Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

December 12, 2021

Contents

1 Sample datatype definitions

theory *Datatypes* **imports** *ZF* **begin**

1.1 A type with four constructors

It has four contructors, of arities 0–3, and two parameters A and B.

```
consts
  data :: [i, i] => i

datatype data(A, B) =
    Con0
  | Con1 (a ∈ A)
  | Con2 (a ∈ A, b ∈ B)
  | Con3 (a ∈ A, b ∈ B, d ∈ data(A, B))

lemma data-unfold: data(A, B) = ({0} + A) + (A × B + A × B × data(A, B))
  by (fast intro!: data.intros [unfolded data.con-defs]
    elim: data.cases [unfolded data.con-defs])
```

Lemmas to justify using *data* in other recursive type definitions.

```
lemma data-mono: [| A ⊆ C; B ⊆ D |] ==> data(A, B) ⊆ data(C, D)
  apply (unfold data.defs)
  apply (rule lfp-mono)
   apply (rule data.bnd-mono)+
  apply (rule univ-mono Un-mono basic-monos | assumption)+
  done

lemma data-univ: data(univ(A), univ(A)) ⊆ univ(A)
  apply (unfold data.defs data.con-defs)
  apply (rule lfp-lowerbound)
  apply (rule-tac [2] subset-trans [OF A-subset-univ Un-upper1, THEN univ-mono])
  apply (fast intro!: zero-in-univ Inl-in-univ Inr-in-univ Pair-in-univ)
  done

lemma data-subset-univ:
    [| A ⊆ univ(C); B ⊆ univ(C) |] ==> data(A, B) ⊆ univ(C)
  by (rule subset-trans [OF data-mono data-univ])
```

1.2 Example of a big enumeration type

Can go up to at least 100 constructors, but it takes nearly 7 minutes ... (back in 1994 that is).

```
consts
  enum :: i

datatype enum =
    C00 | C01 | C02 | C03 | C04 | C05 | C06 | C07 | C08 | C09
  | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 | C19
  | C20 | C21 | C22 | C23 | C24 | C25 | C26 | C27 | C28 | C29
  | C30 | C31 | C32 | C33 | C34 | C35 | C36 | C37 | C38 | C39
  | C40 | C41 | C42 | C43 | C44 | C45 | C46 | C47 | C48 | C49
```

| *C50* | *C51* | *C52* | *C53* | *C54* | *C55* | *C56* | *C57* | *C58* | *C59*

end

2 Binary trees

theory *Binary-Trees* **imports** *ZF* **begin**

2.1 Datatype definition

consts
bt :: *i* => *i*

datatype *bt*(*A*) =
Lf | *Br* (*a* ∈ *A*, *t1* ∈ *bt*(*A*), *t2* ∈ *bt*(*A*))

declare *bt.intros* [*simp*]

lemma *Br-neq-left*: *l* ∈ *bt*(*A*) ==> *Br*(*x*, *l*, *r*) ≠ *l*
by (*induct arbitrary*: *x r set*: *bt*) *auto*

lemma *Br-iff*: *Br*(*a*, *l*, *r*) = *Br*(*a'*, *l'*, *r'*) ⟷ *a* = *a'* & *l* = *l'* & *r* = *r'*
— Proving a freeness theorem.
by (*fast elim*!: *bt.free-elims*)

inductive-cases *BrE*: *Br*(*a*, *l*, *r*) ∈ *bt*(*A*)
— An elimination rule, for type-checking.

Lemmas to justify using *bt* in other recursive type definitions.

lemma *bt-mono*: *A* ⊆ *B* ==> *bt*(*A*) ⊆ *bt*(*B*)
apply (*unfold bt.defs*)
apply (*rule lfp-mono*)
apply (*rule bt.bnd-mono*)+
apply (*rule univ-mono basic-monos* | *assumption*)+
done

lemma *bt-univ*: *bt*(*univ*(*A*)) ⊆ *univ*(*A*)
apply (*unfold bt.defs bt.con-defs*)
apply (*rule lfp-lowerbound*)
apply (*rule-tac* [*2*] *A-subset-univ* [*THEN univ-mono*])
apply (*fast intro*!: *zero-in-univ Inl-in-univ Inr-in-univ Pair-in-univ*)
done

lemma *bt-subset-univ*: *A* ⊆ *univ*(*B*) ==> *bt*(*A*) ⊆ *univ*(*B*)
apply (*rule subset-trans*)
apply (*erule bt-mono*)
apply (*rule bt-univ*)
done

lemma *bt-rec-type*:
  [| $t \in bt(A)$;
    $c \in C(Lf)$;
    !!$x\ y\ z\ r\ s$. [| $x \in A$; $y \in bt(A)$; $z \in bt(A)$; $r \in C(y)$; $s \in C(z)$ |] ==>
    $h(x,\ y,\ z,\ r,\ s) \in C(Br(x,\ y,\ z))$
  |] ==> $bt\text{-}rec(c,\ h,\ t) \in C(t)$
  — Type checking for recursor – example only; not really needed.
  apply (*induct-tac t*)
   apply *simp-all*
  done

2.2 Number of nodes, with an example of tail-recursion

consts *n-nodes* :: i => i
primrec
  $n\text{-}nodes(Lf) = 0$
  $n\text{-}nodes(Br(a,\ l,\ r)) = succ(n\text{-}nodes(l)\ \#+\ n\text{-}nodes(r))$

lemma *n-nodes-type* [*simp*]: $t \in bt(A)$ ==> $n\text{-}nodes(t) \in nat$
  by (*induct set*: *bt*) *auto*

consts *n-nodes-aux* :: i => i
primrec
  $n\text{-}nodes\text{-}aux(Lf) = (\lambda k \in nat.\ k)$
  $n\text{-}nodes\text{-}aux(Br(a,\ l,\ r)) =$
      $(\lambda k \in nat.\ n\text{-}nodes\text{-}aux(r)\ \text{‘}\ (n\text{-}nodes\text{-}aux(l)\ \text{‘}\ succ(k)))$

lemma *n-nodes-aux-eq*:
    $t \in bt(A)$ ==> $k \in nat$ ==> $n\text{-}nodes\text{-}aux(t)\text{‘}k = n\text{-}nodes(t)\ \#+\ k$
  apply (*induct arbitrary*: *k set*: *bt*)
   apply *simp*
  apply (*atomize*, *simp*)
  done

definition
  n-nodes-tail :: i => i **where**
  $n\text{-}nodes\text{-}tail(t) == n\text{-}nodes\text{-}aux(t)\ \text{‘}\ 0$

lemma $t \in bt(A)$ ==> $n\text{-}nodes\text{-}tail(t) = n\text{-}nodes(t)$
  by (*simp add*: *n-nodes-tail-def n-nodes-aux-eq*)

2.3 Number of leaves

consts
  n-leaves :: i => i
primrec
  $n\text{-}leaves(Lf) = 1$
  $n\text{-}leaves(Br(a,\ l,\ r)) = n\text{-}leaves(l)\ \#+\ n\text{-}leaves(r)$

lemma *n-leaves-type* [*simp*]: $t \in bt(A)$ ==> *n-leaves*(t) $\in$ *nat*
by (*induct set*: *bt*) *auto*

2.4 Reflecting trees

consts
bt-reflect :: *i* => *i*
primrec
bt-reflect(*Lf*) = *Lf*
bt-reflect(*Br*(a, l, r)) = *Br*(a, *bt-reflect*(r), *bt-reflect*(l))

lemma *bt-reflect-type* [*simp*]: $t \in bt(A)$ ==> *bt-reflect*(t) $\in bt(A)$
by (*induct set*: *bt*) *auto*

Theorems about *n-leaves*.

lemma *n-leaves-reflect*: $t \in bt(A)$ ==> *n-leaves*(*bt-reflect*(t)) = *n-leaves*(t)
by (*induct set*: *bt*) (*simp-all add*: *add-commute*)

lemma *n-leaves-nodes*: $t \in bt(A)$ ==> *n-leaves*(t) = *succ*(*n-nodes*(t))
by (*induct set*: *bt*) *simp-all*

Theorems about *bt-reflect*.

lemma *bt-reflect-bt-reflect-ident*: $t \in bt(A)$ ==> *bt-reflect*(*bt-reflect*(t)) = t
by (*induct set*: *bt*) *simp-all*

end

3 Terms over an alphabet

theory *Term* **imports** *ZF* **begin**

Illustrates the list functor (essentially the same type as in *Trees-Forest*).

consts
term :: *i* => *i*

datatype *term*(A) = *Apply* ($a \in A$, $l \in$ *list*(*term*(A)))
monos *list-mono*
type-elims *list-univ* [*THEN subsetD*, *elim-format*]

declare *Apply* [*TC*]

definition
term-rec :: [*i*, [*i*, *i*, *i*] => *i*] => *i* **where**
term-rec(t,d) ==
Vrec(t, λt g. *term-case*(λx *zs*. d(x, *zs*, *map*(λz. g‘z, *zs*)), t))

definition
term-map :: [*i* => *i*, *i*] => *i* **where**

```
  term-map(f,t) == term-rec(t, λx zs rs. Apply(f(x), rs))

definition
  term-size :: i => i  where
  term-size(t) == term-rec(t, λx zs rs. succ(list-add(rs)))

definition
  reflect :: i => i  where
  reflect(t) == term-rec(t, λx zs rs. Apply(x, rev(rs)))

definition
  preorder :: i => i  where
  preorder(t) == term-rec(t, λx zs rs. Cons(x, flat(rs)))

definition
  postorder :: i => i  where
  postorder(t) == term-rec(t, λx zs rs. flat(rs) @ [x])

lemma term-unfold: term(A) = A * list(term(A))
  by (fast intro!: term.intros [unfolded term.con-defs]
    elim: term.cases [unfolded term.con-defs])

lemma term-induct2:
  [| t ∈ term(A);
      !!x.      [| x ∈ A |] ==> P(Apply(x,Nil));
      !!x z zs. [| x ∈ A;  z ∈ term(A);  zs: list(term(A));  P(Apply(x,zs))
                |] ==> P(Apply(x, Cons(z,zs)))
     |] ==> P(t)
  — Induction on term(A) followed by induction on list.
  apply (induct-tac t)
  apply (erule list.induct)
   apply (auto dest: list-CollectD)
  done

lemma term-induct-eqn [consumes 1, case-names Apply]:
  [| t ∈ term(A);
      !!x zs. [| x ∈ A;  zs: list(term(A));  map(f,zs) = map(g,zs) |] ==>
              f(Apply(x,zs)) = g(Apply(x,zs))
   |] ==> f(t) = g(t)
  — Induction on term(A) to prove an equation.
  apply (induct-tac t)
  apply (auto dest: map-list-Collect list-CollectD)
  done
```

Lemmas to justify using *term* in other recursive type definitions.

```
lemma term-mono: A ⊆ B ==> term(A) ⊆ term(B)
  apply (unfold term.defs)
  apply (rule lfp-mono)
    apply (rule term.bnd-mono)+
```

apply (*rule univ-mono basic-monos*| *assumption*)+
done

lemma *term-univ*: *term*(*univ*(*A*)) ⊆ *univ*(*A*)
— Easily provable by induction also
apply (*unfold term.defs term.con-defs*)
apply (*rule lfp-lowerbound*)
apply (*rule-tac* [*2*] *A-subset-univ* [*THEN univ-mono*])
apply *safe*
apply (*assumption* | *rule Pair-in-univ list-univ* [*THEN subsetD*])+
done

lemma *term-subset-univ*: *A* ⊆ *univ*(*B*) ==> *term*(*A*) ⊆ *univ*(*B*)
apply (*rule subset-trans*)
apply (*erule term-mono*)
apply (*rule term-univ*)
done

lemma *term-into-univ*: [| *t* ∈ *term*(*A*); *A* ⊆ *univ*(*B*) |] ==> *t* ∈ *univ*(*B*)
by (*rule term-subset-univ* [*THEN subsetD*])

term-rec – by *Vset* recursion.

lemma *map-lemma*: [| *l* ∈ *list*(*A*); *Ord*(*i*); *rank*(*l*)<*i* |]
==> *map*(λ*z*. (λ*x* ∈ *Vset*(*i*).*h*(*x*)) ' *z*, *l*) = *map*(*h*,*l*)
— *map* works correctly on the underlying list of terms.
apply (*induct set*: *list*)
apply *simp*
apply (*subgoal-tac rank* (*a*) <*i* & *rank* (*l*) < *i*)
apply (*simp add*: *rank-of-Ord*)
apply (*simp add*: *list.con-defs*)
apply (*blast dest*: *rank-rls* [*THEN lt-trans*])
done

lemma *term-rec* [*simp*]: *ts* ∈ *list*(*A*) ==>
term-rec(*Apply*(*a*,*ts*), *d*) = *d*(*a*, *ts*, *map* (λ*z*. *term-rec*(*z*,*d*), *ts*))
— Typing premise is necessary to invoke *map-lemma*.
apply (*rule term-rec-def* [*THEN def-Vrec*, *THEN trans*])
apply (*unfold term.con-defs*)
apply (*simp add*: *rank-pair2 map-lemma*)
done

lemma *term-rec-type*:
assumes *t*: *t* ∈ *term*(*A*)
and *a*: !!*x zs r*. [| *x* ∈ *A*; *zs*: *list*(*term*(*A*));
r ∈ *list*(⋃ *t* ∈ *term*(*A*). *C*(*t*)) |]
==> *d*(*x*, *zs*, *r*): *C*(*Apply*(*x*,*zs*))
shows *term-rec*(*t*,*d*) ∈ *C*(*t*)
— Slightly odd typing condition on *r* in the second premise!
using *t*

```
  apply induct
  apply (frule list-CollectD)
  apply (subst term-rec)
   apply (assumption | rule a)+
  apply (erule list.induct)
   apply auto
  done

lemma def-term-rec:
  [| !!t. j(t)==term-rec(t,d);  ts: list(A) |] ==>
    j(Apply(a,ts)) = d(a, ts, map(λZ. j(Z), ts))
  apply (simp only:)
  apply (erule term-rec)
  done

lemma term-rec-simple-type [TC]:
  [| t ∈ term(A);
     !!x zs r. [| x ∈ A;  zs: list(term(A));  r ∈ list(C) |]
               ==> d(x, zs, r): C
   |] ==> term-rec(t,d) ∈ C
  apply (erule term-rec-type)
  apply (drule subset-refl [THEN UN-least, THEN list-mono, THEN subsetD])
  apply simp
  done
```

term-map.

```
lemma term-map [simp]:
  ts ∈ list(A) ==>
    term-map(f, Apply(a, ts)) = Apply(f(a), map(term-map(f), ts))
  by (rule term-map-def [THEN def-term-rec])

lemma term-map-type [TC]:
    [| t ∈ term(A);  !!x. x ∈ A ==> f(x): B |] ==> term-map(f,t) ∈ term(B)
  apply (unfold term-map-def)
  apply (erule term-rec-simple-type)
  apply fast
  done

lemma term-map-type2 [TC]:
    t ∈ term(A) ==> term-map(f,t) ∈ term({f(u). u ∈ A})
  apply (erule term-map-type)
  apply (erule RepFunI)
  done
```

term-size.

```
lemma term-size [simp]:
    ts ∈ list(A) ==> term-size(Apply(a, ts)) = succ(list-add(map(term-size, ts)))
  by (rule term-size-def [THEN def-term-rec])
```

lemma *term-size-type* [*TC*]: $t \in term(A) ==> term\text{-}size(t) \in nat$
by (*auto simp add*: *term-size-def*)

reflect.

lemma *reflect* [*simp*]:
$ts \in list(A) ==> reflect(Apply(a, ts)) = Apply(a, rev(map(reflect, ts)))$
by (*rule reflect-def* [*THEN def-term-rec*])

lemma *reflect-type* [*TC*]: $t \in term(A) ==> reflect(t) \in term(A)$
by (*auto simp add*: *reflect-def*)

preorder.

