*i*,*ri*); *well-ord*(*j*,*rj*); *well-ord*(*k*,*rk*) |]
  ==> (*i* |+| *j*) |+| *k* = *i* |+| (*j* |+| *k*)
apply (*unfold cadd-def*)
apply (*rule cardinal-cong*)
apply (*rule eqpoll-trans*)
apply (*rule sum-eqpoll-cong* [*OF well-ord-cardinal-eqpoll eqpoll-refl*])
apply (*blast intro*: *well-ord-radd*)
apply (*rule sum-assoc-eqpoll* [*THEN eqpoll-trans*])
apply (*rule eqpoll-sym*)
apply (*rule sum-eqpoll-cong* [*OF eqpoll-refl well-ord-cardinal-eqpoll*])
apply (*blast intro*: *well-ord-radd*)
done

33.1.3 0 is the identity for addition

lemma *sum-0-eqpoll*: $0+A \approx A$
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule bij-0-sum*)
done

lemma *cadd-0* [*simp*]: *Card*(*K*) ==> *0* |+| *K* = *K*
apply (*unfold cadd-def*)
apply (*simp add*: *sum-0-eqpoll* [*THEN cardinal-cong*] *Card-cardinal-eq*)
done

33.1.4 Addition by another cardinal

lemma *sum-lepoll-self*: $A \lesssim A+B$
apply (*unfold lepoll-def inj-def*)
apply (*rule-tac x = lam x*:*A*. *Inl* (*x*) **in** *exI*)
apply *simp*
done

lemma *cadd-le-self*:
  [| *Card*(*K*); *Ord*(*L*) |] ==> *K le* (*K* |+| *L*)

apply (*unfold cadd-def*)
apply (*rule le-trans* [*OF Card-cardinal-le well-ord-lepoll-imp-Card-le*], *assumption*)
apply (*rule-tac* [*2*] *sum-lepoll-self*)
apply (*blast intro*: *well-ord-radd well-ord-Memrel Card-is-Ord*)
done

33.1.5 Monotonicity of addition

lemma *sum-lepoll-mono*:
[| $A \lesssim C$; $B \lesssim D$ |] ==> $A + B \lesssim C + D$
apply (*unfold lepoll-def*)
apply (*elim exE*)
apply (*rule-tac x = lam z:A+B. case* (%*w. Inl*(*f'w*), %*y. Inr*(*fa'y*), *z*) **in** *exI*)
apply (*rule-tac d = case* (%*w. Inl*(*converse*(*f*) '*w*), %*y. Inr*(*converse*(*fa*) ' *y*)) **in** *lam-injective*)
apply (*typecheck add*: *inj-is-fun*, *auto*)
done

lemma *cadd-le-mono*:
[| K' *le* K; L' *le* L |] ==> (K' |+| L') *le* (K |+| L)
apply (*unfold cadd-def*)
apply (*safe dest*!: *le-subset-iff* [*THEN iffD1*])
apply (*rule well-ord-lepoll-imp-Card-le*)
apply (*blast intro*: *well-ord-radd well-ord-Memrel*)
apply (*blast intro*: *sum-lepoll-mono subset-imp-lepoll*)
done

33.1.6 Addition of finite cardinals is "ordinary" addition

lemma *sum-succ-eqpoll*: *succ*(*A*)+*B* $\approx$ *succ*(*A*+*B*)
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule-tac c =* %*z. if z=Inl* (*A*) *then A+B else z*
and *d =* %*z. if z=A+B then Inl* (*A*) *else z* **in** *lam-bijective*)
apply *simp-all*
apply (*blast dest*: *sym* [*THEN eq-imp-not-mem*] *elim*: *mem-irrefl*)+
done

lemma *cadd-succ-lemma*:
[| *Ord*(*m*); *Ord*(*n*) |] ==> *succ*(*m*) |+| *n* = |*succ*(*m* |+| *n*)|
apply (*unfold cadd-def*)
apply (*rule sum-succ-eqpoll* [*THEN cardinal-cong*, *THEN trans*])
apply (*rule succ-eqpoll-cong* [*THEN cardinal-cong*])
apply (*rule well-ord-cardinal-eqpoll* [*THEN eqpoll-sym*])
apply (*blast intro*: *well-ord-radd well-ord-Memrel*)
done

lemma *nat-cadd-eq-add*: [| *m*: *nat*; *n*: *nat* |] ==> *m* |+| *n* = *m*#+*n*
apply (*induct-tac m*)
apply (*simp add*: *nat-into-Card* [*THEN cadd-0*])
apply (*simp add*: *cadd-succ-lemma nat-into-Card* [*THEN Card-cardinal-eq*])
done

33.2 Cardinal multiplication

33.2.1 Cardinal multiplication is commutative

lemma *prod-commute-eqpoll*: *A*∗*B* ≈ *B*∗*A*
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule-tac c* = %<*x*,*y*>.<*y*,*x*> **and** *d* = %<*x*,*y*>.<*y*,*x*> **in** *lam-bijective*, *auto*)
done

lemma *cmult-commute*: *i* |∗| *j* = *j* |∗| *i*
apply (*unfold cmult-def*)
apply (*rule prod-commute-eqpoll* [*THEN cardinal-cong*])
done

33.2.2 Cardinal multiplication is associative

lemma *prod-assoc-eqpoll*: (*A*∗*B*)∗*C* ≈ *A*∗(*B*∗*C*)
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule prod-assoc-bij*)
done

lemma *well-ord-cmult-assoc*:
[| *well-ord*(*i*,*ri*); *well-ord*(*j*,*rj*); *well-ord*(*k*,*rk*) |]
==> (*i* |∗| *j*) |∗| *k* = *i* |∗| (*j* |∗| *k*)
apply (*unfold cmult-def*)
apply (*rule cardinal-cong*)
apply (*rule eqpoll-trans*)
apply (*rule prod-eqpoll-cong* [*OF well-ord-cardinal-eqpoll eqpoll-refl*])
apply (*blast intro*: *well-ord-rmult*)
apply (*rule prod-assoc-eqpoll* [*THEN eqpoll-trans*])
apply (*rule eqpoll-sym*)
apply (*rule prod-eqpoll-cong* [*OF eqpoll-refl well-ord-cardinal-eqpoll*])
apply (*blast intro*: *well-ord-rmult*)
done

33.2.3 Cardinal multiplication distributes over addition

lemma *sum-prod-distrib-eqpoll*: (*A*+*B*)∗*C* ≈ (*A*∗*C*)+(*B*∗*C*)
apply (*unfold eqpoll-def*)
apply (*rule exI*)

apply (*rule sum-prod-distrib-bij*)
done

lemma *well-ord-cadd-cmult-distrib*:
  [| *well-ord*(*i*,*ri*); *well-ord*(*j*,*rj*); *well-ord*(*k*,*rk*) |]
   ==> (*i* |+| *j*) |∗| *k* = (*i* |∗| *k*) |+| (*j* |∗| *k*)
apply (*unfold cadd-def cmult-def*)
apply (*rule cardinal-cong*)
apply (*rule eqpoll-trans*)
apply (*rule prod-eqpoll-cong* [*OF well-ord-cardinal-eqpoll eqpoll-refl*])
apply (*blast intro*: *well-ord-radd*)
apply (*rule sum-prod-distrib-eqpoll* [*THEN eqpoll-trans*])
apply (*rule eqpoll-sym*)
apply (*rule sum-eqpoll-cong* [*OF well-ord-cardinal-eqpoll*
      well-ord-cardinal-eqpoll])
apply (*blast intro*: *well-ord-rmult*)+
done

33.2.4 Multiplication by 0 yields 0

lemma *prod-0-eqpoll*: *0*∗*A* ≈ *0*
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule lam-bijective*, *safe*)
done

lemma *cmult-0* [*simp*]: *0* |∗| *i* = *0*
by (*simp add*: *cmult-def prod-0-eqpoll* [*THEN cardinal-cong*])

33.2.5 1 is the identity for multiplication

lemma *prod-singleton-eqpoll*: {*x*}∗*A* ≈ *A*
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule singleton-prod-bij* [*THEN bij-converse-bij*])
done

lemma *cmult-1* [*simp*]: *Card*(*K*) ==> *1* |∗| *K* = *K*
apply (*unfold cmult-def succ-def*)
apply (*simp add*: *prod-singleton-eqpoll* [*THEN cardinal-cong*] *Card-cardinal-eq*)
done

33.3 Some inequalities for multiplication

lemma *prod-square-lepoll*: *A* ≲ *A*∗*A*
apply (*unfold lepoll-def inj-def*)
apply (*rule-tac x* = *lam x*:*A*. <*x*,*x*> **in** *exI*, *simp*)
done

```
lemma cmult-square-le: Card(K) ==> K le K |*| K
apply (unfold cmult-def)
apply (rule le-trans)
apply (rule-tac [2] well-ord-lepoll-imp-Card-le)
apply (rule-tac [3] prod-square-lepoll)
apply (simp add: le-refl Card-is-Ord Card-cardinal-eq)
apply (blast intro: well-ord-rmult well-ord-Memrel Card-is-Ord)
done
```

33.3.1 Multiplication by a non-zero cardinal

```
lemma prod-lepoll-self: b: B ==> A ≲ A*B
apply (unfold lepoll-def inj-def)
apply (rule-tac x = lam x:A. <x,b> in exI, simp)
done
```

```
lemma cmult-le-self:
   [| Card(K);  Ord(L);  0<L |] ==> K le (K |*| L)
apply (unfold cmult-def)
apply (rule le-trans [OF Card-cardinal-le well-ord-lepoll-imp-Card-le])
  apply assumption
 apply (blast intro: well-ord-rmult well-ord-Memrel Card-is-Ord)
apply (blast intro: prod-lepoll-self ltD)
done
```

33.3.2 Monotonicity of multiplication

```
lemma prod-lepoll-mono:
     [| A ≲ C;  B ≲ D |] ==> A * B  ≲  C * D
apply (unfold lepoll-def)
apply (elim exE)
apply (rule-tac x = lam <w,y>:A*B. <f`w, fa`y> in exI)
apply (rule-tac d = %<w,y>. <converse (f) `w, converse (fa) `y>
       in lam-injective)
apply (typecheck add: inj-is-fun, auto)
done
```

```
lemma cmult-le-mono:
    [| K' le K;  L' le L |] ==> (K' |*| L') le (K |*| L)
apply (unfold cmult-def)
apply (safe dest!: le-subset-iff [THEN iffD1])
apply (rule well-ord-lepoll-imp-Card-le)
 apply (blast intro: well-ord-rmult well-ord-Memrel)
apply (blast intro: prod-lepoll-mono subset-imp-lepoll)
done
```

33.4 Multiplication of finite cardinals is "ordinary" multiplication

lemma *prod-succ-eqpoll*: *succ*(*A*)**B* ≈ *B* + *A***B*
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*rule-tac c* = %<*x*,*y*>. *if x*=*A then Inl* (*y*) *else Inr* (<*x*,*y*>)
          and *d* = *case* (%*y*. <*A*,*y*>, %*z*. *z*) **in** *lam-bijective*)
apply *safe*
apply (*simp-all add*: *succI2 if-type mem-imp-not-eq*)
done

lemma *cmult-succ-lemma*:
  [| *Ord*(*m*); *Ord*(*n*) |] ==> *succ*(*m*) |*| *n* = *n* |+| (*m* |*| *n*)
apply (*unfold cmult-def cadd-def*)
apply (*rule prod-succ-eqpoll* [*THEN cardinal-cong*, *THEN trans*])
apply (*rule cardinal-cong* [*symmetric*])
apply (*rule sum-eqpoll-cong* [*OF eqpoll-refl well-ord-cardinal-eqpoll*])
apply (*blast intro*: *well-ord-rmult well-ord-Memrel*)
done

lemma *nat-cmult-eq-mult*: [| *m*: *nat*; *n*: *nat* |] ==> *m* |*| *n* = *m*#**n*
apply (*induct-tac m*)
apply (*simp-all add*: *cmult-succ-lemma nat-cadd-eq-add*)
done

lemma *cmult-2*: *Card*(*n*) ==> *2* |*| *n* = *n* |+| *n*
by (*simp add*: *cmult-succ-lemma Card-is-Ord cadd-commute* [*of* - *0*])

lemma *sum-lepoll-prod*: *2* ≲ *C* ==> *B*+*B* ≲ *C***B*
apply (*rule lepoll-trans*)
apply (*rule sum-eq-2-times* [*THEN equalityD1*, *THEN subset-imp-lepoll*])
apply (*erule prod-lepoll-mono*)
apply (*rule lepoll-refl*)
done

lemma *lepoll-imp-sum-lepoll-prod*: [| *A* ≲ *B*; *2* ≲ *A* |] ==> *A*+*B* ≲ *A***B*
by (*blast intro*: *sum-lepoll-mono sum-lepoll-prod lepoll-trans lepoll-refl*)

33.5 Infinite Cardinals are Limit Ordinals

lemma *nat-cons-lepoll*: *nat* ≲ *A* ==> *cons*(*u*,*A*) ≲ *A*
apply (*unfold lepoll-def*)
apply (*erule exE*)
apply (*rule-tac x* =
        lam z:*cons* (*u*,*A*).
          if z=*u then f*‘*0*
          else if z: *range* (*f*) *then f*‘*succ* (*converse* (*f*) ‘*z*) *else z*
    in *exI*)

apply (*rule-tac d* =
    %y. if y: range(f) then nat-case (u, %z. f'z, converse(f) 'y)
        else y
  in *lam-injective*)
apply (*fast intro!: if-type apply-type intro: inj-is-fun inj-converse-fun*)
apply (*simp add: inj-is-fun [THEN apply-rangeI]*
    inj-converse-fun [THEN apply-rangeI]
    inj-converse-fun [THEN apply-funtype])
done

lemma *nat-cons-eqpoll*: $nat \lesssim A$ ==> $cons(u,A) \approx A$
apply (*erule nat-cons-lepoll [THEN eqpollI]*)
apply (*rule subset-consI [THEN subset-imp-lepoll]*)
done

lemma *nat-succ-eqpoll*: *nat* <= *A* ==> $succ(A) \approx A$
apply (*unfold succ-def*)
apply (*erule subset-imp-lepoll [THEN nat-cons-eqpoll]*)
done

lemma *InfCard-nat*: *InfCard(nat)*
apply (*unfold InfCard-def*)
apply (*blast intro: Card-nat le-refl Card-is-Ord*)
done

lemma *InfCard-is-Card*: *InfCard(K)* ==> *Card(K)*
apply (*unfold InfCard-def*)
apply (*erule conjunct1*)
done

lemma *InfCard-Un*:
  [| *InfCard(K)*; *Card(L)* |] ==> *InfCard(K Un L)*
apply (*unfold InfCard-def*)
apply (*simp add: Card-Un Un-upper1-le [THEN [2] le-trans] Card-is-Ord*)
done

lemma *InfCard-is-Limit*: *InfCard(K)* ==> *Limit(K)*
apply (*unfold InfCard-def*)
apply (*erule conjE*)
apply (*frule Card-is-Ord*)
apply (*rule ltI [THEN non-succ-LimitI]*)
apply (*erule le-imp-subset [THEN subsetD]*)
apply (*safe dest!: Limit-nat [THEN Limit-le-succD]*)
apply (*unfold Card-def*)
apply (*drule trans*)
apply (*erule le-imp-subset [THEN nat-succ-eqpoll, THEN cardinal-cong]*)
apply (*erule Ord-cardinal-le [THEN lt-trans2, THEN lt-irrefl]*)

apply (*rule le-eqI*, *assumption*)
apply (*rule Ord-cardinal*)
done

lemma *ordermap-eqpoll-pred*:
[| *well-ord*(*A*,*r*); *x*:*A* |] ==> *ordermap*(*A*,*r*)'*x* ≈ *Order.pred*(*A*,*x*,*r*)
apply (*unfold eqpoll-def*)
apply (*rule exI*)
apply (*simp add*: *ordermap-eq-image well-ord-is-wf*)
apply (*erule ordermap-bij* [*THEN bij-is-inj*, *THEN restrict-bij*,
THEN bij-converse-bij])
apply (*rule pred-subset*)
done

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*)
apply (*unfold inj-def*)
apply (*force intro*: *lam-type Un-least-lt* [*THEN ltD*] *ltI*)
done

lemma *well-ord-csquare*: *Ord*(*K*) ==> *well-ord*(*K***K*, *csquare-rel*(*K*))
apply (*unfold csquare-rel-def*)
apply (*rule csquare-lam-inj* [*THEN well-ord-rvimage*], *assumption*)
apply (*blast intro*: *well-ord-rmult well-ord-Memrel*)
done

33.5.2 Characterising initial segments of the well-ordering

lemma *csquareD*:
[| <<*x*,*y*>, <*z*,*z*>> : *csquare-rel*(*K*); *x*<*K*; *y*<*K*; *z*<*K* |] ==> *x le z* & *y le z*
apply (*unfold csquare-rel-def*)
apply (*erule rev-mp*)
apply (*elim ltE*)
apply (*simp add*: *rvimage-iff Un-absorb Un-least-mem-iff ltD*)
apply (*safe elim*!: *mem-irrefl intro*!: *Un-upper1-le Un-upper2-le*)
apply (*simp-all add*: *lt-def succI2*)
done

lemma *pred-csquare-subset*:
z<*K* ==> *Order.pred*(*K***K*, <*z*,*z*>, *csquare-rel*(*K*)) <= *succ*(*z*)**succ*(*z*)
apply (*unfold Order.pred-def*)
apply (*safe del*: *SigmaI succCI*)
apply (*erule csquareD* [*THEN conjE*])

apply (*unfold lt-def*, *auto*)
done

lemma *csquare-ltI*:
[| *x*<*z*; *y*<*z*; *z*<*K* |] ==> <<*x*,*y*>, <*z*,*z*>> : *csquare-rel*(*K*)
apply (*unfold csquare-rel-def*)
apply (*subgoal-tac x*<*K* & *y*<*K*)
prefer *2* **apply** (*blast intro*: *lt-trans*)
apply (*elim ltE*)
apply (*simp add*: *rvimage-iff Un-absorb Un-least-mem-iff ltD*)
done

lemma *csquare-or-eqI*:
[| *x le z*; *y le z*; *z*<*K* |] ==> <<*x*,*y*>, <*z*,*z*>> : *csquare-rel*(*K*) | *x*=*z* & *y*=*z*
apply (*unfold csquare-rel-def*)
apply (*subgoal-tac x*<*K* & *y*<*K*)
prefer *2* **apply** (*blast intro*: *lt-trans1*)
apply (*elim ltE*)
apply (*simp add*: *rvimage-iff Un-absorb Un-least-mem-iff ltD*)
apply (*elim succE*)
apply (*simp-all add*: *subset-Un-iff* [*THEN iff-sym*]
subset-Un-iff2 [*THEN iff-sym*] *OrdmemD*)
done

33.5.3 The cardinality of initial segments

lemma *ordermap-z-lt*:
[| *Limit*(*K*); *x*<*K*; *y*<*K*; *z*=*succ*(*x Un y*) |] ==>
ordermap(*K*∗*K*, *csquare-rel*(*K*)) ‘ <*x*,*y*> <
ordermap(*K*∗*K*, *csquare-rel*(*K*)) ‘ <*z*,*z*>
apply (*subgoal-tac z*<*K* & *well-ord* (*K*∗*K*, *csquare-rel* (*K*)))
prefer *2* **apply** (*blast intro*!: *Un-least-lt Limit-has-succ*
Limit-is-Ord [*THEN well-ord-csquare*], *clarify*)
apply (*rule csquare-ltI* [*THEN ordermap-mono*, *THEN ltI*])
apply (*erule-tac* [*4*] *well-ord-is-wf*)
apply (*blast intro*!: *Un-upper1-le Un-upper2-le Ord-ordermap elim*!: *ltE*)+
done

lemma *ordermap-csquare-le*:
[| *Limit*(*K*); *x*<*K*; *y*<*K*; *z*=*succ*(*x Un y*) |]
==> | *ordermap*(*K*∗*K*, *csquare-rel*(*K*)) ‘ <*x*,*y*> | *le* |*succ*(*z*)| |∗| |*succ*(*z*)|
apply (*unfold cmult-def*)
apply (*rule well-ord-rmult* [*THEN well-ord-lepoll-imp-Card-le*])
apply (*rule Ord-cardinal* [*THEN well-ord-Memrel*])+
apply (*subgoal-tac z*<*K*)
prefer *2* **apply** (*blast intro*!: *Un-least-lt Limit-has-succ*)
apply (*rule ordermap-z-lt* [*THEN leI*, *THEN le-imp-lepoll*, *THEN lepoll-trans*],

assumption+)
apply (*rule ordermap-eqpoll-pred* [*THEN eqpoll-imp-lepoll*, *THEN lepoll-trans*])
apply (*erule Limit-is-Ord* [*THEN well-ord-csquare*])
apply (*blast intro*: *ltD*)
apply (*rule pred-csquare-subset* [*THEN subset-imp-lepoll*, *THEN lepoll-trans*], *assumption*)
apply (*elim ltE*)
apply (*rule prod-eqpoll-cong* [*THEN eqpoll-sym*, *THEN eqpoll-imp-lepoll*])
apply (*erule Ord-succ* [*THEN Ord-cardinal-eqpoll*])+
done