lemma *preorder* [*simp*]:
$ts \in list(A) ==> preorder(Apply(a, ts)) = Cons(a, flat(map(preorder, ts)))$
by (*rule preorder-def* [*THEN def-term-rec*])

lemma *preorder-type* [*TC*]: $t \in term(A) ==> preorder(t) \in list(A)$
by (*simp add*: *preorder-def*)

postorder.

lemma *postorder* [*simp*]:
$ts \in list(A) ==> postorder(Apply(a, ts)) = flat(map(postorder, ts))$ @ $[a]$
by (*rule postorder-def* [*THEN def-term-rec*])

lemma *postorder-type* [*TC*]: $t \in term(A) ==> postorder(t) \in list(A)$
by (*simp add*: *postorder-def*)

Theorems about *term-map*.

declare *map-compose* [*simp*]

lemma *term-map-ident*: $t \in term(A) ==> term\text{-}map(\lambda u.\ u, t) = t$
by (*induct rule*: *term-induct-eqn*) *simp*

lemma *term-map-compose*:
$t \in term(A) ==> term\text{-}map(f, term\text{-}map(g,t)) = term\text{-}map(\lambda u.\ f(g(u)), t)$
by (*induct rule*: *term-induct-eqn*) *simp*

lemma *term-map-reflect*:
$t \in term(A) ==> term\text{-}map(f, reflect(t)) = reflect(term\text{-}map(f,t))$
by (*induct rule*: *term-induct-eqn*) (*simp add*: *rev-map-distrib* [*symmetric*])

Theorems about *term-size*.

lemma *term-size-term-map*: $t \in term(A) ==> term\text{-}size(term\text{-}map(f,t)) = term\text{-}size(t)$
by (*induct rule*: *term-induct-eqn*) *simp*

lemma *term-size-reflect*: $t \in term(A) \Longrightarrow term\text{-}size(reflect(t)) = term\text{-}size(t)$
by (*induct rule*: *term-induct-eqn*) (*simp add*: *rev-map-distrib* [*symmetric*] *list-add-rev*)

lemma *term-size-length*: $t \in term(A) \Longrightarrow term\text{-}size(t) = length(preorder(t))$
by (*induct rule*: *term-induct-eqn*) (*simp add*: *length-flat*)

Theorems about *reflect*.

lemma *reflect-reflect-ident*: $t \in term(A) \Longrightarrow reflect(reflect(t)) = t$
by (*induct rule*: *term-induct-eqn*) (*simp add*: *rev-map-distrib*)

Theorems about preorder.

lemma *preorder-term-map*:
$t \in term(A) \Longrightarrow preorder(term\text{-}map(f,t)) = map(f, preorder(t))$
by (*induct rule*: *term-induct-eqn*) (*simp add*: *map-flat*)

lemma *preorder-reflect-eq-rev-postorder*:
$t \in term(A) \Longrightarrow preorder(reflect(t)) = rev(postorder(t))$
by (*induct rule*: *term-induct-eqn*)
(*simp add*: *rev-app-distrib rev-flat rev-map-distrib* [*symmetric*])

end

4 Datatype definition n-ary branching trees

theory *Ntree* **imports** *ZF* **begin**

Demonstrates a simple use of function space in a datatype definition. Based upon theory *Term*.

consts
ntree :: $i \Rightarrow i$
maptree :: $i \Rightarrow i$
maptree2 :: $[i, i] \Rightarrow i$

datatype $ntree(A) = Branch\ (a \in A,\ h \in (\bigcup n \in nat.\ n \to ntree(A)))$
monos *UN-mono* [*OF subset-refl Pi-mono*] — MUST have this form
type-intros *nat-fun-univ* [*THEN subsetD*]
type-elims *UN-E*

datatype $maptree(A) = Sons\ (a \in A,\ h \in maptree(A) -||> maptree(A))$
monos *FiniteFun-mono1* — Use monotonicity in BOTH args
type-intros *FiniteFun-univ1* [*THEN subsetD*]

datatype $maptree2(A, B) = Sons2\ (a \in A,\ h \in B -||> maptree2(A, B))$
monos *FiniteFun-mono* [*OF subset-refl*]
type-intros *FiniteFun-in-univ'*

definition

ntree-rec :: [[*i*, *i*, *i*] => *i*, *i*] => *i* **where**
ntree-rec(*b*) ==
Vrecursor(λ*pr*. *ntree-case*(λ*x h*. *b*(*x*, *h*, λ*i* $\in$ *domain*(*h*). *pr*‘(*h*‘*i*))))

definition
ntree-copy :: *i* => *i* **where**
ntree-copy(*z*) == *ntree-rec*(λ*x h r*. *Branch*(*x*,*r*), *z*)

ntree

lemma *ntree-unfold*: *ntree*(*A*) = *A* $\times$ ($\bigcup$ *n* $\in$ *nat*. *n* −> *ntree*(*A*))
by (*blast intro*: *ntree.intros* [*unfolded ntree.con-defs*]
elim: *ntree.cases* [*unfolded ntree.con-defs*])

lemma *ntree-induct* [*consumes 1*, *case-names Branch*, *induct set*: *ntree*]:
assumes *t*: *t* $\in$ *ntree*(*A*)
and *step*: !!*x n h*. [| *x* $\in$ *A*; *n* $\in$ *nat*; *h* $\in$ *n* −> *ntree*(*A*); $\forall$ *i* $\in$ *n*. *P*(*h*‘*i*)
|] ==> *P*(*Branch*(*x*,*h*))
shows *P*(*t*)
— A nicer induction rule than the standard one.
using *t*
apply *induct*
apply (*erule UN-E*)
apply (*assumption* | *rule step*)+
apply (*fast elim*: *fun-weaken-type*)
apply (*fast dest*: *apply-type*)
done

lemma *ntree-induct-eqn* [*consumes 1*]:
assumes *t*: *t* $\in$ *ntree*(*A*)
and *f*: *f* $\in$ *ntree*(*A*)−>*B*
and *g*: *g* $\in$ *ntree*(*A*)−>*B*
and *step*: !!*x n h*. [| *x* $\in$ *A*; *n* $\in$ *nat*; *h* $\in$ *n* −> *ntree*(*A*); *f O h* = *g O h* |]
==>
f ‘ *Branch*(*x*,*h*) = *g* ‘ *Branch*(*x*,*h*)
shows *f*‘*t*=*g*‘*t*
— Induction on *ntree*(*A*) to prove an equation
using *t*
apply *induct*
apply (*assumption* | *rule step*)+
apply (*insert f g*)
apply (*rule fun-extension*)
apply (*assumption* | *rule comp-fun*)+
apply (*simp add*: *comp-fun-apply*)
done

Lemmas to justify using *Ntree* in other recursive type definitions.

lemma *ntree-mono*: *A* $\subseteq$ *B* ==> *ntree*(*A*) $\subseteq$ *ntree*(*B*)
apply (*unfold ntree.defs*)

apply (*rule lfp-mono*)
apply (*rule ntree.bnd-mono*)+
apply (*assumption* | *rule univ-mono basic-monos*)+
done

lemma *ntree-univ*: $ntree(univ(A)) \subseteq univ(A)$
— Easily provable by induction also
apply (*unfold ntree.defs ntree.con-defs*)
apply (*rule lfp-lowerbound*)
apply (*rule-tac* [*2*] *A-subset-univ* [*THEN univ-mono*])
apply (*blast intro*: *Pair-in-univ nat-fun-univ* [*THEN subsetD*])
done

lemma *ntree-subset-univ*: $A \subseteq univ(B) ==> ntree(A) \subseteq univ(B)$
by (*rule subset-trans* [*OF ntree-mono ntree-univ*])

ntree recursion.

lemma *ntree-rec-Branch*:
$function(h) ==>$
$ntree\text{-}rec(b,\ Branch(x,h)) = b(x,\ h,\ \lambda i \in domain(h).\ ntree\text{-}rec(b,\ h`i))$
apply (*rule ntree-rec-def* [*THEN def-Vrecursor*, *THEN trans*])
apply (*simp add*: *ntree.con-defs rank-pair2* [*THEN* [*2*] *lt-trans*] *rank-apply*)
done

lemma *ntree-copy-Branch* [*simp*]:
$function(h) ==>$
$ntree\text{-}copy\ (Branch(x,\ h)) = Branch(x,\ \lambda i \in domain(h).\ ntree\text{-}copy\ (h`i))$
by (*simp add*: *ntree-copy-def ntree-rec-Branch*)

lemma *ntree-copy-is-ident*: $z \in ntree(A) ==> ntree\text{-}copy(z) = z$
by (*induct z set*: *ntree*)
(*auto simp add*: *domain-of-fun Pi-Collect-iff fun-is-function*)

maptree

lemma *maptree-unfold*: $maptree(A) = A \times (maptree(A) -||> maptree(A))$
by (*fast intro*!: *maptree.intros* [*unfolded maptree.con-defs*]
elim: *maptree.cases* [*unfolded maptree.con-defs*])

lemma *maptree-induct* [*consumes 1*, *induct set*: *maptree*]:
assumes *t*: $t \in maptree(A)$
and *step*: $!!x\ n\ h.\ [|\ x \in A;\ \ h \in maptree(A) -||> maptree(A);$
$\forall y \in field(h).\ P(y)$
$|] ==> P(Sons(x,h))$
shows $P(t)$
— A nicer induction rule than the standard one.
using *t*
apply *induct*
apply (*assumption* | *rule step*)+

apply (*erule Collect-subset* [*THEN FiniteFun-mono1*, *THEN subsetD*])
apply (*drule FiniteFun.dom-subset* [*THEN subsetD*])
apply (*drule Fin.dom-subset* [*THEN subsetD*])
apply *fast*
done

maptree2

lemma *maptree2-unfold*: *maptree2*(*A*, *B*) = *A* × (*B* −||> *maptree2*(*A*, *B*))
by (*fast intro*!: *maptree2.intros* [*unfolded maptree2.con-defs*]
elim: *maptree2.cases* [*unfolded maptree2.con-defs*])

lemma *maptree2-induct* [*consumes 1*, *induct set*: *maptree2*]:
assumes *t*: *t* ∈ *maptree2*(*A*, *B*)
and *step*: !!*x n h*. [| *x* ∈ *A*; *h* ∈ *B* −||> *maptree2*(*A*,*B*); ∀ *y* ∈ *range*(*h*). *P*(*y*)
|] ==> *P*(*Sons2*(*x*,*h*))
shows *P*(*t*)
using *t*
apply *induct*
apply (*assumption* | *rule step*)+
apply (*erule FiniteFun-mono* [*OF subset-refl Collect-subset*, *THEN subsetD*])
apply (*drule FiniteFun.dom-subset* [*THEN subsetD*])
apply (*drule Fin.dom-subset* [*THEN subsetD*])
apply *fast*
done

end

5 Trees and forests, a mutually recursive type definition

theory *Tree-Forest* **imports** *ZF* **begin**

5.1 Datatype definition

consts
tree :: *i* => *i*
forest :: *i* => *i*
tree-forest :: *i* => *i*

datatype *tree*(*A*) = *Tcons* (*a* ∈ *A*, *f* ∈ *forest*(*A*))
and *forest*(*A*) = *Fnil* | *Fcons* (*t* ∈ *tree*(*A*), *f* ∈ *forest*(*A*))

lemmas *tree'induct* =
tree-forest.mutual-induct [*THEN conjunct1*, *THEN spec*, *THEN* [*2*] *rev-mp*, *of concl*: - *t*, *consumes 1*]
and *forest'induct* =

tree-forest.mutual-induct [*THEN conjunct2*, *THEN spec*, *THEN* [*2*] *rev-mp*, *of concl*: - *f*, *consumes 1*]
for *t f*

declare *tree-forest.intros* [*simp*, *TC*]

lemma *tree-def*: *tree*(*A*) == *Part*(*tree-forest*(*A*), *Inl*)
by (*simp only*: *tree-forest.defs*)

lemma *forest-def*: *forest*(*A*) == *Part*(*tree-forest*(*A*), *Inr*)
by (*simp only*: *tree-forest.defs*)

tree-forest(*A*) as the union of *tree*(*A*) and *forest*(*A*).

lemma *tree-subset-TF*: *tree*(*A*) $\subseteq$ *tree-forest*(*A*)
apply (*unfold tree-forest.defs*)
apply (*rule Part-subset*)
done

lemma *treeI* [*TC*]: $x \in$ *tree*(*A*) ==> $x \in$ *tree-forest*(*A*)
by (*rule tree-subset-TF* [*THEN subsetD*])

lemma *forest-subset-TF*: *forest*(*A*) $\subseteq$ *tree-forest*(*A*)
apply (*unfold tree-forest.defs*)
apply (*rule Part-subset*)
done

lemma *treeI'* [*TC*]: $x \in$ *forest*(*A*) ==> $x \in$ *tree-forest*(*A*)
by (*rule forest-subset-TF* [*THEN subsetD*])

lemma *TF-equals-Un*: *tree*(*A*) $\cup$ *forest*(*A*) = *tree-forest*(*A*)
apply (*insert tree-subset-TF forest-subset-TF*)
apply (*auto intro*!: *equalityI tree-forest.intros elim*: *tree-forest.cases*)
done

lemma *tree-forest-unfold*:
tree-forest(*A*) = (*A* $\times$ *forest*(*A*)) + ({*0*} + *tree*(*A*) $\times$ *forest*(*A*))
— NOT useful, but interesting ...
supply *rews* = *tree-forest.con-defs tree-def forest-def*
apply (*unfold tree-def forest-def*)
apply (*fast intro*!: *tree-forest.intros* [*unfolded rews*, *THEN PartD1*]
elim: *tree-forest.cases* [*unfolded rews*])
done

lemma *tree-forest-unfold'*:
tree-forest(*A*) =
A $\times$ *Part*(*tree-forest*(*A*), λw. *Inr*(w)) +
{*0*} + *Part*(*tree-forest*(*A*), λw. *Inl*(w)) $*$ *Part*(*tree-forest*(*A*), λw. *Inr*(w))
by (*rule tree-forest-unfold* [*unfolded tree-def forest-def*])

lemma *tree-unfold*: *tree*(*A*) = {*Inl*(*x*). *x* ∈ *A* × *forest*(*A*)}
apply (*unfold tree-def forest-def*)
apply (*rule Part-Inl* [*THEN subst*])
apply (*rule tree-forest-unfold′* [*THEN subst-context*])
done

lemma *forest-unfold*: *forest*(*A*) = {*Inr*(*x*). *x* ∈ {*0*} + *tree*(*A*)∗*forest*(*A*)}
apply (*unfold tree-def forest-def*)
apply (*rule Part-Inr* [*THEN subst*])
apply (*rule tree-forest-unfold′* [*THEN subst-context*])
done

Type checking for recursor: Not needed; possibly interesting?

lemma *TF-rec-type*:
[| *z* ∈ *tree-forest*(*A*);
!!*x f r*. [| *x* ∈ *A*; *f* ∈ *forest*(*A*); *r* ∈ *C*(*f*)
|] ==> *b*(*x*,*f*,*r*) ∈ *C*(*Tcons*(*x*,*f*));
c ∈ *C*(*Fnil*);
!!*t f r1 r2*. [| *t* ∈ *tree*(*A*); *f* ∈ *forest*(*A*); *r1* ∈ *C*(*t*); *r2* ∈ *C*(*f*)
|] ==> *d*(*t*,*f*,*r1*,*r2*) ∈ *C*(*Fcons*(*t*,*f*))
|] ==> *tree-forest-rec*(*b*,*c*,*d*,*z*) ∈ *C*(*z*)
by (*induct-tac z*) *simp-all*

lemma *tree-forest-rec-type*:
[| !!*x f r*. [| *x* ∈ *A*; *f* ∈ *forest*(*A*); *r* ∈ *D*(*f*)
|] ==> *b*(*x*,*f*,*r*) ∈ *C*(*Tcons*(*x*,*f*));
c ∈ *D*(*Fnil*);
!!*t f r1 r2*. [| *t* ∈ *tree*(*A*); *f* ∈ *forest*(*A*); *r1* ∈ *C*(*t*); *r2* ∈ *D*(*f*)
|] ==> *d*(*t*,*f*,*r1*,*r2*) ∈ *D*(*Fcons*(*t*,*f*))
|] ==> (∀ *t* ∈ *tree*(*A*). *tree-forest-rec*(*b*,*c*,*d*,*t*) ∈ *C*(*t*)) ∧
(∀ *f* ∈ *forest*(*A*). *tree-forest-rec*(*b*,*c*,*d*,*f*) ∈ *D*(*f*))
— Mutually recursive version.
apply (*unfold Ball-def*)
apply (*rule tree-forest.mutual-induct*)
apply *simp-all*
done