lemma *ordertype-csquare-le*:
[| *InfCard*(*K*); *ALL y*:*K*. *InfCard*(*y*) --> *y* |*| *y* = *y* |]
==> *ordertype*(*K***K*, *csquare-rel*(*K*)) *le K*
apply (*frule InfCard-is-Card* [*THEN Card-is-Ord*])
apply (*rule all-lt-imp-le*, *assumption*)
apply (*erule well-ord-csquare* [*THEN Ord-ordertype*])
apply (*rule Card-lt-imp-lt*)
apply (*erule-tac* [*3*] *InfCard-is-Card*)
apply (*erule-tac* [*2*] *ltE*)
apply (*simp add*: *ordertype-unfold*)
apply (*safe elim*!: *ltE*)
apply (*subgoal-tac Ord* (*xa*) & *Ord* (*ya*))
prefer *2* **apply** (*blast intro*: *Ord-in-Ord*, *clarify*)

apply (*rule InfCard-is-Limit* [*THEN ordermap-csquare-le*, *THEN lt-trans1*], (*assumption* | *rule refl* | *erule ltI*)+)
apply (*rule-tac i* = *xa Un ya* **and** *j* = *nat* **in** *Ord-linear2*, *simp-all add*: *Ord-Un Ord-nat*)
prefer *2*
apply (*simp add*: *le-imp-subset* [*THEN nat-succ-eqpoll*, *THEN cardinal-cong*] *le-succ-iff InfCard-def Card-cardinal Un-least-lt Ord-Un ltI nat-le-cardinal Ord-cardinal-le* [*THEN lt-trans1*, *THEN ltD*])

apply (*rule-tac j* = *nat* **in** *lt-trans2*)
apply (*simp add*: *lt-def nat-cmult-eq-mult nat-succI mult-type nat-into-Card* [*THEN Card-cardinal-eq*] *Ord-nat*)
apply (*simp add*: *InfCard-def*)
done

lemma *InfCard-csquare-eq*: *InfCard*(*K*) ==> *K* |*| *K* = *K*
apply (*frule InfCard-is-Card* [*THEN Card-is-Ord*])
apply (*erule rev-mp*)
apply (*erule-tac i*=*K* **in** *trans-induct*)
apply (*rule impI*)
apply (*rule le-anti-sym*)
apply (*erule-tac* [*2*] *InfCard-is-Card* [*THEN cmult-square-le*])

apply (*rule ordertype-csquare-le* [*THEN* [*2*] *le-trans*])
apply (*simp add*: *cmult-def Ord-cardinal-le*
well-ord-csquare [*THEN Ord-ordertype*]
well-ord-csquare [*THEN ordermap-bij*, *THEN bij-imp-eqpoll*,
THEN cardinal-cong], *assumption*+)
done

lemma *well-ord-InfCard-square-eq*:
[| *well-ord*(*A*,*r*); *InfCard*(|*A*|) |] ==> *A*∗*A* ≈ *A*
apply (*rule prod-eqpoll-cong* [*THEN eqpoll-trans*])
apply (*erule well-ord-cardinal-eqpoll* [*THEN eqpoll-sym*])+
apply (*rule well-ord-cardinal-eqE*)
apply (*blast intro*: *Ord-cardinal well-ord-rmult well-ord-Memrel*, *assumption*)
apply (*simp add*: *cmult-def* [*symmetric*] *InfCard-csquare-eq*)
done

lemma *InfCard-square-eqpoll*: *InfCard*(*K*) ==> *K* × *K* ≈ *K*
apply (*rule well-ord-InfCard-square-eq*)
apply (*erule InfCard-is-Card* [*THEN Card-is-Ord*, *THEN well-ord-Memrel*])
apply (*simp add*: *InfCard-is-Card* [*THEN Card-cardinal-eq*])
done

lemma *Inf-Card-is-InfCard*: [| ~*Finite*(*i*); *Card*(*i*) |] ==> *InfCard*(*i*)
by (*simp add*: *InfCard-def Card-is-Ord* [*THEN nat-le-infinite-Ord*])

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*
apply (*rule le-anti-sym*)
prefer *2*
apply (*erule ltE*, *blast intro*: *cmult-le-self InfCard-is-Card*)
apply (*frule InfCard-is-Card* [*THEN Card-is-Ord*, *THEN le-refl*])
apply (*rule cmult-le-mono* [*THEN le-trans*], *assumption*+)
apply (*simp add*: *InfCard-csquare-eq*)
done

lemma *InfCard-cmult-eq*: [| *InfCard*(*K*); *InfCard*(*L*) |] ==> *K* |∗| *L* = *K Un L*
apply (*rule-tac i* = *K* **and** *j* = *L* **in** *Ord-linear-le*)
apply (*typecheck add*: *InfCard-is-Card Card-is-Ord*)
apply (*rule cmult-commute* [*THEN ssubst*])
apply (*rule Un-commute* [*THEN ssubst*])
apply (*simp-all add*: *InfCard-is-Limit* [*THEN Limit-has-0*] *InfCard-le-cmult-eq*
subset-Un-iff2 [*THEN iffD1*] *le-imp-subset*)
done

lemma *InfCard-cdouble-eq*: *InfCard*(*K*) ==> *K* |+| *K* = *K*
apply (*simp add*: *cmult-2* [*symmetric*] *InfCard-is-Card cmult-commute*)

apply (*simp add*: *InfCard-le-cmult-eq InfCard-is-Limit Limit-has-0 Limit-has-succ*)
done

lemma *InfCard-le-cadd-eq*: [| *InfCard*(*K*); *L le K* |] ==> *K* |+| *L* = *K*
apply (*rule le-anti-sym*)
 prefer *2*
 apply (*erule ltE*, *blast intro*: *cadd-le-self InfCard-is-Card*)
apply (*frule InfCard-is-Card* [*THEN Card-is-Ord*, *THEN le-refl*])
apply (*rule cadd-le-mono* [*THEN le-trans*], *assumption*+)
apply (*simp add*: *InfCard-cdouble-eq*)
done

lemma *InfCard-cadd-eq*: [| *InfCard*(*K*); *InfCard*(*L*) |] ==> *K* |+| *L* = *K Un L*
apply (*rule-tac i* = *K* **and** *j* = *L* **in** *Ord-linear-le*)
apply (*typecheck add*: *InfCard-is-Card Card-is-Ord*)
apply (*rule cadd-commute* [*THEN ssubst*])
apply (*rule Un-commute* [*THEN ssubst*])
apply (*simp-all add*: *InfCard-le-cadd-eq subset-Un-iff2* [*THEN iffD1*] *le-imp-subset*)
done

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*))
apply (*unfold jump-cardinal-def*)
apply (*rule Ord-is-Transset* [*THEN* [*2*] *OrdI*])
 prefer *2* **apply** (*blast intro*!: *Ord-ordertype*)
apply (*unfold Transset-def*)
apply (*safe del*: *subsetI*)
apply (*simp add*: *ordertype-pred-unfold*, *safe*)
apply (*rule UN-I*)
apply (*rule-tac* [*2*] *ReplaceI*)
   prefer *4* **apply** (*blast intro*: *well-ord-subset elim*!: *predE*)+
done

lemma *jump-cardinal-iff*:
     i : *jump-cardinal*(*K*) <->
      (*EX r X*. *r* <= *K***K* & *X* <= *K* & *well-ord*(*X*,*r*) & *i* = *ordertype*(*X*,*r*))
apply (*unfold jump-cardinal-def*)
apply (*blast del*: *subsetI*)
done

lemma *K-lt-jump-cardinal*: *Ord*(*K*) ==> *K* < *jump-cardinal*(*K*)
apply (*rule Ord-jump-cardinal* [*THEN* [*2*] *ltI*])
apply (*rule jump-cardinal-iff* [*THEN iffD2*])

apply (*rule-tac x=Memrel*(*K*) **in** *exI*)
apply (*rule-tac x=K* **in** *exI*)
apply (*simp add*: *ordertype-Memrel well-ord-Memrel*)
apply (*simp add*: *Memrel-def subset-iff*)
done

lemma *Card-jump-cardinal-lemma*:
[| *well-ord*(*X*,*r*); *r* <= *K* ∗ *K*; *X* <= *K*;
f : *bij*(*ordertype*(*X*,*r*), *jump-cardinal*(*K*)) |]
==> *jump-cardinal*(*K*) : *jump-cardinal*(*K*)
apply (*subgoal-tac f O ordermap* (*X*,*r*) : *bij* (*X*, *jump-cardinal* (*K*)))
prefer *2* **apply** (*blast intro*: *comp-bij ordermap-bij*)
apply (*rule jump-cardinal-iff* [*THEN iffD2*])
apply (*intro exI conjI*)
apply (*rule subset-trans* [*OF rvimage-type Sigma-mono*], *assumption*+)
apply (*erule bij-is-inj* [*THEN well-ord-rvimage*])
apply (*rule Ord-jump-cardinal* [*THEN well-ord-Memrel*])
apply (*simp add*: *well-ord-Memrel* [*THEN* [*2*] *bij-ordertype-vimage*]
ordertype-Memrel Ord-jump-cardinal)
done

lemma *Card-jump-cardinal*: *Card*(*jump-cardinal*(*K*))
apply (*rule Ord-jump-cardinal* [*THEN CardI*])
apply (*unfold eqpoll-def*)
apply (*safe dest*!: *ltD jump-cardinal-iff* [*THEN iffD1*])
apply (*blast intro*: *Card-jump-cardinal-lemma* [*THEN mem-irrefl*])
done

33.7 Basic Properties of Successor Cardinals

lemma *csucc-basic*: *Ord*(*K*) ==> *Card*(*csucc*(*K*)) & *K* < *csucc*(*K*)
apply (*unfold csucc-def*)
apply (*rule LeastI*)
apply (*blast intro*: *Card-jump-cardinal K-lt-jump-cardinal Ord-jump-cardinal*)+
done

lemmas *Card-csucc* = *csucc-basic* [*THEN conjunct1*, *standard*]

lemmas *lt-csucc* = *csucc-basic* [*THEN conjunct2*, *standard*]

lemma *Ord-0-lt-csucc*: *Ord*(*K*) ==> *0* < *csucc*(*K*)
by (*blast intro*: *Ord-0-le lt-csucc lt-trans1*)

lemma *csucc-le*: [| *Card*(*L*); *K*<*L* |] ==> *csucc*(*K*) *le L*
apply (*unfold csucc-def*)
apply (*rule Least-le*)
apply (*blast intro*: *Card-is-Ord*)+

done

lemma *lt-csucc-iff*: [| *Ord*(*i*); *Card*(*K*) |] ==> *i* < *csucc*(*K*) <-> |*i*| *le* *K*
apply (*rule iffI*)
apply (*rule-tac* [*2*] *Card-lt-imp-lt*)
apply (*erule-tac* [*2*] *lt-trans1*)
apply (*simp-all add*: *lt-csucc Card-csucc Card-is-Ord*)
apply (*rule notI* [*THEN not-lt-imp-le*])
apply (*rule Card-cardinal* [*THEN csucc-le*, *THEN lt-trans1*, *THEN lt-irrefl*], *assumption*)
apply (*rule Ord-cardinal-le* [*THEN lt-trans1*])
apply (*simp-all add*: *Ord-cardinal Card-is-Ord*)
done

lemma *Card-lt-csucc-iff*:
[| *Card*(*K'*); *Card*(*K*) |] ==> *K'* < *csucc*(*K*) <-> *K'* *le* *K*
by (*simp add*: *lt-csucc-iff Card-cardinal-eq Card-is-Ord*)

lemma *InfCard-csucc*: *InfCard*(*K*) ==> *InfCard*(*csucc*(*K*))
by (*simp add*: *InfCard-def Card-csucc Card-is-Ord*
lt-csucc [*THEN leI*, *THEN* [*2*] *le-trans*])

33.7.1 Removing elements from a finite set decreases its cardinality

lemma *Fin-imp-not-cons-lepoll*: *A*: *Fin*(*U*) ==> *x*~:*A* --> ~ *cons*(*x*,*A*) $\lesssim$ *A*
apply (*erule Fin-induct*)
apply (*simp add*: *lepoll-0-iff*)
apply (*subgoal-tac cons* (*x*,*cons* (*xa*,*y*)) = *cons* (*xa*,*cons* (*x*,*y*)))
apply *simp*
apply (*blast dest*!: *cons-lepoll-consD*, *blast*)
done

lemma *Finite-imp-cardinal-cons* [*simp*]:
[| *Finite*(*A*); *a*~:*A* |] ==> |*cons*(*a*,*A*)| = *succ*(|*A*|)
apply (*unfold cardinal-def*)
apply (*rule Least-equality*)
apply (*fold cardinal-def*)
apply (*simp add*: *succ-def*)
apply (*blast intro*: *cons-eqpoll-cong well-ord-cardinal-eqpoll*
elim!: *mem-irrefl* *dest*!: *Finite-imp-well-ord*)
apply (*blast intro*: *Card-cardinal Card-is-Ord*)
apply (*rule notI*)
apply (*rule Finite-into-Fin* [*THEN Fin-imp-not-cons-lepoll*, *THEN mp*, *THEN notE*],
assumption, *assumption*)
apply (*erule eqpoll-sym* [*THEN eqpoll-imp-lepoll*, *THEN lepoll-trans*])
apply (*erule le-imp-lepoll* [*THEN lepoll-trans*])
apply (*blast intro*: *well-ord-cardinal-eqpoll* [*THEN eqpoll-imp-lepoll*]

dest!: *Finite-imp-well-ord*)
done

lemma *Finite-imp-succ-cardinal-Diff*:
[| *Finite*(*A*); *a*:*A* |] ==> *succ*(|*A*−{*a*}|) = |*A*|
apply (*rule-tac b* = *A* **in** *cons-Diff* [*THEN subst*], *assumption*)
apply (*simp add*: *Finite-imp-cardinal-cons Diff-subset* [*THEN subset-Finite*])
apply (*simp add*: *cons-Diff*)
done

lemma *Finite-imp-cardinal-Diff*: [| *Finite*(*A*); *a*:*A* |] ==> |*A*−{*a*}| < |*A*|
apply (*rule succ-leE*)
apply (*simp add*: *Finite-imp-succ-cardinal-Diff*)
done

lemma *Finite-cardinal-in-nat* [*simp*]: *Finite*(*A*) ==> |*A*| : *nat*
apply (*erule Finite-induct*)
apply (*auto simp add*: *cardinal-0 Finite-imp-cardinal-cons*)
done

lemma *card-Un-Int*:
[|*Finite*(*A*); *Finite*(*B*)|] ==> |*A*| #+ |*B*| = |*A Un B*| #+ |*A Int B*|
apply (*erule Finite-induct*, *simp*)
apply (*simp add*: *Finite-Int cons-absorb Un-cons Int-cons-left*)
done

lemma *card-Un-disjoint*:
[|*Finite*(*A*); *Finite*(*B*); *A Int B* = *0*|] ==> |*A Un B*| = |*A*| #+ |*B*|
by (*simp add*: *Finite-Un card-Un-Int*)

lemma *card-partition* [*rule-format*]:
Finite(*C*) ==>
Finite ($\bigcup$ *C*) −−>
($\forall$ *c*$\in$*C*. |*c*| = *k*) −−>
($\forall$ *c1* $\in$ *C*. $\forall$ *c2* $\in$ *C*. *c1* $\neq$ *c2* −−> *c1* $\cap$ *c2* = *0*) −−>
k #∗ |*C*| = |$\bigcup$ *C*|
apply (*erule Finite-induct*, *auto*)
apply (*subgoal-tac x* $\cap \bigcup B$ = *0*)
apply (*auto simp add*: *card-Un-disjoint Finite-Union*
subset-Finite [*of* - $\bigcup$ (*cons*(*x*,*F*))])
done

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* $\approx$ *m* #+ *n*
apply (*rule eqpoll-trans*)

```
apply (rule well-ord-radd [THEN well-ord-cardinal-eqpoll, THEN eqpoll-sym])
apply (erule nat-implies-well-ord)+
apply (simp add: nat-cadd-eq-add [symmetric] cadd-def eqpoll-refl)
done

lemma Ord-subset-natD [rule-format]: Ord(i) ==> i <= nat --> i : nat | i=nat
apply (erule trans-induct3, auto)
apply (blast dest!: nat-le-Limit [THEN le-imp-subset])
done

lemma Ord-nat-subset-into-Card: [| Ord(i); i <= nat |] ==> Card(i)
by (blast dest: Ord-subset-natD intro: Card-nat nat-into-Card)

lemma Finite-Diff-sing-eq-diff-1: [| Finite(A); x:A |] ==> |A−{x}| = |A| #− 1
apply (rule succ-inject)
apply (rule-tac b = |A| in trans)
 apply (simp add: Finite-imp-succ-cardinal-Diff)
apply (subgoal-tac 1 ≲ A)
 prefer 2 apply (blast intro: not-0-is-lepoll-1)
apply (frule Finite-imp-well-ord, clarify)
apply (drule well-ord-lepoll-imp-Card-le)
 apply (auto simp add: cardinal-1)
apply (rule trans)
 apply (rule-tac [2] diff-succ)
  apply (auto simp add: Finite-cardinal-in-nat)
done

lemma cardinal-lt-imp-Diff-not-0 [rule-format]:
     Finite(B) ==> ALL A. |B|<|A| --> A − B ~= 0
apply (erule Finite-induct, auto)
apply (case-tac Finite (A))
 apply (subgoal-tac [2] Finite (cons (x, B)))
  apply (drule-tac [2] B = cons (x, B)  in Diff-Finite)
   apply (auto simp add: Finite-0 Finite-cons)
apply (subgoal-tac |B|<|A|)
 prefer 2 apply (blast intro: lt-trans Ord-cardinal)
apply (case-tac x:A)
 apply (subgoal-tac [2] A − cons (x, B) = A − B)
  apply auto
apply (subgoal-tac |A| le |cons (x, B) |)
 prefer 2
 apply (blast dest: Finite-cons [THEN Finite-imp-well-ord]
             intro: well-ord-lepoll-imp-Card-le subset-imp-lepoll)
apply (auto simp add: Finite-imp-cardinal-cons)
apply (auto dest!: Finite-cardinal-in-nat simp add: le-iff)
apply (blast intro: lt-trans)
done
```

ML⟪
val InfCard-def = thm InfCard-def
val cmult-def = thm cmult-def
val cadd-def = thm cadd-def
val jump-cardinal-def = thm jump-cardinal-def
val csucc-def = thm csucc-def

val sum-commute-eqpoll = thm sum-commute-eqpoll;
val cadd-commute = thm cadd-commute;
val sum-assoc-eqpoll = thm sum-assoc-eqpoll;
val well-ord-cadd-assoc = thm well-ord-cadd-assoc;
val sum-0-eqpoll = thm sum-0-eqpoll;
val cadd-0 = thm cadd-0;
val sum-lepoll-self = thm sum-lepoll-self;
val cadd-le-self = thm cadd-le-self;
val sum-lepoll-mono = thm sum-lepoll-mono;
val cadd-le-mono = thm cadd-le-mono;
val eq-imp-not-mem = thm eq-imp-not-mem;
val sum-succ-eqpoll = thm sum-succ-eqpoll;
val nat-cadd-eq-add = thm nat-cadd-eq-add;
val prod-commute-eqpoll = thm prod-commute-eqpoll;
val cmult-commute = thm cmult-commute;
val prod-assoc-eqpoll = thm prod-assoc-eqpoll;
val well-ord-cmult-assoc = thm well-ord-cmult-assoc;
val sum-prod-distrib-eqpoll = thm sum-prod-distrib-eqpoll;
val well-ord-cadd-cmult-distrib = thm well-ord-cadd-cmult-distrib;
val prod-0-eqpoll = thm prod-0-eqpoll;
val cmult-0 = thm cmult-0;
val prod-singleton-eqpoll = thm prod-singleton-eqpoll;
val cmult-1 = thm cmult-1;
val prod-lepoll-self = thm prod-lepoll-self;
val cmult-le-self = thm cmult-le-self;
val prod-lepoll-mono = thm prod-lepoll-mono;
val cmult-le-mono = thm cmult-le-mono;
val prod-succ-eqpoll = thm prod-succ-eqpoll;
val nat-cmult-eq-mult = thm nat-cmult-eq-mult;
val cmult-2 = thm cmult-2;
val sum-lepoll-prod = thm sum-lepoll-prod;
val lepoll-imp-sum-lepoll-prod = thm lepoll-imp-sum-lepoll-prod;
val nat-cons-lepoll = thm nat-cons-lepoll;
val nat-cons-eqpoll = thm nat-cons-eqpoll;
val nat-succ-eqpoll = thm nat-succ-eqpoll;
val InfCard-nat = thm InfCard-nat;
val InfCard-is-Card = thm InfCard-is-Card;
val InfCard-Un = thm InfCard-Un;
val InfCard-is-Limit = thm InfCard-is-Limit;
val ordermap-eqpoll-pred = thm ordermap-eqpoll-pred;
val ordermap-z-lt = thm ordermap-z-lt;
val InfCard-le-cmult-eq = thm InfCard-le-cmult-eq;