5.2 Operations

consts
map :: [*i* => *i*, *i*] => *i*
size :: *i* => *i*
preorder :: *i* => *i*
list-of-TF :: *i* => *i*
of-list :: *i* => *i*
reflect :: *i* => *i*

primrec
list-of-TF (*Tcons*(*x*,*f*)) = [*Tcons*(*x*,*f*)]

list-of-TF (*Fnil*) = []
list-of-TF (*Fcons*(*t*,*tf*)) = *Cons* (*t*, *list-of-TF*(*tf*))

primrec
of-list([]) = *Fnil*
of-list(*Cons*(*t*,*l*)) = *Fcons*(*t*, *of-list*(*l*))

primrec
map (*h*, *Tcons*(*x*,*f*)) = *Tcons*(*h*(*x*), *map*(*h*,*f*))
map (*h*, *Fnil*) = *Fnil*
map (*h*, *Fcons*(*t*,*tf*)) = *Fcons* (*map*(*h*, *t*), *map*(*h*, *tf*))

primrec
size (*Tcons*(*x*,*f*)) = *succ*(*size*(*f*))
size (*Fnil*) = *0*
size (*Fcons*(*t*,*tf*)) = *size*(*t*) #+ *size*(*tf*)

primrec
preorder (*Tcons*(*x*,*f*)) = *Cons*(*x*, *preorder*(*f*))
preorder (*Fnil*) = *Nil*
preorder (*Fcons*(*t*,*tf*)) = *preorder*(*t*) @ *preorder*(*tf*)

primrec
reflect (*Tcons*(*x*,*f*)) = *Tcons*(*x*, *reflect*(*f*))
reflect (*Fnil*) = *Fnil*
reflect (*Fcons*(*t*,*tf*)) =
  of-list (*list-of-TF* (*reflect*(*tf*)) @ *Cons*(*reflect*(*t*), *Nil*))

list-of-TF and *of-list*.

lemma *list-of-TF-type* [*TC*]:
  z ∈ *tree-forest*(*A*) ==> *list-of-TF*(*z*) ∈ *list*(*tree*(*A*))
by (*induct set*: *tree-forest*) *simp-all*

lemma *of-list-type* [*TC*]: *l* ∈ *list*(*tree*(*A*)) ==> *of-list*(*l*) ∈ *forest*(*A*)
by (*induct set*: *list*) *simp-all*

map.

lemma
assumes !!*x*. *x* ∈ *A* ==> *h*(*x*): *B*
shows *map-tree-type*: *t* ∈ *tree*(*A*) ==> *map*(*h*,*t*) ∈ *tree*(*B*)
  and *map-forest-type*: *f* ∈ *forest*(*A*) ==> *map*(*h*,*f*) ∈ *forest*(*B*)
using *assms*
by (*induct rule*: *tree'induct forest'induct*) *simp-all*

size.

lemma *size-type* [*TC*]: *z* ∈ *tree-forest*(*A*) ==> *size*(*z*) ∈ *nat*
by (*induct set*: *tree-forest*) *simp-all*

preorder.

lemma *preorder-type* [*TC*]: $z \in$ *tree-forest*(A) ==> *preorder*(z) $\in$ *list*(A)
by (*induct set*: *tree-forest*) *simp-all*

Theorems about *list-of-TF* and *of-list*.

lemma *forest-induct* [*consumes 1*, *case-names Fnil Fcons*]:
[| $f \in$ *forest*(A);
R(*Fnil*);
!!t f. [| $t \in$ *tree*(A); $f \in$ *forest*(A); $R(f)$ |] ==> R(*Fcons*(t,f))
|] ==> $R(f)$
— Essentially the same as list induction.
apply (*erule tree-forest.mutual-induct*
[*THEN conjunct2*, *THEN spec*, *THEN* [*2*] *rev-mp*])
apply (*rule TrueI*)
apply *simp*
apply *simp*
done

lemma *forest-iso*: $f \in$ *forest*(A) ==> *of-list*(*list-of-TF*(f)) = f
by (*induct rule*: *forest-induct*) *simp-all*

lemma *tree-list-iso*: *ts*: *list*(*tree*(A)) ==> *list-of-TF*(*of-list*(*ts*)) = *ts*
by (*induct set*: *list*) *simp-all*

Theorems about *map*.

lemma *map-ident*: $z \in$ *tree-forest*(A) ==> *map*($\lambda u.\ u$, z) = z
by (*induct set*: *tree-forest*) *simp-all*

lemma *map-compose*:
$z \in$ *tree-forest*(A) ==> *map*(h, *map*(j,z)) = *map*($\lambda u.\ h(j(u))$, z)
by (*induct set*: *tree-forest*) *simp-all*

Theorems about *size*.

lemma *size-map*: $z \in$ *tree-forest*(A) ==> *size*(*map*(h,z)) = *size*(z)
by (*induct set*: *tree-forest*) *simp-all*

lemma *size-length*: $z \in$ *tree-forest*(A) ==> *size*(z) = *length*(*preorder*(z))
by (*induct set*: *tree-forest*) (*simp-all add*: *length-app*)

Theorems about *preorder*.

lemma *preorder-map*:
$z \in$ *tree-forest*(A) ==> *preorder*(*map*(h,z)) = *List.map*(h, *preorder*(z))
by (*induct set*: *tree-forest*) (*simp-all add*: *map-app-distrib*)

end

6 Infinite branching datatype definitions

theory *Brouwer* **imports** *ZFC* **begin**

6.1 The Brouwer ordinals

consts
brouwer :: *i*

datatype $\subseteq$ *Vfrom*(*0*, *csucc*(*nat*))
brouwer = *Zero* | *Suc* ($b \in$ *brouwer*) | *Lim* ($h \in$ *nat* −> *brouwer*)
monos *Pi-mono*
type-intros *inf-datatype-intros*

lemma *brouwer-unfold*: *brouwer* = {*0*} + *brouwer* + (*nat* −> *brouwer*)
by (*fast intro*!: *brouwer.intros* [*unfolded brouwer.con-defs*]
elim: *brouwer.cases* [*unfolded brouwer.con-defs*])

lemma *brouwer-induct2* [*consumes 1*, *case-names Zero Suc Lim*]:
assumes *b*: $b \in$ *brouwer*
and *cases*:
P(*Zero*)
!!*b*. [| $b \in$ *brouwer*; *P*(*b*) |] ==> *P*(*Suc*(*b*))
!!*h*. [| $h \in$ *nat* −> *brouwer*; $\forall i \in$ *nat*. *P*(*h*‘*i*) |] ==> *P*(*Lim*(*h*))
shows *P*(*b*)
— A nicer induction rule than the standard one.
using *b*
apply *induct*
apply (*rule cases*(*1*))
apply (*erule* (*1*) *cases*(*2*))
apply (*rule cases*(*3*))
apply (*fast elim*: *fun-weaken-type*)
apply (*fast dest*: *apply-type*)
done

6.2 The Martin-Löf wellordering type

consts
Well :: [*i*, *i* => *i*] => *i*

datatype $\subseteq$ *Vfrom*($A \cup (\bigcup x \in A.\ B(x))$, *csucc*(*nat* $\cup$ $|\bigcup x \in A.\ B(x)|$))
— The union with *nat* ensures that the cardinal is infinite.
Well(*A*, *B*) = *Sup* ($a \in A$, $f \in B(a)$ −> *Well*(*A*, *B*))
monos *Pi-mono*
type-intros *le-trans* [*OF UN-upper-cardinal le-nat-Un-cardinal*] *inf-datatype-intros*

lemma *Well-unfold*: *Well*(*A*, *B*) = ($\sum x \in A.\ B(x)$ −> *Well*(*A*, *B*))
by (*fast intro*!: *Well.intros* [*unfolded Well.con-defs*]
elim: *Well.cases* [*unfolded Well.con-defs*])

lemma *Well-induct2* [*consumes 1*, *case-names step*]:
  assumes *w*: $w \in$ *Well*(A, B)
    and *step*: !!a f. [| $a \in A$; $f \in B(a)$ −> *Well*(A,B); $\forall y \in B(a)$. $P(f\text{`}y)$ |] ==> P(*Sup*(a,f))
  shows $P(w)$
  — A nicer induction rule than the standard one.
  using *w*
  apply *induct*
  apply (*assumption* | *rule step*)+
   apply (*fast elim*: *fun-weaken-type*)
  apply (*fast dest*: *apply-type*)
  done

lemma *Well-bool-unfold*: *Well*(*bool*, $\lambda x.\ x$) = *1* + (*1* −> *Well*(*bool*, $\lambda x.\ x$))
  — In fact it's isomorphic to *nat*, but we need a recursion operator
  — for *Well* to prove this.
  apply (*rule Well-unfold* [*THEN trans*])
  apply (*simp add*: *Sigma-bool succ-def*)
  done

end

7 The Mutilated Chess Board Problem, formalized inductively

theory *Mutil* **imports** *ZF* **begin**

Originator is Max Black, according to J A Robinson. Popularized as the Mutilated Checkerboard Problem by J McCarthy.

consts
  domino :: *i*
  tiling :: *i* => *i*

inductive
  domains *domino* $\subseteq$ *Pow*(*nat* $\times$ *nat*)
  intros
    horiz: [| $i \in$ *nat*; $j \in$ *nat* |] ==> {<*i*,*j*>, <*i*,*succ*(*j*)>} $\in$ *domino*
    vertl: [| $i \in$ *nat*; $j \in$ *nat* |] ==> {<*i*,*j*>, <*succ*(*i*),*j*>} $\in$ *domino*
  type-intros *empty-subsetI cons-subsetI PowI SigmaI nat-succI*

inductive
  domains *tiling*(A) $\subseteq$ *Pow*($\bigcup$(A))
  intros
    empty: *0* $\in$ *tiling*(A)
    Un: [| $a \in A$; $t \in$ *tiling*(A); $a \cap t$ = *0* |] ==> $a \cup t \in$ *tiling*(A)
  type-intros *empty-subsetI Union-upper Un-least PowI*
  type-elims *PowD* [*elim-format*]

definition
evnodd :: [*i*, *i*] => *i* **where**
evnodd(*A*,*b*) == {*z* ∈ *A*. ∃ *i j*. *z* = <*i*,*j*> ∧ (*i* #+ *j*) *mod 2* = *b*}

7.1 Basic properties of evnodd

lemma *evnodd-iff*: <*i*,*j*>: *evnodd*(*A*,*b*) ⟷ <*i*,*j*>: *A* & (*i*#+*j*) *mod 2* = *b*
by (*unfold evnodd-def*) *blast*

lemma *evnodd-subset*: *evnodd*(*A*, *b*) ⊆ *A*
by (*unfold evnodd-def*) *blast*

lemma *Finite-evnodd*: *Finite*(*X*) ==> *Finite*(*evnodd*(*X*,*b*))
by (*rule lepoll-Finite*, *rule subset-imp-lepoll*, *rule evnodd-subset*)

lemma *evnodd-Un*: *evnodd*(*A* ∪ *B*, *b*) = *evnodd*(*A*,*b*) ∪ *evnodd*(*B*,*b*)
by (*simp add*: *evnodd-def Collect-Un*)

lemma *evnodd-Diff*: *evnodd*(*A* − *B*, *b*) = *evnodd*(*A*,*b*) − *evnodd*(*B*,*b*)
by (*simp add*: *evnodd-def Collect-Diff*)

lemma *evnodd-cons* [*simp*]:
evnodd(*cons*(<*i*,*j*>,*C*), *b*) =
(*if* (*i*#+*j*) *mod 2* = *b then cons*(<*i*,*j*>, *evnodd*(*C*,*b*)) *else evnodd*(*C*,*b*))
by (*simp add*: *evnodd-def Collect-cons*)

lemma *evnodd-0* [*simp*]: *evnodd*(*0*, *b*) = *0*
by (*simp add*: *evnodd-def*)

7.2 Dominoes

lemma *domino-Finite*: *d* ∈ *domino* ==> *Finite*(*d*)
by (*blast intro*!: *Finite-cons Finite-0 elim*: *domino.cases*)

lemma *domino-singleton*:
[| *d* ∈ *domino*; *b*<*2* |] ==> ∃ *i*′ *j*′. *evnodd*(*d*,*b*) = {<*i*′,*j*′>}
apply (*erule domino.cases*)
apply (*rule-tac* [*2*] *k1* = *i*#+*j* **in** *mod2-cases* [*THEN disjE*])
apply (*rule-tac k1* = *i*#+*j* **in** *mod2-cases* [*THEN disjE*])
apply (*rule add-type* | *assumption*)+

apply (*auto simp add*: *mod-succ succ-neq-self dest*: *ltD*)
done

7.3 Tilings

The union of two disjoint tilings is a tiling

lemma *tiling-UnI*:

```
    t ∈ tiling(A) ==> u ∈ tiling(A) ==> t ∩ u = 0 ==> t ∪ u ∈ tiling(A)
  apply (induct set: tiling)
   apply (simp add: tiling.intros)
  apply (simp add: Un-assoc subset-empty-iff [THEN iff-sym])
  apply (blast intro: tiling.intros)
  done

lemma tiling-domino-Finite: t ∈ tiling(domino) ==> Finite(t)
  apply (induct set: tiling)
   apply (rule Finite-0)
  apply (blast intro!: Finite-Un intro: domino-Finite)
  done

lemma tiling-domino-0-1: t ∈ tiling(domino) ==> |evnodd(t,0)| = |evnodd(t,1)|
  apply (induct set: tiling)
   apply (simp add: evnodd-def)
  apply (rule-tac b1 = 0 in domino-singleton [THEN exE])
    prefer 2
    apply simp
   apply assumption
  apply (rule-tac b1 = 1 in domino-singleton [THEN exE])
    prefer 2
    apply simp
   apply assumption
  apply safe
  apply (subgoal-tac ∀ p b. p ∈ evnodd (a,b) ⟶ p∉evnodd (t,b))
   apply (simp add: evnodd-Un Un-cons tiling-domino-Finite
     evnodd-subset [THEN subset-Finite] Finite-imp-cardinal-cons)
  apply (blast dest!: evnodd-subset [THEN subsetD] elim: equalityE)
  done

lemma dominoes-tile-row:
    [| i ∈ nat;  n ∈ nat |] ==> {i} ∗ (n #+ n) ∈ tiling(domino)
  apply (induct-tac n)
   apply (simp add: tiling.intros)
  apply (simp add: Un-assoc [symmetric] Sigma-succ2)
  apply (rule tiling.intros)
    prefer 2 apply assumption
   apply (rename-tac n′)
   apply (subgoal-tac
     {i}∗{succ (n′#+n′) } ∪ {i}∗{n′#+n′} =
       {<i,n′#+n′>, <i,succ (n′#+n′) >})
    prefer 2 apply blast
  apply (simp add: domino.horiz)
  apply (blast elim: mem-irrefl mem-asym)
  done

lemma dominoes-tile-matrix:
    [| m ∈ nat;  n ∈ nat |] ==> m ∗ (n #+ n) ∈ tiling(domino)
```

```
  apply (induct-tac m)
   apply (simp add: tiling.intros)
  apply (simp add: Sigma-succ1)
  apply (blast intro: tiling-UnI dominoes-tile-row elim: mem-irrefl)
  done

lemma eq-lt-E: [| x=y; x<y |] ==> P
  by auto

theorem mutil-not-tiling: [| m ∈ nat;  n ∈ nat;
         t = (succ(m)#+succ(m))*(succ(n)#+succ(n));
         t' = t − {<0,0>} − {<succ(m#+m), succ(n#+n)>} |]
      ==> t' ∉ tiling(domino)
  apply (rule notI)
  apply (drule tiling-domino-0-1)
  apply (erule-tac x = |A| for A in eq-lt-E)
  apply (subgoal-tac t ∈ tiling (domino))
   prefer 2
   apply (simp only: nat-succI add-type dominoes-tile-matrix)
  apply (simp add: evnodd-Diff mod2-add-self mod2-succ-succ
    tiling-domino-0-1 [symmetric])
  apply (rule lt-trans)
   apply (rule Finite-imp-cardinal-Diff,
     simp add: tiling-domino-Finite Finite-evnodd Finite-Diff,
     simp add: evnodd-iff nat-0-le [THEN ltD] mod2-add-self)+
  done

end


theory FoldSet imports ZF begin

consts fold-set :: [i, i, [i,i]=>i, i] => i

inductive
  domains fold-set(A, B, f,e) ⊆ Fin(A)*B
  intros
    emptyI: e∈B ==> <0, e>∈fold-set(A, B, f,e)
    consI:  [| x∈A; x ∉C;  <C,y> ∈ fold-set(A, B,f,e); f(x,y):B |]
               ==>  <cons(x,C), f(x,y)>∈fold-set(A, B, f, e)
  type-intros Fin.intros

definition
  fold :: [i, [i,i]=>i, i, i] => i  (‹fold[-]'(-,-,-')›)  where
   fold[B](f,e, A) == THE x. <A, x>∈fold-set(A, B, f,e)

definition
   setsum :: [i=>i, i] => i  where
   setsum(g, C) == if Finite(C) then
```