val InfCard-cmult-eq = thm InfCard-cmult-eq;
val InfCard-cdouble-eq = thm InfCard-cdouble-eq;
val InfCard-le-cadd-eq = thm InfCard-le-cadd-eq;
val InfCard-cadd-eq = thm InfCard-cadd-eq;
val Ord-jump-cardinal = thm Ord-jump-cardinal;
val jump-cardinal-iff = thm jump-cardinal-iff;
val K-lt-jump-cardinal = thm K-lt-jump-cardinal;
val Card-jump-cardinal = thm Card-jump-cardinal;
val csucc-basic = thm csucc-basic;
val Card-csucc = thm Card-csucc;
val lt-csucc = thm lt-csucc;
val Ord-0-lt-csucc = thm Ord-0-lt-csucc;
val csucc-le = thm csucc-le;
val lt-csucc-iff = thm lt-csucc-iff;
val Card-lt-csucc-iff = thm Card-lt-csucc-iff;
val InfCard-csucc = thm InfCard-csucc;
val Finite-into-Fin = thm Finite-into-Fin;
val Fin-into-Finite = thm Fin-into-Finite;
val Finite-Fin-iff = thm Finite-Fin-iff;
val Finite-Un = thm Finite-Un;
val Finite-Union = thm Finite-Union;
val Finite-induct = thm Finite-induct;
val Fin-imp-not-cons-lepoll = thm Fin-imp-not-cons-lepoll;
val Finite-imp-cardinal-cons = thm Finite-imp-cardinal-cons;
val Finite-imp-succ-cardinal-Diff = thm Finite-imp-succ-cardinal-Diff;
val Finite-imp-cardinal-Diff = thm Finite-imp-cardinal-Diff;
val nat-implies-well-ord = thm nat-implies-well-ord;
val nat-sum-eqpoll-sum = thm nat-sum-eqpoll-sum;
val Diff-sing-Finite = thm Diff-sing-Finite;
val Diff-Finite = thm Diff-Finite;
val Ord-subset-natD = thm Ord-subset-natD;
val Ord-nat-subset-into-Card = thm Ord-nat-subset-into-Card;
val Finite-cardinal-in-nat = thm Finite-cardinal-in-nat;
val Finite-Diff-sing-eq-diff-1 = thm Finite-Diff-sing-eq-diff-1;
val cardinal-lt-imp-Diff-not-0 = thm cardinal-lt-imp-Diff-not-0;
⟫

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*) == ⋃*n*∈*nat*. *F^n* (*x*)

notation (*xsymbols*)
iterates-omega ((-^ω '(-')) [*60*,*1000*] *60*)
notation (*HTML* **output**)
iterates-omega ((-^ω '(-')) [*60*,*1000*] *60*)

lemma *iterates-triv*:
[| *n*∈*nat*; *F*(*x*) = *x* |] ==> *F^n* (*x*) = *x*
by (*induct n rule*: *nat-induct*, *simp-all*)

lemma *iterates-type* [*TC*]:
[| *n*:*nat*; *a*: *A*; !!*x*. *x*:*A* ==> *F*(*x*) : *A* |]
==> *F^n* (*a*) : *A*
by (*induct n rule*: *nat-induct*, *simp-all*)

lemma *iterates-omega-triv*:
F(*x*) = *x* ==> *F^ω* (*x*) = *x*
by (*simp add*: *iterates-omega-def iterates-triv*)

lemma *Ord-iterates* [*simp*]:
[| *n*∈*nat*; !!*i*. *Ord*(*i*) ==> *Ord*(*F*(*i*)); *Ord*(*x*) |]
==> *Ord*(*F^n* (*x*))
by (*induct n rule*: *nat-induct*, *simp-all*)

lemma *iterates-commute*: *n* ∈ *nat* ==> *F*(*F^n* (*x*)) = *F^n* (*F*(*x*))
by (*induct-tac n*, *simp-all*)

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*, λ*x r*.
if x=*0 then a*
else if Limit(*x*) *then c*(*x*, λ*y*∈*x*. *r*'*y*)
else b(*Arith.pred*(*x*), *r* ' *Arith.pred*(*x*)))

lemma *transrec3-0* [*simp*]: *transrec3*(*0*,*a*,*b*,*c*) = *a*
by (*rule transrec3-def* [*THEN def-transrec*, *THEN trans*], *simp*)

lemma *transrec3-succ* [*simp*]:
transrec3(*succ*(*i*),*a*,*b*,*c*) = *b*(*i*, *transrec3*(*i*,*a*,*b*,*c*))

by (*rule transrec3-def* [*THEN def-transrec*, *THEN trans*], *simp*)

lemma *transrec3-Limit*:
  Limit(*i*) ==>
   transrec3(*i*,*a*,*b*,*c*) = *c*(*i*, λ*j*∈*i*. *transrec3*(*j*,*a*,*b*,*c*))
by (*rule transrec3-def* [*THEN def-transrec*, *THEN trans*], *force*)

declaration ⟪ *fn* - =>
 Simplifier.map-ss (*fn ss* => *ss setmksimps* (*map mk-eq o Ord-atomize o gen-all*))
⟫

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*)
apply (*case-tac A*=*0*)
apply (*simp add*: *Pi-empty1*)

apply (*blast intro*: *AC*)
done

lemma *AC-ball-Pi*: ∀ *x* ∈ *A*. ∃ *y*. *y* ∈ *B*(*x*) ==> ∃ *y*. *y* ∈ *Pi*(*A*,*B*)
apply (*rule AC-Pi*)
apply (*erule bspec*, *assumption*)
done

lemma *AC-Pi-Pow*: ∃ *f*. *f* ∈ (Π *X* ∈ *Pow*(*C*)−{*0*}. *X*)
apply (*rule-tac B1* = %*x*. *x* **in** *AC-Pi* [*THEN exE*])
apply (*erule-tac* [*2*] *exI*, *blast*)
done

lemma *AC-func*:
    [| !!x. x ∈ A ==> (∃ y. y ∈ x) |] ==> ∃ f ∈ A−>Union(A). ∀ x ∈ A. f'x ∈ x
apply (*rule-tac B1* = %x. x **in** *AC-Pi* [*THEN exE*])
prefer *2* **apply** (*blast dest*: *apply-type intro*: *Pi-type*, *blast*)
done

lemma *non-empty-family*: [| *0* ∉ A; x ∈ A |] ==> ∃ y. y ∈ x
by (*subgoal-tac* x ≠ *0*, *blast*+)

lemma *AC-func0*: *0* ∉ A ==> ∃ f ∈ A−>Union(A). ∀ x ∈ A. f'x ∈ x
apply (*rule AC-func*)
apply (*simp-all add*: *non-empty-family*)
done

lemma *AC-func-Pow*: ∃ f ∈ (Pow(C)−{*0*}) −> C. ∀ x ∈ Pow(C)−{*0*}. f'x ∈ x
apply (*rule AC-func0* [*THEN bexE*])
apply (*rule-tac* [*2*] *bexI*)
prefer *2* **apply** *assumption*
apply (*erule-tac* [*2*] *fun-weaken-type*, *blast*+)
done

lemma *AC-Pi0*: *0* ∉ A ==> ∃ f. f ∈ (Π x ∈ A. x)
apply (*rule AC-Pi*)
apply (*simp-all add*: *non-empty-family*)
done

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=>i **where**
   Subset-rel(A) == {z ∈ A∗A . ∃ x y. z=<x,y> & x<=y & x≠y}

definition
  chain :: i=>i **where**
   chain(A) == {F ∈ Pow(A). ∀ X∈F. ∀ Y∈F. X<=Y | Y<=X}

definition
  super :: [i,i]=>i **where**
   super(A,c) == {d ∈ chain(A). c<=d & c≠d}

definition

maxchain :: *i=>i* **where**
maxchain(A) == {*c* ∈ *chain(A)*. *super(A,c)=0*}

definition
increasing :: *i=>i* **where**
increasing(A) == {*f* ∈ *Pow(A)−>Pow(A)*. ∀ *x*. *x<=A −−> x<=f‘x*}

Lemma for the inductive definition below

lemma *Union-in-Pow*: *Y* ∈ *Pow(Pow(A))* ==> *Union(Y)* ∈ *Pow(A)*
by *blast*

We could make the inductive definition conditional on *next* ∈ *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*]=>*i*

inductive
domains *TFin(S,next)* <= *Pow(S)*
intros
nextI: [| *x* ∈ *TFin(S,next)*; *next* ∈ *increasing(S)* |]
==> *next‘x* ∈ *TFin(S,next)*

Pow-UnionI: *Y* ∈ *Pow(TFin(S,next))* ==> *Union(Y)* ∈ *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*: (∀ *x*∈*C*. *x<=A* | *B<=x*) ==> *Union(C)<=A* | *B<=Union(C)*
by *blast*

lemma *Inter-lemma0*:
[| *c* ∈ *C*; ∀ *x*∈*C*. *A<=x* | *x<=B* |] ==> *A* <= *Inter(C)* | *Inter(C)* <= *B*
by *blast*

36.2 The Transfinite Construction

lemma *increasingD1*: *f* ∈ *increasing(A)* ==> *f* ∈ *Pow(A)−>Pow(A)*
apply (*unfold increasing-def*)
apply (*erule CollectD1*)
done

lemma *increasingD2*: [| *f* ∈ *increasing(A)*; *x<=A* |] ==> *x* <= *f‘x*
by (*unfold increasing-def*, *blast*)

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 ∈ TFin(S,next);
     !!x. [| x ∈ TFin(S,next);  P(x);  next ∈ increasing(S) |] ==> P(next`x);
     !!Y. [| Y <= TFin(S,next);  ∀y∈Y. P(y) |] ==> P(Union(Y))
  |] ==> P(n)
by (erule TFin.induct, blast+)
```

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 ∈ TFin(S,next);  m ∈ TFin(S,next);
       ∀x ∈ TFin(S,next) . x<=m --> x=m | next`x<=m |]
     ==> n<=m | next`m<=n
apply (erule TFin-induct)
apply (erule-tac [2] Union-lemma0)

apply (blast dest: increasing-trans)
done
```

Lemma 2 of section 3.2. Interesting in its own right! Requires $next \in increasing(S)$ in the second induction step.

```
lemma TFin-linear-lemma2:
   [| m ∈ TFin(S,next);  next ∈ increasing(S) |]
    ==> ∀n ∈ TFin(S,next). n<=m --> n=m | next`n <= m
apply (erule TFin-induct)
apply (rule impI [THEN ballI])
```

case split using *TFin-linear-lemma1*

```
apply (rule-tac n1 = n and m1 = x in TFin-linear-lemma1 [THEN disjE],
       assumption+)
apply (blast del: subsetI
             intro: increasing-trans subsetI, blast)
```

second induction step

```
apply (rule impI [THEN ballI])
apply (rule Union-lemma0 [THEN disjE])
apply (erule-tac [3] disjI2)
prefer 2 apply blast
```

apply (*rule ballI*)
apply (*drule bspec*, *assumption*)
apply (*drule subsetD*, *assumption*)
apply (*rule-tac n1 = n* **and** *m1 = x* **in** *TFin-linear-lemma1* [*THEN disjE*],
  assumption+, *blast*)
apply (*erule increasingD2* [*THEN subset-trans*, *THEN disjI1*])
apply (*blast dest*: *TFin-is-subset*)+
done

a more convenient form for Lemma 2

lemma *TFin-subsetD*:
  [| *n<=m*; *m* ∈ *TFin*(*S*,*next*); *n* ∈ *TFin*(*S*,*next*); *next* ∈ *increasing*(*S*) |]
  ==> *n=m* | *next`n <= m*
by (*blast dest*: *TFin-linear-lemma2* [*rule-format*])

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*:
  [| *m* ∈ *TFin*(*S*,*next*); *n* ∈ *TFin*(*S*,*next*); *next* ∈ *increasing*(*S*) |]
  ==> *n <= m* | *m<=n*
apply (*rule disjE*)
apply (*rule TFin-linear-lemma1* [*OF - -TFin-linear-lemma2*])
apply (*assumption+*, *erule disjI2*)
apply (*blast del*: *subsetI*
    intro: *subsetI increasingD2* [*THEN subset-trans*] *TFin-is-subset*)
done

Lemma 3 of section 3.3

lemma *equal-next-upper*:
  [| *n* ∈ *TFin*(*S*,*next*); *m* ∈ *TFin*(*S*,*next*); *m = next`m* |] ==> *n <= m*
apply (*erule TFin-induct*)
apply (*drule TFin-subsetD*)
apply (*assumption+*, *force*, *blast*)
done

Property 3.3 of section 3.3

lemma *equal-next-Union*:
  [| *m* ∈ *TFin*(*S*,*next*); *next* ∈ *increasing*(*S*) |]
  ==> *m = next`m* <-> *m = Union*(*TFin*(*S*,*next*))
apply (*rule iffI*)
apply (*rule Union-upper* [*THEN equalityI*])
apply (*rule-tac* [*2*] *equal-next-upper* [*THEN Union-least*])
apply (*assumption+*)
apply (*erule ssubst*)
apply (*rule increasingD2* [*THEN equalityI*], *assumption*)
apply (*blast del*: *subsetI*
    intro: *subsetI TFin-UnionI TFin.nextI TFin-is-subset*)+
done

36.4 Hausdorff's Theorem: Every Set Contains a Maximal Chain

NOTE: We assume the partial ordering is ⊆, the subset relation!

* Defining the "next" operation for Hausdorff's Theorem *

lemma *chain-subset-Pow*: *chain*(*A*) <= *Pow*(*A*)
apply (*unfold chain-def*)
apply (*rule Collect-subset*)
done

lemma *super-subset-chain*: *super*(*A*,*c*) <= *chain*(*A*)
apply (*unfold super-def*)
apply (*rule Collect-subset*)
done

lemma *maxchain-subset-chain*: *maxchain*(*A*) <= *chain*(*A*)
apply (*unfold maxchain-def*)
apply (*rule Collect-subset*)
done

lemma *choice-super*:
[| *ch* ∈ (Π *X* ∈ *Pow*(*chain*(*S*)) − {*0*}. *X*); *X* ∈ *chain*(*S*); *X* ∉ *maxchain*(*S*) |]
==> *ch* ‘ *super*(*S*,*X*) ∈ *super*(*S*,*X*)
apply (*erule apply-type*)
apply (*unfold super-def maxchain-def*, *blast*)
done

lemma *choice-not-equals*:
[| *ch* ∈ (Π *X* ∈ *Pow*(*chain*(*S*)) − {*0*}. *X*); *X* ∈ *chain*(*S*); *X* ∉ *maxchain*(*S*) |]
==> *ch* ‘ *super*(*S*,*X*) ≠ *X*
apply (*rule notI*)
apply (*drule choice-super*, *assumption*, *assumption*)
apply (*simp add*: *super-def*)
done

This justifies Definition 4.4

lemma *Hausdorff-next-exists*:
ch ∈ (Π *X* ∈ *Pow*(*chain*(*S*))−{*0*}. *X*) ==>
∃ *next* ∈ *increasing*(*S*). ∀ *X* ∈ *Pow*(*S*).
next‘*X* = *if*(*X* ∈ *chain*(*S*)−*maxchain*(*S*), *ch*‘*super*(*S*,*X*), *X*)
apply (*rule-tac x*=λ*X*∈*Pow*(*S*).
if X ∈ *chain*(*S*) − *maxchain*(*S*) *then ch* ‘ *super*(*S*, *X*) *else X*
in *bexI*)
apply *force*
apply (*unfold increasing-def*)
apply (*rule CollectI*)

apply (*rule lam-type*)
apply (*simp* (*no-asm-simp*))
apply (*blast dest*: *super-subset-chain* [*THEN subsetD*]
chain-subset-Pow [*THEN subsetD*] *choice-super*)

Now, verify that it increases

apply (*simp* (*no-asm-simp*) *add*: *Pow-iff subset-refl*)
apply *safe*
apply (*drule choice-super*)
apply (*assumption*+)
apply (*simp add*: *super-def*, *blast*)
done

Lemma 4

lemma *TFin-chain-lemma4*:
[| *c* ∈ *TFin*(*S*,*next*);
ch ∈ (Π *X* ∈ *Pow*(*chain*(*S*))−{*0*}. *X*);
next ∈ *increasing*(*S*);
∀ *X* ∈ *Pow*(*S*). *next*'*X* =
if(*X* ∈ *chain*(*S*)−*maxchain*(*S*), *ch*'*super*(*S*,*X*), *X*) |]
==> *c* ∈ *chain*(*S*)
apply (*erule TFin-induct*)
apply (*simp* (*no-asm-simp*) *add*: *chain-subset-Pow* [*THEN subsetD*, *THEN PowD*]
choice-super [*THEN super-subset-chain* [*THEN subsetD*]])
apply (*unfold chain-def*)
apply (*rule CollectI*, *blast*, *safe*)
apply (*rule-tac m1*=*B* **and** *n1*=*Ba* **in** *TFin-subset-linear* [*THEN disjE*], *fast*+)

Blast-tac's slow

done

theorem *Hausdorff*: ∃ *c*. *c* ∈ *maxchain*(*S*)
apply (*rule AC-Pi-Pow* [*THEN exE*])
apply (*rule Hausdorff-next-exists* [*THEN bexE*], *assumption*)
apply (*rename-tac ch next*)
apply (*subgoal-tac Union* (*TFin* (*S*,*next*)) ∈ *chain* (*S*))
prefer *2*
apply (*blast intro*!: *TFin-chain-lemma4 subset-refl* [*THEN TFin-UnionI*])
apply (*rule-tac x* = *Union* (*TFin* (*S*,*next*)) **in** *exI*)
apply (*rule classical*)
apply (*subgoal-tac next* ' *Union* (*TFin* (*S*,*next*)) = *Union* (*TFin* (*S*,*next*)))
apply (*rule-tac* [*2*] *equal-next-Union* [*THEN iffD2*, *symmetric*])
apply (*rule-tac* [*2*] *subset-refl* [*THEN TFin-UnionI*])
prefer *2* **apply** *assumption*
apply (*rule-tac* [*2*] *refl*)
apply (*simp add*: *subset-refl* [*THEN TFin-UnionI*,
THEN TFin.dom-subset [*THEN subsetD*, *THEN PowD*]])
apply (*erule choice-not-equals* [*THEN notE*])
apply (*assumption*+)
done

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*:
  [| $c \in chain(A)$; $z \in A$; $\forall x \in c.\ x<=z$ |] ==> $cons(z,c) \in chain(A)$
by (*unfold chain-def*, *blast*)

lemma *Zorn*: $\forall c \in chain(S).\ Union(c) \in S$ ==> $\exists y \in S.\ \forall z \in S.\ y<=z$ --> $y=z$
apply (*rule Hausdorff* [*THEN exE*])
apply (*simp add*: *maxchain-def*)
apply (*rename-tac c*)
apply (*rule-tac x = Union* (*c*) **in** *bexI*)
prefer *2* **apply** *blast*
apply *safe*
apply (*rename-tac z*)
apply (*rule classical*)
apply (*subgoal-tac cons* (*z*,*c*) ∈ *super* (*S*,*c*))
apply (*blast elim*: *equalityE*)
apply (*unfold super-def*, *safe*)
apply (*fast elim*: *chain-extend*)
apply (*fast elim*: *equalityE*)
done

Alternative version of Zorn's Lemma

theorem *Zorn2*:
 $\forall c \in chain(S).\ \exists y \in S.\ \forall x \in c.\ x <= y$ ==> $\exists y \in S.\ \forall z \in S.\ y<=z$ --> $y=z$
apply (*cut-tac Hausdorff maxchain-subset-chain*)
apply (*erule exE*)
apply (*drule subsetD*, *assumption*)
apply (*drule bspec*, *assumption*, *erule bexE*)
apply (*rule-tac x = y* **in** *bexI*)
 prefer *2* **apply** *assumption*
apply *clarify*
apply *rule* **apply** *assumption*
apply *rule*
apply (*rule ccontr*)
apply (*frule-tac z=z* **in** *chain-extend*)
 apply (*assumption*, *blast*)
apply (*unfold maxchain-def super-def*)
apply (*blast elim*!: *equalityCE*)
done

36.6 Zermelo's Theorem: Every Set can be Well-Ordered

Lemma 5

lemma *TFin-well-lemma5*:
[| $n \in TFin(S,next)$; $Z <= TFin(S,next)$; $z{:}Z$; ~ $Inter(Z) \in Z$ |]
==> $\forall\, m \in Z.\ n <= m$
apply (*erule TFin-induct*)
prefer *2* **apply** *blast*

second induction step is easy

apply (*rule ballI*)
apply (*rule bspec* [*THEN TFin-subsetD*, *THEN disjE*], *auto*)
apply (*subgoal-tac m* = *Inter* (*Z*))
apply *blast+*
done

Well-ordering of *TFin*(*S*, *next*)

lemma *well-ord-TFin-lemma*: [| $Z <= TFin(S,next)$; $z \in Z$ |] ==> $Inter(Z) \in Z$
apply (*rule classical*)
apply (*subgoal-tac Z* = {*Union* (*TFin* (*S*,*next*))})
apply (*simp* (*no-asm-simp*) *add*: *Inter-singleton*)
apply (*erule equal-singleton*)
apply (*rule Union-upper* [*THEN equalityI*])
apply (*rule-tac* [*2*] *subset-refl* [*THEN TFin-UnionI*, *THEN TFin-well-lemma5*, *THEN bspec*], *blast+*)
done