fold[int](%x y. g(x) $+ y, #0, C) else #0

inductive-cases *empty-fold-setE*: <*0*, *x*> ∈ *fold-set*(*A*, *B*, *f*,*e*)
inductive-cases *cons-fold-setE*: <*cons*(*x*,*C*), *y*> ∈ *fold-set*(*A*, *B*, *f*,*e*)

lemma *cons-lemma1*: [| *x*∉*C*; *x*∉*B* |] ==> *cons*(*x*,*B*)=*cons*(*x*,*C*) ⟷ *B* = *C*
by (*auto elim*: *equalityE*)

lemma *cons-lemma2*: [| *cons*(*x*, *B*)=*cons*(*y*, *C*); *x*≠*y*; *x*∉*B*; *y*∉*C* |]
  ==> *B* − {*y*} = *C*−{*x*} & *x*∈*C* & *y*∈*B*
apply (*auto elim*: *equalityE*)
done

lemma *fold-set-mono-lemma*:
  <*C*, *x*> ∈ *fold-set*(*A*, *B*, *f*, *e*)
  ==> ∀ *D*. *A*<=*D* ⟶ <*C*, *x*> ∈ *fold-set*(*D*, *B*, *f*, *e*)
apply (*erule fold-set.induct*)
apply (*auto intro*: *fold-set.intros*)
done

lemma *fold-set-mono*: *C*<=*A* ==> *fold-set*(*C*, *B*, *f*, *e*) ⊆ *fold-set*(*A*, *B*, *f*, *e*)
apply *clarify*
apply (*frule fold-set.dom-subset* [*THEN subsetD*], *clarify*)
apply (*auto dest*: *fold-set-mono-lemma*)
done

lemma *fold-set-lemma*:
  <*C*, *x*>∈*fold-set*(*A*, *B*, *f*, *e*) ==> <*C*, *x*>∈*fold-set*(*C*, *B*, *f*, *e*) & *C*<=*A*
apply (*erule fold-set.induct*)
apply (*auto intro*!: *fold-set.intros intro*: *fold-set-mono* [*THEN subsetD*])
done

lemma *Diff1-fold-set*:
  [| <*C*−{*x*},*y*> ∈ *fold-set*(*A*, *B*, *f*,*e*); *x*∈*C*; *x*∈*A*; *f*(*x*, *y*):*B* |]
  ==> <*C*, *f*(*x*, *y*)> ∈ *fold-set*(*A*, *B*, *f*, *e*)
apply (*frule fold-set.dom-subset* [*THEN subsetD*])
apply (*erule cons-Diff* [*THEN subst*], *rule fold-set.intros*, *auto*)
done

locale *fold-typing* =
 fixes *A* **and** *B* **and** *e* **and** *f*
 assumes *ftype* [*intro*,*simp*]: [|*x* ∈ *A*; *y* ∈ *B*|] ==> *f*(*x*,*y*) ∈ *B*

```
    and etype [intro,simp]:  e ∈ B
    and fcomm:  [|x ∈ A; y ∈ A; z ∈ B|] ==> f(x, f(y, z))=f(y, f(x, z))


lemma (in fold-typing) Fin-imp-fold-set:
    C∈Fin(A) ==> (∃ x. <C, x> ∈ fold-set(A, B, f,e))
apply (erule Fin-induct)
apply (auto dest: fold-set.dom-subset [THEN subsetD]
         intro: fold-set.intros etype ftype)
done

lemma Diff-sing-imp:
    [|C − {b} = D − {a}; a ≠ b; b ∈ C|] ==> C = cons(b,D) − {a}
by (blast elim: equalityE)

lemma (in fold-typing) fold-set-determ-lemma [rule-format]:
n∈nat
 ==> ∀ C. |C|<n ⟶
   (∀ x. <C, x> ∈ fold-set(A, B, f,e)⟶
         (∀ y. <C, y> ∈ fold-set(A, B, f,e) ⟶ y=x))
apply (erule nat-induct)
 apply (auto simp add: le-iff)
apply (erule fold-set.cases)
 apply (force elim!: empty-fold-setE)
apply (erule fold-set.cases)
 apply (force elim!: empty-fold-setE, clarify)

apply (frule-tac a = Ca in fold-set.dom-subset [THEN subsetD, THEN SigmaD1])
apply (frule-tac a = Cb in fold-set.dom-subset [THEN subsetD, THEN SigmaD1])
apply (simp add: Fin-into-Finite [THEN Finite-imp-cardinal-cons])
apply (case-tac x=xb, auto)
apply (simp add: cons-lemma1, blast)

case x ≠ xb

apply (drule cons-lemma2, safe)
apply (frule Diff-sing-imp, assumption+)

* LEVEL 17

apply (subgoal-tac |Ca| ≤ |Cb|)
 prefer 2
 apply (rule succ-le-imp-le)
 apply (simp add: Fin-into-Finite Finite-imp-succ-cardinal-Diff
              Fin-into-Finite [THEN Finite-imp-cardinal-cons])
apply (rule-tac C1 = Ca−{xb} in Fin-imp-fold-set [THEN exE])
 apply (blast intro: Diff-subset [THEN Fin-subset])

* LEVEL 24 *

apply (frule Diff1-fold-set, blast, blast)
apply (blast dest!: ftype fold-set.dom-subset [THEN subsetD])
```

apply (*subgoal-tac ya = f(xb,xa)*)
 prefer *2* **apply** (*blast del*: *equalityCE*)
apply (*subgoal-tac* <*Cb*−{*x*}, *xa*> ∈ *fold-set*(*A*,*B*,*f*,*e*))
 prefer *2* **apply** *simp*
apply (*subgoal-tac yb* = *f* (*x*, *xa*))
 apply (*drule-tac* [*2*] *C* = *Cb* **in** *Diff1-fold-set*, *simp-all*)
  apply (*blast intro*: *fcomm dest*!: *fold-set.dom-subset* [*THEN subsetD*])
 apply (*blast intro*: *ftype dest*!: *fold-set.dom-subset* [*THEN subsetD*], *blast*)
done

lemma (**in** *fold-typing*) *fold-set-determ*:
  [| <*C*, *x*>∈*fold-set*(*A*, *B*, *f*, *e*);
    <*C*, *y*>∈*fold-set*(*A*, *B*, *f*, *e*)|] ==> *y*=*x*
apply (*frule fold-set.dom-subset* [*THEN subsetD*], *clarify*)
apply (*drule Fin-into-Finite*)
apply (*unfold Finite-def*, *clarify*)
apply (*rule-tac n* = *succ* (*n*) **in** *fold-set-determ-lemma*)
apply (*auto intro*: *eqpoll-imp-lepoll* [*THEN lepoll-cardinal-le*])
done

lemma (**in** *fold-typing*) *fold-equality*:
  <*C*,*y*> ∈ *fold-set*(*A*,*B*,*f*,*e*) ==> *fold*[*B*](*f*,*e*,*C*) = *y*
apply (*unfold fold-def*)
apply (*frule fold-set.dom-subset* [*THEN subsetD*], *clarify*)
apply (*rule the-equality*)
 apply (*rule-tac* [*2*] *A*=*C* **in** *fold-typing.fold-set-determ*)
apply (*force dest*: *fold-set-lemma*)
apply (*auto dest*: *fold-set-lemma*)
apply (*simp add*: *fold-typing-def*, *auto*)
apply (*auto dest*: *fold-set-lemma intro*: *ftype etype fcomm*)
done

lemma *fold-0* [*simp*]: *e* ∈ *B* ==> *fold*[*B*](*f*,*e*,*0*) = *e*
apply (*unfold fold-def*)
apply (*blast elim*!: *empty-fold-setE intro*: *fold-set.intros*)
done

This result is the right-to-left direction of the subsequent result

lemma (**in** *fold-typing*) *fold-set-imp-cons*:
  [| <*C*, *y*> ∈ *fold-set*(*C*, *B*, *f*, *e*); *C* ∈ *Fin*(*A*); *c* ∈ *A*; *c*∉*C* |]
   ==> <*cons*(*c*, *C*), *f*(*c*,*y*)> ∈ *fold-set*(*cons*(*c*, *C*), *B*, *f*, *e*)
apply (*frule FinD* [*THEN fold-set-mono*, *THEN subsetD*])
 apply *assumption*
apply (*frule fold-set.dom-subset* [*of A*, *THEN subsetD*])
apply (*blast intro*!: *fold-set.consI intro*: *fold-set-mono* [*THEN subsetD*])
done

```
lemma (in fold-typing) fold-cons-lemma [rule-format]:
[| C ∈ Fin(A); c ∈ A; c∉C |]
     ==> <cons(c, C), v> ∈ fold-set(cons(c, C), B, f, e) ⟷
          (∃ y. <C, y> ∈ fold-set(C, B, f, e) & v = f(c, y))
apply auto
 prefer 2 apply (blast intro: fold-set-imp-cons)
 apply (frule-tac Fin.consI [of c, THEN FinD, THEN fold-set-mono, THEN subsetD], assumption+)
apply (frule-tac fold-set.dom-subset [of A, THEN subsetD])
apply (drule FinD)
apply (rule-tac A1 = cons(c,C) and f1=f and B1=B and C1=C and e1=e in fold-typing.Fin-imp-fold-set [THEN exE])
apply (blast intro: fold-typing.intro ftype etype fcomm)
apply (blast intro: Fin-subset [of - cons(c,C)] Finite-into-Fin
             dest: Fin-into-Finite)
apply (rule-tac x = x in exI)
apply (auto intro: fold-set.intros)
apply (drule-tac fold-set-lemma [of C], blast)
apply (blast intro!: fold-set.consI
             intro: fold-set-determ fold-set-mono [THEN subsetD]
             dest: fold-set.dom-subset [THEN subsetD])
done

lemma (in fold-typing) fold-cons:
     [| C∈Fin(A); c∈A; c∉C|]
      ==> fold[B](f, e, cons(c, C)) = f(c, fold[B](f, e, C))
apply (unfold fold-def)
apply (simp add: fold-cons-lemma)
apply (rule the-equality, auto)
 apply (subgoal-tac [2] ⟨C, y⟩ ∈ fold-set(A, B, f, e))
  apply (drule Fin-imp-fold-set)
apply (auto dest: fold-set-lemma  simp add: fold-def [symmetric] fold-equality)
apply (blast intro: fold-set-mono [THEN subsetD] dest!: FinD)
done

lemma (in fold-typing) fold-type [simp,TC]:
     C∈Fin(A) ==> fold[B](f,e,C):B
apply (erule Fin-induct)
apply (simp-all add: fold-cons ftype etype)
done

lemma (in fold-typing) fold-commute [rule-format]:
     [| C∈Fin(A); c∈A |]
      ==> (∀ y∈B. f(c, fold[B](f, y, C)) = fold[B](f, f(c, y), C))
apply (erule Fin-induct)
apply (simp-all add: fold-typing.fold-cons [of A B - f]
                     fold-typing.fold-type [of A B - f]
                     fold-typing-def fcomm)
done
```

lemma (**in** *fold-typing*) *fold-nest-Un-Int*:
  [| *C*∈*Fin*(*A*); *D*∈*Fin*(*A*) |]
   ==> *fold*[*B*](*f*, *fold*[*B*](*f*, *e*, *D*), *C*) =
     fold[*B*](*f*, *fold*[*B*](*f*, *e*, (*C* ∩ *D*)), *C* ∪ *D*)
apply (*erule Fin-induct*, *auto*)
apply (*simp add*: *Un-cons Int-cons-left fold-type fold-commute*
    fold-typing.fold-cons [*of A* - - *f*]
    fold-typing-def fcomm cons-absorb)
done

lemma (**in** *fold-typing*) *fold-nest-Un-disjoint*:
  [| *C*∈*Fin*(*A*); *D*∈*Fin*(*A*); *C* ∩ *D* = *0* |]
   ==> *fold*[*B*](*f*,*e*,*C* ∪ *D*) = *fold*[*B*](*f*, *fold*[*B*](*f*,*e*,*D*), *C*)
by (*simp add*: *fold-nest-Un-Int*)

lemma *Finite-cons-lemma*: *Finite*(*C*) ==> *C*∈*Fin*(*cons*(*c*, *C*))
apply (*drule Finite-into-Fin*)
apply (*blast intro*: *Fin-mono* [*THEN subsetD*])
done