This theorem just packages the previous result

lemma *well-ord-TFin*:
next ∈ *increasing*(*S*)
==> *well-ord*(*TFin*(*S*,*next*), *Subset-rel*(*TFin*(*S*,*next*)))
apply (*rule well-ordI*)
apply (*unfold Subset-rel-def linear-def*)

Prove the well-foundedness goal

apply (*rule wf-onI*)
apply (*frule well-ord-TFin-lemma*, *assumption*)
apply (*drule-tac x* = *Inter* (*Z*) **in** *bspec*, *assumption*)
apply *blast*

Now prove the linearity goal

apply (*intro ballI*)
apply (*case-tac x*=*y*)
apply *blast*

The $x \neq y$ case remains

apply (*rule-tac n1*=*x* **and** *m1*=*y* **in** *TFin-subset-linear* [*THEN disjE*],
assumption+, *blast+*)
done

* Defining the "next" operation for Zermelo's Theorem *

lemma *choice-Diff*:
[| $ch \in (\Pi\ X \in Pow(S) - \{0\}.\ X)$; $X \subseteq S$; $X{\neq}S$ |] ==> ch ‘ $(S{-}X) \in S{-}X$
apply (*erule apply-type*)
apply (*blast elim*!: *equalityE*)
done

This justifies Definition 6.1

lemma *Zermelo-next-exists*:
$ch \in (\Pi\ X \in Pow(S){-}\{0\}.\ X)$ ==>
$\exists\, next \in increasing(S).\ \forall X \in Pow(S).$
$next`X = (if\ X{=}S\ then\ S\ else\ cons(ch`(S{-}X),\ X))$
apply (*rule-tac* $x{=}\lambda X{\in}Pow(S).\ if\ X{=}S\ then\ S\ else\ cons(ch`(S{-}X),\ X)$
in *bexI*)
apply *force*
apply (*unfold increasing-def*)
apply (*rule CollectI*)
apply (*rule lam-type*)

Type checking is surprisingly hard!

apply (*simp* (*no-asm-simp*) *add*: *Pow-iff cons-subset-iff subset-refl*)
apply (*blast intro*!: *choice-Diff* [*THEN DiffD1*])

Verify that it increases

apply (*intro allI impI*)
apply (*simp add*: *Pow-iff subset-consI subset-refl*)
done

The construction of the injection

lemma *choice-imp-injection*:
[| $ch \in (\Pi\ X \in Pow(S){-}\{0\}.\ X)$;
$next \in increasing(S)$;
$\forall X \in Pow(S).\ next`X = if(X{=}S,\ S,\ cons(ch`(S{-}X),\ X))$ |]
==> $(\lambda\ x \in S.\ Union(\{y \in TFin(S,next).\ x \notin y\}))$
$\in inj(S,\ TFin(S,next) - \{S\})$
apply (*rule-tac* d = %y. ch‘ $(S{-}y)$ **in** *lam-injective*)
apply (*rule DiffI*)
apply (*rule Collect-subset* [*THEN TFin-UnionI*])
apply (*blast intro*!: *Collect-subset* [*THEN TFin-UnionI*] *elim*: *equalityE*)
apply (*subgoal-tac* $x \notin Union\ (\{y \in TFin\ (S,next)\ .\ x \notin y\})$)
prefer *2* **apply** (*blast elim*: *equalityE*)
apply (*subgoal-tac* $Union\ (\{y \in TFin\ (S,next)\ .\ x \notin y\}) \neq S$)
prefer *2* **apply** (*blast elim*: *equalityE*)

For proving $x \in next`Union(...)$. Abrial and Laffitte's justification appears to be faulty.

apply (*subgoal-tac* ~ $next$ ‘ $Union\ (\{y \in TFin\ (S,next)\ .\ x \notin y\})$
<= $Union\ (\{y \in TFin\ (S,next)\ .\ x \notin y\})$)

```
 prefer 2
 apply (simp del: Union-iff
             add: Collect-subset [THEN TFin-UnionI, THEN TFin-is-subset]
             Pow-iff cons-subset-iff subset-refl choice-Diff [THEN DiffD2])
apply (subgoal-tac x ∈ next ‘ Union ({y ∈ TFin (S,next) . x ∉ y}) )
 prefer 2
 apply (blast intro!: Collect-subset [THEN TFin-UnionI] TFin.nextI)
```

End of the lemmas!

```
apply (simp add: Collect-subset [THEN TFin-UnionI, THEN TFin-is-subset])
done
```

The wellordering theorem

```
theorem AC-well-ord: ∃ r. well-ord(S,r)
apply (rule AC-Pi-Pow [THEN exE])
apply (rule Zermelo-next-exists [THEN bexE], assumption)
apply (rule exI)
apply (rule well-ord-rvimage)
apply (erule-tac [2] well-ord-TFin)
apply (rule choice-imp-injection [THEN inj-weaken-type], blast+)
done
```

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 ⊆ s ==> Chain(r) ⊆ Chain(s)
  unfolding Chain-def
  by blast

theorem Zorn-po:
  assumes po: Partial-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 −
  have Preorder(r) using po by (simp add: partial-order-on-def)
  — Mirror r in the set of subsets below (wrt r) elements of A (?).
  let ?B = lam x:field(r). r −‘‘ {x} let ?S = ?B ‘‘ field(r)
  have ALL C:chain(?S). EX U:?S. ALL A:C. A ⊆ U
  proof (clarsimp simp: chain-def Subset-rel-def bex-image-simp)
    fix C
    assume 1: C ⊆ ?S and 2: ALL A:C. ALL B:C. A ⊆ B | B ⊆ A
    let ?A = {x : field(r). EX M:C. M = ?B‘x}
    have C = ?B ‘‘ ?A using 1
      apply (auto simp: image-def)
```

apply *rule*
apply *rule*
apply (*drule subsetD*) **apply** *assumption*
apply (*erule CollectE*)
apply *rule* **apply** *assumption*
apply (*erule bexE*)
apply *rule* **prefer** *2* **apply** *assumption*
apply *rule*
apply (*erule lamE*) **apply** *simp*
apply *assumption*

apply (*thin-tac C* ⊆ *?X*)
apply (*fast elim*: *lamE*)
done
have *?A* : *Chain*(*r*)
proof (*simp add*: *Chain-def subsetI*, *intro conjI ballI impI*)
fix *a b*
assume *a* : *field*(*r*) *r* −“ {*a*} : *C b* : *field*(*r*) *r* −“ {*b*} : *C*
hence *r* −“ {*a*} ⊆ *r* −“ {*b*} | *r* −“ {*b*} ⊆ *r* −“ {*a*} **using** *2* **by** *auto*
then show <*a*, *b*> : *r* | <*b*, *a*> : *r*
using ‹*Preorder*(*r*)› ‹*a* : *field*(*r*)› ‹*b* : *field*(*r*)›
by (*simp add*: *subset-vimage1-vimage1-iff*)
qed
then obtain *u* **where** *uA*: *u* : *field*(*r*) *ALL a*:*?A*. <*a*, *u*> : *r*
using *u*
apply *auto*
apply (*drule bspec*) **apply** *assumption*
apply *auto*
done
have *ALL A*:*C*. *A* ⊆ *r* −“ {*u*}
proof (*auto intro*!: *vimageI*)
fix *a B*
assume *aB*: *B* : *C a* : *B*
with *1* **obtain** *x* **where** *x* : *field*(*r*) *B* = *r* −“ {*x*}
apply −
apply (*drule subsetD*) **apply** *assumption*
apply (*erule imageE*)
apply (*erule lamE*)
apply *simp*
done
then show <*a*, *u*> : *r* **using** *uA aB* ‹*Preorder*(*r*)›
by (*auto simp*: *preorder-on-def refl-def*) (*blast dest*: *trans-onD*)+
qed
then show *EX U*:*field*(*r*). *ALL A*:*C*. *A* ⊆ *r* −“ {*U*}
using ‹*u* : *field*(*r*)› **..**
qed
from *Zorn2* [*OF this*]
obtain *m B* **where** *m* : *field*(*r*) *B* = *r* −“ {*m*}
ALL x:*field*(*r*). *B* ⊆ *r* −“ {*x*} −−> *B* = *r* −“ {*x*}

```
    by (auto elim!: lamE simp: ball-image-simp)
  then have ALL a:field(r). &lt;m, a&gt; : r --> a = m
    using po ‹Preorder(r)› ‹m : field(r)›
   by (auto simp: subset-vimage1-vimage1-iff Partial-order-eq-vimage1-vimage1-iff)
  then show ?thesis using ‹m : field(r)› by blast
qed

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
apply (rule AC-well-ord [THEN exE])
apply (erule well-ord-cardinal-eqpoll)
done
```

The theorem $||A|| = |A|$

```
lemmas cardinal-idem = cardinal-eqpoll [THEN cardinal-cong, standard, simp]

lemma cardinal-eqE: |X| = |Y| ==> X eqpoll Y
apply (rule AC-well-ord [THEN exE])
apply (rule AC-well-ord [THEN exE])
apply (rule well-ord-cardinal-eqE, assumption+)
done

lemma cardinal-eqpoll-iff: |X| = |Y| <-> X eqpoll Y
by (blast intro: cardinal-cong cardinal-eqE)

lemma cardinal-disjoint-Un:
    [| |A|=|B|;  |C|=|D|;  A Int C = 0;  B Int D = 0 |]
     ==> |A Un C| = |B Un D|
by (simp add: cardinal-eqpoll-iff eqpoll-disjoint-Un)

lemma lepoll-imp-Card-le: A lepoll B ==> |A| le |B|
apply (rule AC-well-ord [THEN exE])
apply (erule well-ord-lepoll-imp-Card-le, assumption)
done