7.4 The Operator *setsum*

lemma *setsum-0* [*simp*]: *setsum*(*g*, *0*) = *#0*
by (*simp add*: *setsum-def*)

lemma *setsum-cons* [*simp*]:
  Finite(*C*) ==>
   setsum(*g*, *cons*(*c*,*C*)) =
    (*if c* ∈ *C then setsum*(*g*,*C*) *else g*(*c*) $+ *setsum*(*g*,*C*))
apply (*auto simp add*: *setsum-def Finite-cons cons-absorb*)
apply (*rule-tac A* = *cons* (*c*, *C*) **in** *fold-typing.fold-cons*)
apply (*auto intro*: *fold-typing.intro Finite-cons-lemma*)
done

lemma *setsum-K0*: *setsum*((%*i*. *#0*), *C*) = *#0*
apply (*case-tac Finite* (*C*))
 prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule Finite-induct*, *auto*)
done

lemma *setsum-Un-Int*:
  [| *Finite*(*C*); *Finite*(*D*) |]
   ==> *setsum*(*g*, *C* ∪ *D*) $+ *setsum*(*g*, *C* ∩ *D*)
    = *setsum*(*g*, *C*) $+ *setsum*(*g*, *D*)
apply (*erule Finite-induct*)
apply (*simp-all add*: *Int-cons-right cons-absorb Un-cons Int-commute Finite-Un*
    Int-lower1 [*THEN subset-Finite*])

done

lemma *setsum-type* [*simp*,*TC*]: *setsum*(g, C):*int*
apply (*case-tac Finite* (C))
prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule Finite-induct*, *auto*)
done

lemma *setsum-Un-disjoint*:
[| *Finite*(C); *Finite*(D); $C \cap D = 0$ |]
==> *setsum*(g, $C \cup D$) = *setsum*(g, C) \$+ *setsum*(g,D)
apply (*subst setsum-Un-Int* [*symmetric*])
apply (*subgoal-tac* [*3*] *Finite* ($C \cup D$))
apply (*auto intro*: *Finite-Un*)
done

lemma *Finite-RepFun* [*rule-format* (*no-asm*)]:
Finite(I) ==> ($\forall i \in I$. *Finite*($C(i)$)) $\longrightarrow$ *Finite*(*RepFun*(I, C))
apply (*erule Finite-induct*, *auto*)
done

lemma *setsum-UN-disjoint* [*rule-format* (*no-asm*)]:
Finite(I)
==> ($\forall i \in I$. *Finite*($C(i)$)) $\longrightarrow$
($\forall i \in I$. $\forall j \in I$. $i \neq j \longrightarrow C(i) \cap C(j) = 0$) $\longrightarrow$
setsum(f, $\bigcup i \in I$. $C(i)$) = *setsum* (%i. *setsum*(f, $C(i)$), I)
apply (*erule Finite-induct*, *auto*)
apply (*subgoal-tac* $\forall i \in B$. $x \neq i$)
prefer *2* **apply** *blast*
apply (*subgoal-tac* C (x) $\cap$ ($\bigcup i \in B$. C (i)) = *0*)
prefer *2* **apply** *blast*
apply (*subgoal-tac Finite* ($\bigcup i \in B$. C (i)) & *Finite* (C (x)) & *Finite* (B))
apply (*simp* (*no-asm-simp*) *add*: *setsum-Un-disjoint*)
apply (*auto intro*: *Finite-Union Finite-RepFun*)
done

lemma *setsum-addf*: *setsum*(%x. $f(x)$ \$+ $g(x)$,C) = *setsum*(f, C) \$+ *setsum*(g, C)
apply (*case-tac Finite* (C))
prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule Finite-induct*, *auto*)
done

lemma *fold-set-cong*:
[| $A=A'$; $B=B'$; $e=e'$; ($\forall x \in A'$. $\forall y \in B'$. $f(x,y) = f'(x,y)$) |]
==> *fold-set*(A,B,f,e) = *fold-set*(A',B',f',e')
apply (*simp add*: *fold-set-def*)

```
apply (intro refl iff-refl lfp-cong Collect-cong disj-cong ex-cong, auto)
done

lemma fold-cong:
[| B=B′; A=A′; e=e′;
   !!x y. [|x∈A′; y∈B′|] ==> f(x,y) = f′(x,y) |] ==>
  fold[B](f,e,A) = fold[B′](f′, e′, A′)
apply (simp add: fold-def)
apply (subst fold-set-cong)
apply (rule-tac [5] refl, simp-all)
done

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

lemma setsum-Un:
     [| Finite(A); Finite(B) |]
      ==> setsum(f, A ∪ B) =
          setsum(f, A) $+ setsum(f, B) $− setsum(f, A ∩ B)
apply (subst setsum-Un-Int [symmetric], auto)
done


lemma setsum-zneg-or-0 [rule-format (no-asm)]:
     Finite(A) ==> (∀ x∈A. g(x) $≤ #0) ⟶ setsum(g, A) $≤ #0
apply (erule Finite-induct)
apply (auto intro: zneg-or-0-add-zneg-or-0-imp-zneg-or-0)
done

lemma setsum-succD-lemma [rule-format]:
     Finite(A)
      ==> ∀ n∈nat. setsum(f,A) = $# succ(n) ⟶ (∃ a∈A. #0 $< f(a))
apply (erule Finite-induct)
apply (auto simp del: int-of-0 int-of-succ simp add: not-zless-iff-zle int-of-0 [symmetric])
apply (subgoal-tac setsum (f, B) $≤ #0)
apply simp-all
prefer 2 apply (blast intro: setsum-zneg-or-0)
apply (subgoal-tac $# 1 $≤ f (x) $+ setsum (f, B) )
apply (drule zdiff-zle-iff [THEN iffD2])
apply (subgoal-tac $# 1 $≤ $# 1 $− setsum (f,B) )
apply (drule-tac x = $# 1 in zle-trans)
apply (rule-tac [2] j = #1 in zless-zle-trans, auto)
done

lemma setsum-succD:
     [| setsum(f, A) = $# succ(n); n∈nat |]==> ∃ a∈A. #0 $< f(a)
apply (case-tac Finite (A) )
```

apply (*blast intro*: *setsum-succD-lemma*)
apply (*unfold setsum-def*)
apply (*auto simp del*: *int-of-0 int-of-succ simp add*: *int-succ-int-1* [*symmetric*] *int-of-0* [*symmetric*])
done

lemma *g-zpos-imp-setsum-zpos* [*rule-format*]:
  $Finite(A)$ ==> $(\forall x \in A.\ \#0\ \$\leq\ g(x)) \longrightarrow \#0\ \$\leq\ setsum(g,\ A)$
apply (*erule Finite-induct*)
apply (*simp* (*no-asm*))
apply (*auto intro*: *zpos-add-zpos-imp-zpos*)
done

lemma *g-zpos-imp-setsum-zpos2* [*rule-format*]:
  [| $Finite(A)$; $\forall x.\ \#0\ \$\leq\ g(x)$ |] ==> $\#0\ \$\leq\ setsum(g,\ A)$
apply (*erule Finite-induct*)
apply (*auto intro*: *zpos-add-zpos-imp-zpos*)
done

lemma *g-zspos-imp-setsum-zspos* [*rule-format*]:
  $Finite(A)$
   ==> $(\forall x \in A.\ \#0\ \$<\ g(x)) \longrightarrow A \neq 0 \longrightarrow (\#0\ \$<\ setsum(g,\ A))$
apply (*erule Finite-induct*)
apply (*auto intro*: *zspos-add-zspos-imp-zspos*)
done

lemma *setsum-Diff* [*rule-format*]:
  $Finite(A)$ ==> $\forall a.\ M(a) = \#0 \longrightarrow setsum(M,\ A) = setsum(M,\ A - \{a\})$
apply (*erule Finite-induct*)
apply (*simp-all add*: *Diff-cons-eq Finite-Diff*)
done

end

8 The accessible part of a relation

theory *Acc* **imports** *ZF* **begin**

Inductive definition of $acc(r)$; see [3].

consts
  acc :: *i* => *i*

inductive
  domains $acc(r) \subseteq field(r)$
  intros
    vimage: [| $r-``\{a\}$: $Pow(acc(r))$; $a \in field(r)$ |] ==> $a \in acc(r)$
  monos *Pow-mono*

The introduction rule must require $a \in field(r)$, otherwise $acc(r)$ would be a proper class!

The intended introduction rule:

lemma *accI*: [| !!*b*. <*b*,*a*>:*r* ==> *b* ∈ *acc*(*r*); *a* ∈ *field*(*r*) |] ==> *a* ∈ *acc*(*r*)
  by (*blast intro*: *acc.intros*)

lemma *acc-downward*: [| *b* ∈ *acc*(*r*); <*a*,*b*>: *r* |] ==> *a* ∈ *acc*(*r*)
  by (*erule acc.cases*) *blast*

lemma *acc-induct* [*consumes 1*, *case-names vimage*, *induct set*: *acc*]:
    [| *a* ∈ *acc*(*r*);
        !!*x*. [| *x* ∈ *acc*(*r*); ∀ *y*. <*y*,*x*>:*r* ⟶ *P*(*y*) |] ==> *P*(*x*)
     |] ==> *P*(*a*)
  by (*erule acc.induct*) (*blast intro*: *acc.intros*)

lemma *wf-on-acc*: *wf*[*acc*(*r*)](*r*)
  apply (*rule wf-onI2*)
  apply (*erule acc-induct*)
  apply *fast*
  done

lemma *acc-wfI*: *field*(*r*) ⊆ *acc*(*r*) ⟹ *wf*(*r*)
  by (*erule wf-on-acc* [*THEN wf-on-subset-A*, *THEN wf-on-field-imp-wf*])

lemma *acc-wfD*: *wf*(*r*) ==> *field*(*r*) ⊆ *acc*(*r*)
  apply (*rule subsetI*)
  apply (*erule wf-induct2*, *assumption*)
   apply (*blast intro*: *accI*)+
  done

lemma *wf-acc-iff*: *wf*(*r*) ⟷ *field*(*r*) ⊆ *acc*(*r*)
  by (*rule iffI*, *erule acc-wfD*, *erule acc-wfI*)

end

theory *Multiset*
imports *FoldSet Acc*
begin

abbreviation (*input*)
  — Short cut for multiset space
  Mult :: *i*=>*i* **where**
  Mult(*A*) == *A* −||> *nat*−{*0*}

definition

  funrestrict :: [*i*,*i*] => *i* **where**
  funrestrict(*f*,*A*) == λ*x* ∈ *A*. *f*‘*x*

definition

multiset :: *i* => *o* **where**
multiset(*M*) == ∃ *A*. *M* ∈ *A* −> *nat*−{*0*} & *Finite*(*A*)

definition
mset-of :: *i*=>*i* **where**
mset-of(*M*) == *domain*(*M*)

definition
munion :: [*i*, *i*] => *i* (**infixl** ‹+#› *65*) **where**
M +# *N* == λ*x* ∈ *mset-of*(*M*) ∪ *mset-of*(*N*).
if x ∈ *mset-of*(*M*) ∩ *mset-of*(*N*) *then* (*M*‘*x*) #+ (*N*‘*x*)
else (*if x* ∈ *mset-of*(*M*) *then M*‘*x else N*‘*x*)

definition

normalize :: *i* => *i* **where**
normalize(*f*) ==
if (∃ *A*. *f* ∈ *A* −> *nat* & *Finite*(*A*)) *then*
funrestrict(*f*, {*x* ∈ *mset-of*(*f*). *0* < *f*‘*x*})
else 0

definition
mdiff :: [*i*, *i*] => *i* (**infixl** ‹−#› *65*) **where**
M −# *N* == *normalize*(λ*x* ∈ *mset-of*(*M*).
if x ∈ *mset-of*(*N*) *then M*‘*x* #− *N*‘*x else M*‘*x*)

definition

msingle :: *i* => *i* (‹{#-#}›) **where**
{#*a*#} == {<*a*, *1*>}

definition
MCollect :: [*i*, *i*=>*o*] => *i* **where**
MCollect(*M*, *P*) == *funrestrict*(*M*, {*x* ∈ *mset-of*(*M*). *P*(*x*)})

definition

mcount :: [*i*, *i*] => *i* **where**
mcount(*M*, *a*) == *if a* ∈ *mset-of*(*M*) *then M*‘*a else 0*

definition
msize :: *i* => *i* **where**
msize(*M*) == *setsum*(%*a*. $# *mcount*(*M*,*a*), *mset-of*(*M*))

abbreviation
melem :: [*i*,*i*] => *o* (‹(-/ :# -)› [*50*, *51*] *50*) **where**

a :# *M* == *a* ∈ *mset-of*(*M*)

syntax
-*MColl* :: [*pttrn*, *i*, *o*] => *i* (‹(1{# - ∈ -./ -#})›)
translations
{#*x* ∈ *M*. *P*#} == *CONST MCollect*(*M*, λ*x*. *P*)

definition

multirel1 :: [*i*,*i*]=>*i* **where**
multirel1(*A*, *r*) ==
{<*M*, *N*> ∈ *Mult*(*A*)∗*Mult*(*A*).
∃ *a* ∈ *A*. ∃ *M0* ∈ *Mult*(*A*). ∃ *K* ∈ *Mult*(*A*).
N=*M0* +# {#*a*#} & *M*=*M0* +# *K* & (∀ *b* ∈ *mset-of*(*K*). <*b*,*a*> ∈ *r*)}

definition
multirel :: [*i*, *i*] => *i* **where**
multirel(*A*, *r*) == *multirel1*(*A*, *r*)^+

definition
omultiset :: *i* => *o* **where**
omultiset(*M*) == ∃ *i*. *Ord*(*i*) & *M* ∈ *Mult*(*field*(*Memrel*(*i*)))

definition
mless :: [*i*, *i*] => *o* (**infixl** ‹<#› *50*) **where**
M <# *N* == ∃ *i*. *Ord*(*i*) & <*M*, *N*> ∈ *multirel*(*field*(*Memrel*(*i*)), *Memrel*(*i*))

definition
mle :: [*i*, *i*] => *o* (**infixl** ‹<#=› *50*) **where**
M <#= *N* == (*omultiset*(*M*) & *M* = *N*) | *M* <# *N*

8.1 Properties of the original "restrict" from ZF.thy

lemma *funrestrict-subset*: [| *f* ∈ *Pi*(*C*,*B*); *A*⊆*C* |] ==> *funrestrict*(*f*,*A*) ⊆ *f*
by (*auto simp add*: *funrestrict-def lam-def intro*: *apply-Pair*)

lemma *funrestrict-type*:
[| !!*x*. *x* ∈ *A* ==> *f*‘*x* ∈ *B*(*x*) |] ==> *funrestrict*(*f*,*A*) ∈ *Pi*(*A*,*B*)
by (*simp add*: *funrestrict-def lam-type*)

lemma *funrestrict-type2*: [| *f* ∈ *Pi*(*C*,*B*); *A*⊆*C* |] ==> *funrestrict*(*f*,*A*) ∈ *Pi*(*A*,*B*)
by (*blast intro*: *apply-type funrestrict-type*)

lemma *funrestrict* [*simp*]: *a* ∈ *A* ==> *funrestrict*(*f*,*A*) ‘ *a* = *f*‘*a*
by (*simp add*: *funrestrict-def*)

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

lemma *domain-funrestrict* [*simp*]: *domain*(*funrestrict*(*f*,*C*)) = *C*
by (*auto simp add*: *funrestrict-def lam-def*)

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

declare *domain-of-fun* [*simp*]
declare *domainE* [*rule del*]

A useful simplification rule

lemma *multiset-fun-iff*:
  (*f* ∈ *A* −> *nat*−{*0*}) ⟷ *f* ∈ *A*−>*nat*&(∀ *a* ∈ *A*. *f*'*a* ∈ *nat* & *0* < *f*'*a*)
apply *safe*
apply (*rule-tac* [*4*] *B1* = *range* (*f*) **in** *Pi-mono* [*THEN subsetD*])
apply (*auto intro*!: *Ord-0-lt*
    dest: *apply-type Diff-subset* [*THEN Pi-mono*, *THEN subsetD*]
    simp add: *range-of-fun apply-iff*)
done

lemma *multiset-into-Mult*: [| *multiset*(*M*); *mset-of*(*M*)⊆*A* |] ==> *M* ∈ *Mult*(*A*)
apply (*simp add*: *multiset-def*)
apply (*auto simp add*: *multiset-fun-iff mset-of-def*)
apply (*rule-tac B1* = *nat*−{*0*} **in** *FiniteFun-mono* [*THEN subsetD*], *simp-all*)
apply (*rule Finite-into-Fin* [*THEN* [*2*] *Fin-mono* [*THEN subsetD*], *THEN fun-FiniteFunI*])
apply (*simp-all* (*no-asm-simp*) *add*: *multiset-fun-iff*)
done

lemma *Mult-into-multiset*: *M* ∈ *Mult*(*A*) ==> *multiset*(*M*) & *mset-of*(*M*)⊆*A*
apply (*simp add*: *multiset-def mset-of-def*)
apply (*frule FiniteFun-is-fun*)
apply (*drule FiniteFun-domain-Fin*)
apply (*frule FinD*, *clarify*)
apply (*rule-tac x* = *domain* (*M*) **in** *exI*)
apply (*blast intro*: *Fin-into-Finite*)
done

lemma *Mult-iff-multiset*: *M* ∈ *Mult*(*A*) ⟷ *multiset*(*M*) & *mset-of*(*M*)⊆*A*
by (*blast dest*: *Mult-into-multiset intro*: *multiset-into-Mult*)

lemma *multiset-iff-Mult-mset-of*: *multiset*(*M*) ⟷ *M* ∈ *Mult*(*mset-of*(*M*))

by (*auto simp add*: *Mult-iff-multiset*)

The *multiset* operator

lemma *multiset-0* [*simp*]: *multiset*(*0*)
by (*auto intro*: *FiniteFun.intros simp add*: *multiset-iff-Mult-mset-of*)

The *mset-of* operator

lemma *multiset-set-of-Finite* [*simp*]: *multiset*(M) ==> *Finite*(*mset-of*(M))
by (*simp add*: *multiset-def mset-of-def*, *auto*)

lemma *mset-of-0* [*iff*]: *mset-of*(*0*) = *0*
by (*simp add*: *mset-of-def*)

lemma *mset-is-0-iff*: *multiset*(M) ==> *mset-of*(M)=*0* $\longleftrightarrow$ M=*0*
by (*auto simp add*: *multiset-def mset-of-def*)

lemma *mset-of-single* [*iff*]: *mset-of*({#*a*#}) = {*a*}
by (*simp add*: *msingle-def mset-of-def*)

lemma *mset-of-union* [*iff*]: *mset-of*(M +# N) = *mset-of*(M) $\cup$ *mset-of*(N)
by (*simp add*: *mset-of-def munion-def*)

lemma *mset-of-diff* [*simp*]: *mset-of*(M)$\subseteq$$A$ ==> *mset-of*(M −# N) $\subseteq$ A
by (*auto simp add*: *mdiff-def multiset-def normalize-def mset-of-def*)

lemma *msingle-not-0* [*iff*]: {#*a*#} $\neq$ *0* & *0* $\neq$ {#*a*#}
by (*simp add*: *msingle-def*)

lemma *msingle-eq-iff* [*iff*]: ({#*a*#} = {#*b*#}) $\longleftrightarrow$ (*a* = *b*)
by (*simp add*: *msingle-def*)

lemma *msingle-multiset* [*iff*,*TC*]: *multiset*({#*a*#})
apply (*simp add*: *multiset-def msingle-def*)
apply (*rule-tac x* = {*a*} **in** *exI*)
apply (*auto intro*: *Finite-cons Finite-0 fun-extend3*)
done

lemmas *Collect-Finite* = *Collect-subset* [*THEN subset-Finite*]

lemma *normalize-idem* [*simp*]: *normalize*(*normalize*(f)) = *normalize*(f)
apply (*simp add*: *normalize-def funrestrict-def mset-of-def*)
apply (*case-tac* $\exists$ *A*. *f* $\in$ *A* −> *nat* & *Finite* (*A*))
apply *clarify*
apply (*drule-tac x* = {*x* $\in$ *domain* (*f*) . *0* < *f* ‘ *x*} **in** *spec*)
apply *auto*

apply (*auto intro*!: *lam-type simp add*: *Collect-Finite*)
done

lemma *normalize-multiset* [*simp*]: *multiset*(M) ==> *normalize*(M) = M
by (*auto simp add*: *multiset-def normalize-def mset-of-def funrestrict-def multiset-fun-iff*)

lemma *multiset-normalize* [*simp*]: *multiset*(*normalize*(f))
apply (*simp add*: *normalize-def*)
apply (*simp add*: *normalize-def mset-of-def multiset-def*, *auto*)
apply (*rule-tac* $x = \{x \in A\ .\ 0{<}f\text{`}x\}$ **in** *exI*)
apply (*auto intro*: *Collect-subset* [*THEN subset-Finite*] *funrestrict-type*)
done

lemma *munion-multiset* [*simp*]: [| *multiset*(M); *multiset*(N) |] ==> *multiset*(M +# N)
apply (*unfold multiset-def munion-def mset-of-def*, *auto*)
apply (*rule-tac* $x = A \cup Aa$ **in** *exI*)
apply (*auto intro*!: *lam-type intro*: *Finite-Un simp add*: *multiset-fun-iff zero-less-add*)
done

lemma *mdiff-multiset* [*simp*]: *multiset*(M −# N)
by (*simp add*: *mdiff-def*)

lemma *munion-0* [*simp*]: *multiset*(M) ==> M +# *0* = M & *0* +# M = M
apply (*simp add*: *multiset-def*)
apply (*auto simp add*: *munion-def mset-of-def*)
done

lemma *munion-commute*: M +# N = N +# M
by (*auto intro*!: *lam-cong simp add*: *munion-def*)

lemma *munion-assoc*: (M +# N) +# K = M +# (N +# K)
apply (*unfold munion-def mset-of-def*)
apply (*rule lam-cong*, *auto*)
done

lemma *munion-lcommute*: M +# (N +# K) = N +# (M +# K)
apply (*unfold munion-def mset-of-def*)

apply (*rule lam-cong*, *auto*)
done

lemmas *munion-ac* = *munion-commute munion-assoc munion-lcommute*

lemma *mdiff-self-eq-0* [*simp*]: $M -\# M = 0$
by (*simp add*: *mdiff-def normalize-def mset-of-def*)

lemma *mdiff-0* [*simp*]: $0 -\# M = 0$
by (*simp add*: *mdiff-def normalize-def*)

lemma *mdiff-0-right* [*simp*]: *multiset*(M) ==> $M -\# 0 = M$
by (*auto simp add*: *multiset-def mdiff-def normalize-def multiset-fun-iff mset-of-def funrestrict-def*)

lemma *mdiff-union-inverse2* [*simp*]: *multiset*(M) ==> $M +\# \{\#a\#\} -\# \{\#a\#\} = M$
apply (*unfold multiset-def munion-def mdiff-def msingle-def normalize-def mset-of-def*)
apply (*auto cong add*: *if-cong simp add*: *ltD multiset-fun-iff funrestrict-def subset-Un-iff2* [*THEN iffD1*])
prefer *2* **apply** (*force intro*!: *lam-type*)
apply (*subgoal-tac* [*2*] $\{x \in A \cup \{a\} \,.\, x \neq a \wedge x \in A\} = A$)
apply (*rule fun-extension*, *auto*)
apply (*drule-tac* $x = A \cup \{a\}$ **in** *spec*)
apply (*simp add*: *Finite-Un*)
apply (*force intro*!: *lam-type*)
done

lemma *mcount-type* [*simp*,*TC*]: *multiset*(M) ==> *mcount*(M, a) $\in$ *nat*
by (*auto simp add*: *multiset-def mcount-def mset-of-def multiset-fun-iff*)

lemma *mcount-0* [*simp*]: *mcount*(*0*, a) = *0*
by (*simp add*: *mcount-def*)

lemma *mcount-single* [*simp*]: *mcount*($\{\#b\#\}$, a) = (*if* a=b *then 1 else 0*)
by (*simp add*: *mcount-def mset-of-def msingle-def*)

lemma *mcount-union* [*simp*]: [| *multiset*(M); *multiset*(N) |]
==> *mcount*($M +\# N$, a) = *mcount*(M, a) #+ *mcount* (N, a)
apply (*auto simp add*: *multiset-def multiset-fun-iff mcount-def munion-def mset-of-def*)
done

lemma *mcount-diff* [*simp*]:
multiset(M) ==> *mcount*($M -\# N$, a) = *mcount*(M, a) #− *mcount*(N, a)
apply (*simp add*: *multiset-def*)

apply (*auto dest*!: *not-lt-imp-le*
simp add: *mdiff-def multiset-fun-iff mcount-def normalize-def mset-of-def*)
apply (*force intro*!: *lam-type*)
apply (*force intro*!: *lam-type*)
done

lemma *mcount-elem*: [| *multiset*(M); $a \in$ *mset-of*(M) |] ==> 0 < *mcount*(M, a)
apply (*simp add*: *multiset-def*, *clarify*)
apply (*simp add*: *mcount-def mset-of-def*)
apply (*simp add*: *multiset-fun-iff*)
done

lemma *msize-0* [*simp*]: *msize*(0) = #0
by (*simp add*: *msize-def*)

lemma *msize-single* [*simp*]: *msize*({#a#}) = #1
by (*simp add*: *msize-def*)

lemma *msize-type* [*simp*,*TC*]: *msize*(M) $\in$ *int*
by (*simp add*: *msize-def*)

lemma *msize-zpositive*: *multiset*(M)==> #0 \$$\leq$ *msize*(M)
by (*auto simp add*: *msize-def intro*: *g-zpos-imp-setsum-zpos*)

lemma *msize-int-of-nat*: *multiset*(M) ==> $\exists\, n \in$ *nat*. *msize*(M)= \$# n
apply (*rule not-zneg-int-of*)
apply (*simp-all* (*no-asm-simp*) *add*: *msize-type* [*THEN znegative-iff-zless-0*] *not-zless-iff-zle msize-zpositive*)
done

lemma *not-empty-multiset-imp-exist*:
[| $M \neq 0$; *multiset*(M) |] ==> $\exists\, a \in$ *mset-of*(M). 0 < *mcount*(M, a)
apply (*simp add*: *multiset-def*)
apply (*erule not-emptyE*)
apply (*auto simp add*: *mset-of-def mcount-def multiset-fun-iff*)
apply (*blast dest*!: *fun-is-rel*)
done

lemma *msize-eq-0-iff*: *multiset*(M) ==> *msize*(M)=#0 $\longleftrightarrow$ M=0
apply (*simp add*: *msize-def*, *auto*)
apply (*rule-tac* P = *setsum* (u,v) $\neq$ #0 **for** u v **in** *swap*)
apply *blast*
apply (*drule not-empty-multiset-imp-exist*, *assumption*, *clarify*)
apply (*subgoal-tac Finite* (*mset-of* (M) $-$ {a}))
prefer *2* **apply** (*simp add*: *Finite-Diff*)
apply (*subgoal-tac setsum* (%x. \$# *mcount* (M, x), *cons* (a, *mset-of* (M) $-${a}))=#0)
prefer *2* **apply** (*simp add*: *cons-Diff*, *simp*)

apply (*subgoal-tac* $\#0 \ \$\leq$ *setsum* (%x. $\$\#$ *mcount* (M, x), *mset-of* (M) $-$ $\{a\}$))
apply (*rule-tac* [*2*] *g-zpos-imp-setsum-zpos*)
apply (*auto simp add*: *Finite-Diff not-zless-iff-zle* [*THEN iff-sym*] *znegative-iff-zless-0* [*THEN iff-sym*])
apply (*rule not-zneg-int-of* [*THEN bexE*])
apply (*auto simp del*: *int-of-0 simp add*: *int-of-add* [*symmetric*] *int-of-0* [*symmetric*])
done

lemma *setsum-mcount-Int*:
Finite(A) ==> *setsum*(%a. $\$\#$ *mcount*(N, a), $A \cap$ *mset-of*(N))
= *setsum*(%a. $\$\#$ *mcount*(N, a), A)
apply (*induct rule*: *Finite-induct*)
apply *auto*
apply (*subgoal-tac Finite* ($B \cap$ *mset-of* (N)))
prefer *2* **apply** (*blast intro*: *subset-Finite*)
apply (*auto simp add*: *mcount-def Int-cons-left*)
done

lemma *msize-union* [*simp*]:
[| *multiset*(M); *multiset*(N) |] ==> *msize*(M +# N) = *msize*(M) $+ *msize*($N$)
apply (*simp add*: *msize-def setsum-Un setsum-addf int-of-add setsum-mcount-Int*)
apply (*subst Int-commute*)
apply (*simp add*: *setsum-mcount-Int*)
done

lemma *msize-eq-succ-imp-elem*: [|*msize*(M)= $\$\#$ *succ*(n); $n \in$ *nat*|] ==> $\exists\, a$. $a \in$ *mset-of*(M)
apply (*unfold msize-def*)
apply (*blast dest*: *setsum-succD*)
done

lemma *equality-lemma*:
[| *multiset*(M); *multiset*(N); $\forall\, a$. *mcount*(M, a)=*mcount*(N, a) |]
==> *mset-of*(M)=*mset-of*(N)
apply (*simp add*: *multiset-def*)
apply (*rule sym*, *rule equalityI*)
apply (*auto simp add*: *multiset-fun-iff mcount-def mset-of-def*)
apply (*drule-tac* [!] x=x **in** *spec*)
apply (*case-tac* [*2*] $x \in Aa$, *case-tac* $x \in A$, *auto*)
done

lemma *multiset-equality*:
[| *multiset*(M); *multiset*(N) |]==> M=$N \longleftrightarrow$($\forall\, a$. *mcount*(M, a)=*mcount*(N, a))
apply *auto*
apply (*subgoal-tac mset-of* (M) = *mset-of* (N))
prefer *2* **apply** (*blast intro*: *equality-lemma*)
apply (*simp add*: *multiset-def mset-of-def*)

apply (*auto simp add*: *multiset-fun-iff*)
apply (*rule fun-extension*)
apply (*blast*, *blast*)
apply (*drule-tac x* = *x* **in** *spec*)
apply (*auto simp add*: *mcount-def mset-of-def*)
done

lemma *munion-eq-0-iff* [*simp*]: [|*multiset*(*M*); *multiset*(*N*)|]==>(*M* +# *N* =*0*) ⟷ (*M*=*0* & *N*=*0*)
by (*auto simp add*: *multiset-equality*)

lemma *empty-eq-munion-iff* [*simp*]: [|*multiset*(*M*); *multiset*(*N*)|]==>(*0*=*M* +# *N*) ⟷ (*M*=*0* & *N*=*0*)
apply (*rule iffI*, *drule sym*)
apply (*simp-all add*: *multiset-equality*)
done

lemma *munion-right-cancel* [*simp*]:
[| *multiset*(*M*); *multiset*(*N*); *multiset*(*K*) |]==>(*M* +# *K* = *N* +# *K*)⟷(*M*=*N*)
by (*auto simp add*: *multiset-equality*)

lemma *munion-left-cancel* [*simp*]:
[|*multiset*(*K*); *multiset*(*M*); *multiset*(*N*)|] ==>(*K* +# *M* = *K* +# *N*) ⟷ (*M* = *N*)
by (*auto simp add*: *multiset-equality*)

lemma *nat-add-eq-1-cases*: [| *m* ∈ *nat*; *n* ∈ *nat* |] ==> (*m* #+ *n* = *1*) ⟷ (*m*=*1* & *n*=*0*) | (*m*=*0* & *n*=*1*)
by (*induct-tac n*) *auto*

lemma *munion-is-single*:
[|*multiset*(*M*); *multiset*(*N*)|]
==> (*M* +# *N* = {#*a*#}) ⟷ (*M*={#*a*#} & *N*=*0*) | (*M* = *0* & *N* = {#*a*#})
apply (*simp* (*no-asm-simp*) *add*: *multiset-equality*)
apply *safe*
apply *simp-all*
apply (*case-tac aa*=*a*)
apply (*drule-tac* [*2*] *x* = *aa* **in** *spec*)
apply (*drule-tac x* = *a* **in** *spec*)
apply (*simp add*: *nat-add-eq-1-cases*, *simp*)
apply (*case-tac aaa*=*aa*, *simp*)
apply (*drule-tac x* = *aa* **in** *spec*)
apply (*simp add*: *nat-add-eq-1-cases*)
apply (*case-tac aaa*=*a*)
apply (*drule-tac* [*4*] *x* = *aa* **in** *spec*)
apply (*drule-tac* [*3*] *x* = *a* **in** *spec*)

apply (*drule-tac* [*2*] $x = aaa$ **in** *spec*)
apply (*drule-tac* $x = aa$ **in** *spec*)
apply (*simp-all add*: *nat-add-eq-1-cases*)
done