lemma cadd-assoc: (i |+| j) |+| k = i |+| (j |+| k)
apply (rule AC-well-ord [THEN exE])
apply (rule AC-well-ord [THEN exE])
apply (rule AC-well-ord [THEN exE])
apply (rule well-ord-cadd-assoc, assumption+)
done
```

lemma *cmult-assoc*: (i |*| j) |*| k = i |*| (j |*| k)
apply (*rule AC-well-ord* [*THEN exE*])
apply (*rule AC-well-ord* [*THEN exE*])
apply (*rule AC-well-ord* [*THEN exE*])
apply (*rule well-ord-cmult-assoc*, *assumption*+)
done

lemma *cadd-cmult-distrib*: (i |+| j) |*| k = (i |*| k) |+| (j |*| k)
apply (*rule AC-well-ord* [*THEN exE*])
apply (*rule AC-well-ord* [*THEN exE*])
apply (*rule AC-well-ord* [*THEN exE*])
apply (*rule well-ord-cadd-cmult-distrib*, *assumption*+)
done

lemma *InfCard-square-eq*: InfCard(|A|) ==> A*A eqpoll A
apply (*rule AC-well-ord* [*THEN exE*])
apply (*erule well-ord-InfCard-square-eq*, *assumption*)
done

37.2 The relationship between cardinality and le-pollence

lemma *Card-le-imp-lepoll*: |A| le |B| ==> A lepoll B
apply (*rule cardinal-eqpoll*
[*THEN eqpoll-sym*, *THEN eqpoll-imp-lepoll*, *THEN lepoll-trans*])
apply (*erule le-imp-subset* [*THEN subset-imp-lepoll*, *THEN lepoll-trans*])
apply (*rule cardinal-eqpoll* [*THEN eqpoll-imp-lepoll*])
done

lemma *le-Card-iff*: Card(K) ==> |A| le K <-> A lepoll K
apply (*erule Card-cardinal-eq* [*THEN subst*], *rule iffI*,
erule Card-le-imp-lepoll)
apply (*erule lepoll-imp-Card-le*)
done

lemma *cardinal-0-iff-0* [*simp*]: |A| = 0 <-> A = 0
apply *auto*
apply (*drule cardinal-0* [*THEN ssubst*])
apply (*blast intro*: *eqpoll-0-iff* [*THEN iffD1*] *cardinal-eqpoll-iff* [*THEN iffD1*])
done

lemma *cardinal-lt-iff-lesspoll*: Ord(i) ==> i < |A| <-> i lesspoll A
apply (*cut-tac A = A* **in** *cardinal-eqpoll*)
apply (*auto simp add*: *eqpoll-iff*)
apply (*blast intro*: *lesspoll-trans2 lt-Card-imp-lesspoll Card-cardinal*)
apply (*force intro*: *cardinal-lt-imp-lt lesspoll-cardinal-lt lesspoll-trans2*
simp add: *cardinal-idem*)
done

```
lemma cardinal-le-imp-lepoll: i ≤ |A| ==> i ≲ A
apply (blast intro: lt-Ord Card-le-imp-lepoll Ord-cardinal-le le-trans)
done
```

37.3 Other Applications of AC

```
lemma surj-implies-inj: f: surj(X,Y) ==> EX g. g: inj(Y,X)
apply (unfold surj-def)
apply (erule CollectE)
apply (rule-tac A1 = Y and B1 = %y. f-``{y} in AC-Pi [THEN exE])
apply (fast elim!: apply-Pair)
apply (blast dest: apply-type Pi-memberD
          intro: apply-equality Pi-type f-imp-injective)
done


lemma surj-implies-cardinal-le: f: surj(X,Y) ==> |Y| le |X|
apply (rule lepoll-imp-Card-le)
apply (erule surj-implies-inj [THEN exE])
apply (unfold lepoll-def)
apply (erule exI)
done


lemma cardinal-UN-le:
     [| InfCard(K);  ALL i:K. |X(i)| le K |] ==> |⋃i∈K. X(i)| le K
apply (simp add: InfCard-is-Card le-Card-iff)
apply (rule lepoll-trans)
 prefer 2
 apply (rule InfCard-square-eq [THEN eqpoll-imp-lepoll])
 apply (simp add: InfCard-is-Card Card-cardinal-eq)
apply (unfold lepoll-def)
apply (frule InfCard-is-Card [THEN Card-is-Ord])
apply (erule AC-ball-Pi [THEN exE])
apply (rule exI)

apply (subgoal-tac ALL z: (⋃i∈K. X (i)). z: X (LEAST i. z:X (i)) &
                  (LEAST i. z:X (i)) : K)
 prefer 2
 apply (fast intro!: Least-le [THEN lt-trans1, THEN ltD] ltI
           elim!: LeastI Ord-in-Ord)
apply (rule-tac c = %z. <LEAST i. z:X (i), f ` (LEAST i. z:X (i)) ` z>
          and d = %<i,j>. converse (f`i) ` j in lam-injective)

by (blast intro: inj-is-fun [THEN apply-type] dest: apply-type, force)



lemma cardinal-UN-lt-csucc:
```

[| *InfCard*(*K*); *ALL* *i*:*K*. |*X*(*i*)| < *csucc*(*K*) |]
==> |⋃*i*∈*K*. *X*(*i*)| < *csucc*(*K*)
by (*simp add*: *Card-lt-csucc-iff cardinal-UN-le InfCard-is-Card Card-cardinal*)

lemma *cardinal-UN-Ord-lt-csucc*:
[| *InfCard*(*K*); *ALL* *i*:*K*. *j*(*i*) < *csucc*(*K*) |]
==> (⋃*i*∈*K*. *j*(*i*)) < *csucc*(*K*)
apply (*rule cardinal-UN-lt-csucc* [*THEN Card-lt-imp-lt*], *assumption*)
apply (*blast intro*: *Ord-cardinal-le* [*THEN lt-trans1*] *elim*: *ltE*)
apply (*blast intro*!: *Ord-UN elim*: *ltE*)
apply (*erule InfCard-is-Card* [*THEN Card-is-Ord*, *THEN Card-csucc*])
done

lemma *inj-UN-subset*:
[| *f*: *inj*(*A*,*B*); *a*:*A* |] ==>
(⋃*x*∈*A*. *C*(*x*)) <= (⋃*y*∈*B*. *C*(*if y*: *range*(*f*) *then converse*(*f*)‘*y else a*))
apply (*rule UN-least*)
apply (*rule-tac x1*= *f*‘*x* **in** *subset-trans* [*OF - UN-upper*])
apply (*simp add*: *inj-is-fun* [*THEN apply-rangeI*])
apply (*blast intro*: *inj-is-fun* [*THEN apply-type*])
done

lemma *le-UN-Ord-lt-csucc*:
[| *InfCard*(*K*); |*W*| *le K*; *ALL* *w*:*W*. *j*(*w*) < *csucc*(*K*) |]
==> (⋃*w*∈*W*. *j*(*w*)) < *csucc*(*K*)
apply (*case-tac W*=*0*)

apply (*simp add*: *InfCard-is-Card Card-is-Ord* [*THEN Card-csucc*]
Card-is-Ord Ord-0-lt-csucc)
apply (*simp add*: *InfCard-is-Card le-Card-iff lepoll-def*)
apply (*safe intro*!: *equalityI*)
apply (*erule swap*)
apply (*rule lt-subset-trans* [*OF inj-UN-subset cardinal-UN-Ord-lt-csucc*], *assumption+*)
apply (*simp add*: *inj-converse-fun* [*THEN apply-type*])
apply (*blast intro*!: *Ord-UN elim*: *ltE*)
done

ML
⟪
val cardinal-0-iff-0 = *thm cardinal-0-iff-0*;
val cardinal-lt-iff-lesspoll = *thm cardinal-lt-iff-lesspoll*;
⟫

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*)
apply (*rule-tac x* = ⋃ *d*∈*D*. *LEAST i*. *f*'*d* : *Vfrom* (*A*,*i*) **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* [*2*] *le-UN-Ord-lt-csucc*)
apply (*rule-tac* [*4*] *ballI*, *erule-tac* [*4*] *fun-Limit-VfromE*, *simp-all*)
prefer *2* **apply** (*fast elim*: *Least-le* [*THEN lt-trans1*] *ltE*)
apply (*rule Pi-type*)
apply (*rename-tac* [*2*] *d*)
apply (*erule-tac* [*2*] *fun-Limit-VfromE*, *simp-all*)
apply (*subgoal-tac f*'*d* : *Vfrom* (*A*, *LEAST i*. *f*'*d* : *Vfrom* (*A*,*i*)))
 apply (*erule Vfrom-mono* [*OF subset-refl UN-upper*, *THEN subsetD*])
 apply *assumption*
apply (*fast elim*: *LeastI ltE*)
done

lemma *subset-Vcsucc*:
  [| *D* <= *Vfrom*(*A*,*csucc*(*K*)); |*D*| *le K*; *InfCard*(*K*) |]
  ==> *EX j*. *D* <= *Vfrom*(*A*,*j*) & *j* < *csucc*(*K*)
by (*simp add*: *subset-iff-id fun-Vcsucc-lemma*)

lemma *fun-Vcsucc*:
  [| |*D*| *le K*; *InfCard*(*K*); *D* <= *Vfrom*(*A*,*csucc*(*K*)) |] ==>
    D −> *Vfrom*(*A*,*csucc*(*K*)) <= *Vfrom*(*A*,*csucc*(*K*))
apply (*safe dest*!: *fun-Vcsucc-lemma subset-Vcsucc*)
apply (*rule Vfrom* [*THEN ssubst*])
apply (*drule fun-is-rel*)

apply (*rule-tac a1* = *succ* (*succ* (*j Un ja*)) **in** *UN-I* [*THEN UnI2*])
apply (*blast intro*: *ltD InfCard-csucc InfCard-is-Limit Limit-has-succ*
      Un-least-lt)
apply (*erule subset-trans* [*THEN PowI*])
apply (*fast intro*: *Pair-in-Vfrom Vfrom-UnI1 Vfrom-UnI2*)
done

lemma *fun-in-Vcsucc*:
[| *f*: *D* −> *Vfrom*(*A*, *csucc*(*K*)); |*D*| *le K*; *InfCard*(*K*);
D <= *Vfrom*(*A*,*csucc*(*K*)) |]
==> *f*: *Vfrom*(*A*,*csucc*(*K*))
by (*blast intro*: *fun-Vcsucc* [*THEN subsetD*])

lemmas *fun-in-Vcsucc′* = *fun-in-Vcsucc* [*OF - - - subsetI*]

lemma *Card-fun-Vcsucc*:
InfCard(*K*) ==> *K* −> *Vfrom*(*A*,*csucc*(*K*)) <= *Vfrom*(*A*,*csucc*(*K*))
apply (*frule InfCard-is-Card* [*THEN Card-is-Ord*])
apply (*blast del*: *subsetI*
intro: *fun-Vcsucc Ord-cardinal-le i-subset-Vfrom*
lt-csucc [*THEN leI*, *THEN le-imp-subset*, *THEN subset-trans*])
done

lemma *Card-fun-in-Vcsucc*:
[| *f*: *K* −> *Vfrom*(*A*, *csucc*(*K*)); *InfCard*(*K*) |] ==> *f*: *Vfrom*(*A*,*csucc*(*K*))
by (*blast intro*: *Card-fun-Vcsucc* [*THEN subsetD*])

lemma *Limit-csucc*: *InfCard*(*K*) ==> *Limit*(*csucc*(*K*))
by (*erule InfCard-csucc* [*THEN InfCard-is-Limit*])

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