lemma *msingle-is-union*: [| *multiset*(*M*); *multiset*(*N*) |]
==> ({#*a*#} = *M* +# *N*) ⟷ ({#*a*#} = *M* & *N*=*0* | *M* = *0* & {#*a*#} = *N*)
apply (*subgoal-tac* ({#*a*#} = *M* +# *N*) ⟷ (*M* +# *N* = {#*a*#}))
apply (*simp* (*no-asm-simp*) *add*: *munion-is-single*)
apply *blast*
apply (*blast dest*: *sym*)
done

lemma *setsum-decr*:
Finite(*A*)
==> (∀ *M*. *multiset*(*M*) ⟶
(∀ *a* ∈ *mset-of*(*M*). *setsum*(%*z*. $# *mcount*(*M*(*a*:=*M*‘*a* #− *1*), *z*), *A*) =
(*if a* ∈ *A then setsum*(%*z*. $# *mcount*(*M*, *z*), *A*) $− #*1*
else setsum(%*z*. $# *mcount*(*M*, *z*), *A*))))
apply (*unfold multiset-def*)
apply (*erule Finite-induct*)
apply (*auto simp add*: *multiset-fun-iff*)
apply (*unfold mset-of-def mcount-def*)
apply (*case-tac x* ∈ *A*, *auto*)
apply (*subgoal-tac* $# *M* ‘ *x* $+ #−*1* = $# *M* ‘ *x* $− $# *1*)
apply (*erule ssubst*)
apply (*rule int-of-diff*, *auto*)
done

lemma *setsum-decr2*:
Finite(*A*)
==> ∀ *M*. *multiset*(*M*) ⟶ (∀ *a* ∈ *mset-of*(*M*).
setsum(%*x*. $# *mcount*(*funrestrict*(*M*, *mset-of*(*M*)−{*a*}), *x*), *A*) =
(*if a* ∈ *A then setsum*(%*x*. $# *mcount*(*M*, *x*), *A*) $− $# *M*‘*a*
else setsum(%*x*. $# *mcount*(*M*, *x*), *A*)))
apply (*simp add*: *multiset-def*)
apply (*erule Finite-induct*)
apply (*auto simp add*: *multiset-fun-iff mcount-def mset-of-def*)
done

lemma *setsum-decr3*: [| *Finite*(*A*); *multiset*(*M*); *a* ∈ *mset-of*(*M*) |]
==> *setsum*(%*x*. $# *mcount*(*funrestrict*(*M*, *mset-of*(*M*)−{*a*}), *x*), *A* − {*a*}) =
(*if a* ∈ *A then setsum*(%*x*. $# *mcount*(*M*, *x*), *A*) $− $# *M*‘*a*
else setsum(%*x*. $# *mcount*(*M*, *x*), *A*))
apply (*subgoal-tac setsum* (%*x*. $# *mcount* (*funrestrict* (*M*, *mset-of* (*M*) −{*a*}),*x*),*A*−{*a*})

= *setsum* (%*x*. $# *mcount* (*funrestrict* (*M*, *mset-of* (*M*) −{*a*}),*x*),*A*))
apply (*rule-tac* [*2*] *setsum-Diff* [*symmetric*])
apply (*rule sym*, *rule ssubst*, *blast*)
apply (*rule sym*, *drule setsum-decr2*, *auto*)
apply (*simp add*: *mcount-def mset-of-def*)
done

lemma *nat-le-1-cases*: *n* ∈ *nat* ==> *n* ≤ *1* ⟷ (*n*=*0* | *n*=*1*)
by (*auto elim*: *natE*)

lemma *succ-pred-eq-self*: [| *0*<*n*; *n* ∈ *nat* |] ==> *succ*(*n* #− *1*) = *n*
apply (*subgoal-tac 1* ≤ *n*)
apply (*drule add-diff-inverse2*, *auto*)
done

Specialized for use in the proof below.

lemma *multiset-funrestict*:
⟦∀ *a*∈*A*. *M* ‘ *a* ∈ *nat* ∧ *0* < *M* ‘ *a*; *Finite*(*A*)⟧
⟹ *multiset*(*funrestrict*(*M*, *A* − {*a*}))
apply (*simp add*: *multiset-def multiset-fun-iff*)
apply (*rule-tac x*=*A*−{*a*} **in** *exI*)
apply (*auto intro*: *Finite-Diff funrestrict-type*)
done

lemma *multiset-induct-aux*:
assumes *prem1*: !!*M a*. [| *multiset*(*M*); *a*∉*mset-of*(*M*); *P*(*M*) |] ==> *P*(*cons*(<*a*, *1*>, *M*))
and *prem2*: !!*M b*. [| *multiset*(*M*); *b* ∈ *mset-of*(*M*); *P*(*M*) |] ==> *P*(*M*(*b*:= *M*‘*b* #+ *1*))
shows
[| *n* ∈ *nat*; *P*(*0*) |]
==> (∀ *M*. *multiset*(*M*)⟶
(*setsum*(%*x*. $# *mcount*(*M*, *x*), {*x* ∈ *mset-of*(*M*). *0* < *M*‘*x*}) = $# *n*) ⟶ *P*(*M*))
apply (*erule nat-induct*, *clarify*)
apply (*frule msize-eq-0-iff*)
apply (*auto simp add*: *mset-of-def multiset-def multiset-fun-iff msize-def*)
apply (*subgoal-tac setsum* (%*x*. $# *mcount* (*M*, *x*), *A*) =$# *succ* (*x*))
apply (*drule setsum-succD*, *auto*)
apply (*case-tac 1* <*M*‘*a*)
apply (*drule-tac* [*2*] *not-lt-imp-le*)
apply (*simp-all add*: *nat-le-1-cases*)
apply (*subgoal-tac M*= (*M* (*a*:=*M*‘*a* #− *1*)) (*a*:= (*M* (*a*:=*M*‘*a* #− *1*))‘*a* #+ *1*))
apply (*rule-tac* [*2*] *A* = *A* **and** *B* = %*x*. *nat* **and** *D* = %*x*. *nat* **in** *fun-extension*)
apply (*rule-tac* [*3*] *update-type*)+
apply (*simp-all* (*no-asm-simp*))
apply (*rule-tac* [*2*] *impI*)
apply (*rule-tac* [*2*] *succ-pred-eq-self* [*symmetric*])

```
apply (simp-all (no-asm-simp))
apply (rule subst, rule sym, blast, rule prem2)
apply (simp (no-asm) add: multiset-def multiset-fun-iff)
apply (rule-tac x = A in exI)
apply (force intro: update-type)
apply (simp (no-asm-simp) add: mset-of-def mcount-def)
apply (drule-tac x = M (a := M ‘ a #− 1)  in spec)
apply (drule mp, drule-tac [2] mp, simp-all)
apply (rule-tac x = A in exI)
apply (auto intro: update-type)
apply (subgoal-tac Finite ({x ∈ cons (a, A) . x≠a⟶0<M‘x}) )
prefer 2 apply (blast intro: Collect-subset [THEN subset-Finite] Finite-cons)
apply (drule-tac A = {x ∈ cons (a, A) . x≠a⟶0<M‘x} in setsum-decr)
apply (drule-tac x = M in spec)
apply (subgoal-tac multiset (M) )
 prefer 2
 apply (simp add: multiset-def multiset-fun-iff)
 apply (rule-tac x = A in exI, force)
apply (simp-all add: mset-of-def)
apply (drule-tac psi = ∀ x ∈ A. u(x) for u in asm-rl)
apply (drule-tac x = a in bspec)
apply (simp (no-asm-simp))
apply (subgoal-tac cons (a, A) = A)
prefer 2 apply blast
apply simp
apply (subgoal-tac M=cons (<a, M‘a>, funrestrict (M, A−{a})))
 prefer 2
 apply (rule fun-cons-funrestrict-eq)
 apply (subgoal-tac cons (a, A−{a}) = A)
  apply force
  apply force
apply (rule-tac a = cons (<a, 1>, funrestrict (M, A − {a})) in ssubst)
apply simp
apply (frule multiset-funrestict, assumption)
apply (rule prem1, assumption)
apply (simp add: mset-of-def)
apply (drule-tac x = funrestrict (M, A−{a})  in spec)
apply (drule mp)
apply (rule-tac x = A−{a} in exI)
apply (auto intro: Finite-Diff funrestrict-type simp add: funrestrict)
apply (frule-tac A = A and M = M and a = a in setsum-decr3)
apply (simp (no-asm-simp) add: multiset-def multiset-fun-iff)
apply blast
apply (simp (no-asm-simp) add: mset-of-def)
apply (drule-tac b = if u then v else w for u v w in sym, simp-all)
apply (subgoal-tac {x ∈ A − {a} . 0 < funrestrict (M, A − {x}) ‘ x} = A − {a})
apply (auto intro!: setsum-cong simp add: zdiff-eq-iff zadd-commute multiset-def
multiset-fun-iff mset-of-def)
done
```

lemma *multiset-induct2*:
[| *multiset*(*M*); *P*(*0*);
(!!*M a*. [| *multiset*(*M*); *a*∉*mset-of*(*M*); *P*(*M*) |] ==> *P*(*cons*(<*a*, *1*>, *M*)));
(!!*M b*. [| *multiset*(*M*); *b* ∈ *mset-of*(*M*); *P*(*M*) |] ==> *P*(*M*(*b*:= *M*‘*b* #+ *1*)))
|]
==> *P*(*M*)
apply (*subgoal-tac* ∃ *n* ∈ *nat*. *setsum* (λ*x*. $# *mcount* (*M*, *x*), {*x* ∈ *mset-of* (*M*) . *0* < *M* ‘ *x*}) = $# *n*)
apply (*rule-tac* [*2*] *not-zneg-int-of*)
apply (*simp-all* (*no-asm-simp*) *add*: *znegative-iff-zless-0 not-zless-iff-zle*)
apply (*rule-tac* [*2*] *g-zpos-imp-setsum-zpos*)
prefer *2* **apply** (*blast intro*: *multiset-set-of-Finite Collect-subset* [*THEN subset-Finite*])
prefer *2* **apply** (*simp add*: *multiset-def multiset-fun-iff*, *clarify*)
apply (*rule multiset-induct-aux* [*rule-format*], *auto*)
done

lemma *munion-single-case1*:
[| *multiset*(*M*); *a* ∉*mset-of*(*M*) |] ==> *M* +# {#*a*#} = *cons*(<*a*, *1*>, *M*)
apply (*simp add*: *multiset-def msingle-def*)
apply (*auto simp add*: *munion-def*)
apply (*unfold mset-of-def*, *simp*)
apply (*rule fun-extension*, *rule lam-type*, *simp-all*)
apply (*auto simp add*: *multiset-fun-iff fun-extend-apply*)
apply (*drule-tac c* = *a* **and** *b* = *1* **in** *fun-extend3*)
apply (*auto simp add*: *cons-eq Un-commute* [*of* - {*a*}])
done

lemma *munion-single-case2*:
[| *multiset*(*M*); *a* ∈ *mset-of*(*M*) |] ==> *M* +# {#*a*#} = *M*(*a*:=*M*‘*a* #+ *1*)
apply (*simp add*: *multiset-def*)
apply (*auto simp add*: *munion-def multiset-fun-iff msingle-def*)
apply (*unfold mset-of-def*, *simp*)
apply (*subgoal-tac A* ∪ {*a*} = *A*)
apply (*rule fun-extension*)
apply (*auto dest*: *domain-type intro*: *lam-type update-type*)
done

lemma *multiset-induct*:
assumes *M*: *multiset*(*M*)
and *P0*: *P*(*0*)
and *step*: !!*M a*. [| *multiset*(*M*); *P*(*M*) |] ==> *P*(*M* +# {#*a*#})
shows *P*(*M*)
apply (*rule multiset-induct2* [*OF M*])
apply (*simp-all add*: *P0*)
apply (*frule-tac* [*2*] *a* = *b* **in** *munion-single-case2* [*symmetric*])

apply (*frule-tac* $a = a$ **in** *munion-single-case1* [*symmetric*])
apply (*auto intro*: *step*)
done

lemma *MCollect-multiset* [*simp*]:
  $multiset(M) ==> multiset(\{\# \ x \in M. \ P(x)\#\})$
apply (*simp add*: *MCollect-def multiset-def mset-of-def*, *clarify*)
apply (*rule-tac* $x = \{x \in A. \ P \ (x) \ \}$ **in** *exI*)
apply (*auto dest*: *CollectD1* [*THEN* [*2*] *apply-type*]
    intro: *Collect-subset* [*THEN subset-Finite*] *funrestrict-type*)
done

lemma *mset-of-MCollect* [*simp*]:
  $multiset(M) ==> mset\text{-}of(\{\# \ x \in M. \ P(x) \ \#\}) \subseteq mset\text{-}of(M)$
by (*auto simp add*: *mset-of-def MCollect-def multiset-def funrestrict-def*)

lemma *MCollect-mem-iff* [*iff*]:
  $x \in mset\text{-}of(\{\#x \in M. \ P(x)\#\}) \longleftrightarrow \ x \in mset\text{-}of(M) \ \& \ P(x)$
by (*simp add*: *MCollect-def mset-of-def*)

lemma *mcount-MCollect* [*simp*]:
  $mcount(\{\# \ x \in M. \ P(x) \ \#\}, \ a) = (if \ P(a) \ then \ mcount(M,a) \ else \ 0)$
by (*simp add*: *mcount-def MCollect-def mset-of-def*)

lemma *multiset-partition*: $multiset(M) ==> M = \{\# \ x \in M. \ P(x) \ \#\} +\# \ \{\# \ x \in M. \ \sim \ P(x) \ \#\}$
by (*simp add*: *multiset-equality*)

lemma *natify-elem-is-self* [*simp*]:
  $[| \ multiset(M); \ a \in mset\text{-}of(M) \ |] ==> natify(M`a) = M`a$
by (*auto simp add*: *multiset-def mset-of-def multiset-fun-iff*)

lemma *munion-eq-conv-diff*: $[| \ multiset(M); \ multiset(N) \ |]$
 $==> \ (M +\# \ \{\#a\#\} = N +\# \ \{\#b\#\}) \longleftrightarrow \ (M = N \ \& \ a = b \ |$
   $M = N -\# \ \{\#a\#\} +\# \ \{\#b\#\} \ \& \ N = M -\# \ \{\#b\#\} +\# \ \{\#a\#\})$
apply (*simp del*: *mcount-single add*: *multiset-equality*)
apply (*rule iffI*, *erule-tac* [*2*] *disjE*, *erule-tac* [*3*] *conjE*)
apply (*case-tac* $a=b$, *auto*)
apply (*drule-tac* $x = a$ **in** *spec*)
apply (*drule-tac* [*2*] $x = b$ **in** *spec*)
apply (*drule-tac* [*3*] $x = aa$ **in** *spec*)
apply (*drule-tac* [*4*] $x = a$ **in** *spec*, *auto*)
apply (*subgoal-tac* [!] *mcount* (N,a) :*nat*)
apply (*erule-tac* [*3*] *natE*, *erule natE*, *auto*)
done

lemma *melem-diff-single*:
multiset(*M*) ==>
  k ∈ *mset-of*(*M* −# {#*a*#}) ⟷ (*k*=*a* & *1* < *mcount*(*M*,*a*)) | (*k*≠ *a* & *k* ∈ *mset-of*(*M*))
apply (*simp add*: *multiset-def*)
apply (*simp add*: *normalize-def mset-of-def msingle-def mdiff-def mcount-def*)
apply (*auto dest*: *domain-type intro*: *zero-less-diff* [*THEN iffD1*]
            simp add: *multiset-fun-iff apply-iff*)
apply (*force intro*!: *lam-type*)
apply (*force intro*!: *lam-type*)
apply (*force intro*!: *lam-type*)
done

lemma *munion-eq-conv-exist*:
[| *M* ∈ *Mult*(*A*); *N* ∈ *Mult*(*A*) |]
  ==> (*M* +# {#*a*#} = *N* +# {#*b*#}) ⟷
      (*M*=*N* & *a*=*b* | (∃ *K* ∈ *Mult*(*A*). *M*= *K* +# {#*b*#} & *N*=*K* +# {#*a*#}))
by (*auto simp add*: *Mult-iff-multiset melem-diff-single munion-eq-conv-diff*)

8.2 Multiset Orderings

lemma *multirel1-type*: *multirel1*(*A*, *r*) ⊆ *Mult*(*A*)∗*Mult*(*A*)
by (*auto simp add*: *multirel1-def*)

lemma *multirel1-0* [*simp*]: *multirel1*(*0*, *r*) =*0*
by (*auto simp add*: *multirel1-def*)

lemma *multirel1-iff*:
 <*N*, *M*> ∈ *multirel1*(*A*, *r*) ⟷
  (∃ *a*. *a* ∈ *A* &
  (∃ *M0*. *M0* ∈ *Mult*(*A*) & (∃ *K*. *K* ∈ *Mult*(*A*) &
   M=*M0* +# {#*a*#} & *N*=*M0* +# *K* & (∀ *b* ∈ *mset-of*(*K*). <*b*,*a*> ∈ *r*))))
by (*auto simp add*: *multirel1-def Mult-iff-multiset Bex-def*)

Monotonicity of *multirel1*

lemma *multirel1-mono1*: *A*⊆*B* ==> *multirel1*(*A*, *r*)⊆*multirel1*(*B*, *r*)
apply (*auto simp add*: *multirel1-def*)
apply (*auto simp add*: *Un-subset-iff Mult-iff-multiset*)
apply (*rule-tac x* = *a* **in** *bexI*)
apply (*rule-tac x* = *M0* **in** *bexI*, *simp*)
apply (*rule-tac x* = *K* **in** *bexI*)
apply (*auto simp add*: *Mult-iff-multiset*)
done

lemma *multirel1-mono2*: *r*⊆*s* ==> *multirel1*(*A*,*r*)⊆*multirel1*(*A*, *s*)
apply (*simp add*: *multirel1-def*, *auto*)
apply (*rule-tac x* = *a* **in** *bexI*)
apply (*rule-tac x* = *M0* **in** *bexI*)

apply (*simp-all add*: *Mult-iff-multiset*)
apply (*rule-tac x* = *K* **in** *bexI*)
apply (*simp-all add*: *Mult-iff-multiset*, *auto*)
done

lemma *multirel1-mono*:
  [| *A*⊆*B*; *r*⊆*s* |] ==> *multirel1*(*A*, *r*) ⊆ *multirel1*(*B*, *s*)
apply (*rule subset-trans*)
apply (*rule multirel1-mono1*)
apply (*rule-tac* [*2*] *multirel1-mono2*, *auto*)
done

8.3 Toward the proof of well-foundedness of multirel1

lemma *not-less-0* [*iff*]: <*M*,*0*> ∉ *multirel1*(*A*, *r*)
by (*auto simp add*: *multirel1-def Mult-iff-multiset*)

lemma *less-munion*: [| <*N*, *M0* +# {#*a*#}> ∈ *multirel1*(*A*, *r*); *M0* ∈ *Mult*(*A*) |] ==>
 (∃ *M*. <*M*, *M0*> ∈ *multirel1*(*A*, *r*) & *N* = *M* +# {#*a*#}) |
 (∃ *K*. *K* ∈ *Mult*(*A*) & (∀ *b* ∈ *mset-of*(*K*). <*b*, *a*> ∈ *r*) & *N* = *M0* +# *K*)
apply (*frule multirel1-type* [*THEN subsetD*])
apply (*simp add*: *multirel1-iff*)
apply (*auto simp add*: *munion-eq-conv-exist*)
apply (*rule-tac x*=*Ka* +# *K* **in** *exI*, *auto*, *simp add*: *Mult-iff-multiset*)
apply (*simp* (*no-asm-simp*) *add*: *munion-left-cancel munion-assoc*)
apply (*auto simp add*: *munion-commute*)
done

lemma *multirel1-base*: [| *M* ∈ *Mult*(*A*); *a* ∈ *A* |] ==> <*M*, *M* +# {#*a*#}> ∈ *multirel1*(*A*, *r*)
apply (*auto simp add*: *multirel1-iff*)
apply (*simp add*: *Mult-iff-multiset*)
apply (*rule-tac x* = *a* **in** *exI*, *clarify*)
apply (*rule-tac x* = *M* **in** *exI*, *simp*)
apply (*rule-tac x* = *0* **in** *exI*, *auto*)
done

lemma *acc-0*: *acc*(*0*)=*0*
by (*auto intro*!: *equalityI dest*: *acc.dom-subset* [*THEN subsetD*])

lemma *lemma1*: [| ∀ *b* ∈ *A*. <*b*,*a*> ∈ *r* ⟶
  (∀ *M* ∈ *acc*(*multirel1*(*A*, *r*)). *M* +# {#*b*#}:*acc*(*multirel1*(*A*, *r*)));
  M0 ∈ *acc*(*multirel1*(*A*, *r*)); *a* ∈ *A*;
  ∀ *M*. <*M*,*M0*> ∈ *multirel1*(*A*, *r*) ⟶ *M* +# {#*a*#} ∈ *acc*(*multirel1*(*A*, *r*)) |]
 ==> *M0* +# {#*a*#} ∈ *acc*(*multirel1*(*A*, *r*))
apply (*subgoal-tac M0* ∈ *Mult*(*A*))
 prefer *2*
 apply (*erule acc.cases*)

apply (*erule fieldE*)
apply (*auto dest*: *multirel1-type* [*THEN subsetD*])
apply (*rule accI*)
apply (*rename-tac N*)
apply (*drule less-munion*, *blast*)
apply (*auto simp add*: *Mult-iff-multiset*)
apply (*erule-tac P* = ∀ *x* ∈ *mset-of* (*K*) . <*x*, *a*> ∈ *r* **in** *rev-mp*)
apply (*erule-tac P* = *mset-of* (*K*) ⊆*A* **in** *rev-mp*)
apply (*erule-tac M* = *K* **in** *multiset-induct*)

apply (*simp* (*no-asm-simp*))

apply (*simp add*: *Ball-def Un-subset-iff*, *clarify*)
apply (*drule-tac x* = *aa* **in** *spec*, *simp*)
apply (*subgoal-tac aa* ∈ *A*)
prefer *2* **apply** *blast*
apply (*drule-tac x* = *M0* +# *M* **and** *P* =
%*x*. *x* ∈ *acc*(*multirel1*(*A*, *r*)) ⟶ *Q*(*x*) **for** *Q* **in** *spec*)
apply (*simp add*: *munion-assoc* [*symmetric*])

apply (*auto intro*!: *multirel1-base* [*THEN fieldI2*] *simp add*: *Mult-iff-multiset*)
done

lemma *lemma2*: [| ∀ *b* ∈ *A*. <*b*,*a*> ∈ *r*
⟶ (∀ *M* ∈ *acc*(*multirel1*(*A*, *r*)). *M* +# {#*b*#} :*acc*(*multirel1*(*A*, *r*)));
M ∈ *acc*(*multirel1*(*A*, *r*)); *a* ∈ *A*|] ==> *M* +# {#*a*#} ∈ *acc*(*multirel1*(*A*,
r))
apply (*erule acc-induct*)
apply (*blast intro*: *lemma1*)
done

lemma *lemma3*: [| *wf*[*A*](*r*); *a* ∈ *A* |]
==> ∀ *M* ∈ *acc*(*multirel1*(*A*, *r*)). *M* +# {#*a*#} ∈ *acc*(*multirel1*(*A*, *r*))
apply (*erule-tac a* = *a* **in** *wf-on-induct*, *blast*)
apply (*blast intro*: *lemma2*)
done

lemma *lemma4*: *multiset*(*M*) ==> *mset-of*(*M*)⊆*A* ⟶
wf[*A*](*r*) ⟶ *M* ∈ *field*(*multirel1*(*A*, *r*)) ⟶ *M* ∈ *acc*(*multirel1*(*A*, *r*))
apply (*erule multiset-induct*)

apply *clarify*
apply (*rule accI*, *force*)
apply (*simp add*: *multirel1-def*)

apply *clarify*
apply *simp*
apply (*subgoal-tac mset-of* (*M*) ⊆*A*)

prefer *2* **apply** *blast*
apply *clarify*
apply (*drule-tac a* = *a* **in** *lemma3*, *blast*)
apply (*subgoal-tac M* ∈ *field* (*multirel1* (*A*,*r*)))
apply *blast*
apply (*rule multirel1-base* [*THEN fieldI1*])
apply (*auto simp add*: *Mult-iff-multiset*)
done

lemma *all-accessible*: [| *wf*[*A*](*r*); *M* ∈ *Mult*(*A*); *A* ≠ *0*|] ==> *M* ∈ *acc*(*multirel1*(*A*, *r*))
apply (*erule not-emptyE*)
apply (*rule lemma4* [*THEN mp*, *THEN mp*, *THEN mp*])
apply (*rule-tac* [*4*] *multirel1-base* [*THEN fieldI1*])
apply (*auto simp add*: *Mult-iff-multiset*)
done

lemma *wf-on-multirel1*: *wf*[*A*](*r*) ==> *wf*[*A*−||>*nat*−{*0*}](*multirel1*(*A*, *r*))
apply (*case-tac A*=*0*)
apply (*simp* (*no-asm-simp*))
apply (*rule wf-imp-wf-on*)
apply (*rule wf-on-field-imp-wf*)
apply (*simp* (*no-asm-simp*) *add*: *wf-on-0*)
apply (*rule-tac A* = *acc* (*multirel1* (*A*,*r*)) **in** *wf-on-subset-A*)
apply (*rule wf-on-acc*)
apply (*blast intro*: *all-accessible*)
done

lemma *wf-multirel1*: *wf*(*r*) ==>*wf*(*multirel1*(*field*(*r*), *r*))
apply (*simp* (*no-asm-use*) *add*: *wf-iff-wf-on-field*)
apply (*drule wf-on-multirel1*)
apply (*rule-tac A* = *field* (*r*) −||> *nat* − {*0*} **in** *wf-on-subset-A*)
apply (*simp* (*no-asm-simp*))
apply (*rule field-rel-subset*)
apply (*rule multirel1-type*)
done

lemma *multirel-type*: *multirel*(*A*, *r*) ⊆ *Mult*(*A*)∗*Mult*(*A*)
apply (*simp add*: *multirel-def*)
apply (*rule trancl-type* [*THEN subset-trans*])
apply (*auto dest*: *multirel1-type* [*THEN subsetD*])
done

lemma *multirel-mono*:
[| *A*⊆*B*; *r*⊆*s* |] ==> *multirel*(*A*, *r*)⊆*multirel*(*B*,*s*)
apply (*simp add*: *multirel-def*)

apply (*rule trancl-mono*)
apply (*rule multirel1-mono*, *auto*)
done

lemma *add-diff-eq*: *k* ∈ *nat* ==> *0* < *k* ⟶ *n* #+ *k* #− *1* = *n* #+ (*k* #− *1*)
by (*erule nat-induct*, *auto*)

lemma *mdiff-union-single-conv*: [|*a* ∈ *mset-of*(*J*); *multiset*(*I*); *multiset*(*J*) |]
==> *I* +# *J* −# {#*a*#} = *I* +# (*J*−# {#*a*#})
apply (*simp* (*no-asm-simp*) *add*: *multiset-equality*)
apply (*case-tac a* ∉ *mset-of* (*I*))
apply (*auto simp add*: *mcount-def mset-of-def multiset-def multiset-fun-iff*)
apply (*auto dest*: *domain-type simp add*: *add-diff-eq*)
done

lemma *diff-add-commute*: [| *n* ≤ *m*; *m* ∈ *nat*; *n* ∈ *nat*; *k* ∈ *nat* |] ==> *m* #− *n* #+ *k* = *m* #+ *k* #− *n*
by (*auto simp add*: *le-iff less-iff-succ-add*)

lemma *multirel-implies-one-step*:
<*M*,*N*> ∈ *multirel*(*A*, *r*) ==>
trans[*A*](*r*) ⟶
(∃ *I J K*.
I ∈ *Mult*(*A*) & *J* ∈ *Mult*(*A*) & *K* ∈ *Mult*(*A*) &
N = *I* +# *J* & *M* = *I* +# *K* & *J* ≠ *0* &
(∀ *k* ∈ *mset-of*(*K*). ∃ *j* ∈ *mset-of*(*J*). <*k*,*j*> ∈ *r*))
apply (*simp add*: *multirel-def Ball-def Bex-def*)
apply (*erule converse-trancl-induct*)
apply (*simp-all add*: *multirel1-iff Mult-iff-multiset*)

apply *clarify*
apply (*rule-tac x* = *M0* **in** *exI*, *force*)

apply *clarify*
apply *hypsubst-thin*
apply (*case-tac a* ∈ *mset-of* (*Ka*))
apply (*rule-tac x* = *I* **in** *exI*, *simp* (*no-asm-simp*))
apply (*rule-tac x* = *J* **in** *exI*, *simp* (*no-asm-simp*))
apply (*rule-tac x* = (*Ka* −# {#*a*#}) +# *K* **in** *exI*, *simp* (*no-asm-simp*))
apply (*simp-all add*: *Un-subset-iff*)
apply (*simp* (*no-asm-simp*) *add*: *munion-assoc* [*symmetric*])
apply (*drule-tac t* = %*M*. *M*−#{#*a*#} **in** *subst-context*)
apply (*simp add*: *mdiff-union-single-conv melem-diff-single*, *clarify*)
apply (*erule disjE*, *simp*)

apply (*erule disjE*, *simp*)
apply (*drule-tac* $x = a$ **and** $P = \%x.\ x :\#\ Ka \longrightarrow Q(x)$ **for** Q **in** *spec*)
apply *clarify*
apply (*rule-tac* $x = xa$ **in** *exI*)
apply (*simp* (*no-asm-simp*))
apply (*blast dest*: *trans-onD*)

apply (*subgoal-tac* $a :\#\ I$)
apply (*rule-tac* $x = I-\#\{\#a\#\}$ **in** *exI*, *simp* (*no-asm-simp*))
apply (*rule-tac* $x = J+\#\{\#a\#\}$ **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *Un-subset-iff*)
apply (*rule-tac* $x = Ka +\#\ K$ **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *Un-subset-iff*)
apply (*rule conjI*)
apply (*simp* (*no-asm-simp*) *add*: *multiset-equality mcount-elem* [*THEN succ-pred-eq-self*])
apply (*rule conjI*)
apply (*drule-tac* $t = \%M.\ M-\#\{\#a\#\}$ **in** *subst-context*)
apply (*simp add*: *mdiff-union-inverse2*)
apply (*simp-all* (*no-asm-simp*) *add*: *multiset-equality*)
apply (*rule diff-add-commute* [*symmetric*])
apply (*auto intro*: *mcount-elem*)
apply (*subgoal-tac* $a \in$ *mset-of* $(I +\#\ Ka)$)
apply (*drule-tac* [*2*] *sym*, *auto*)
done

lemma *melem-imp-eq-diff-union* [*simp*]: $[\mid a \in mset\text{-}of(M);\ multiset(M)\ \mid] ==>$
$M -\#\ \{\#a\#\} +\#\ \{\#a\#\} = M$
by (*simp add*: *multiset-equality mcount-elem* [*THEN succ-pred-eq-self*])

lemma *msize-eq-succ-imp-eq-union*:
$[\mid msize(M)=\$\#\ succ(n);\ M \in Mult(A);\ n \in nat\ \mid]$
$==> \exists\, a\ N.\ M = N +\#\ \{\#a\#\}\ \&\ N \in Mult(A)\ \&\ a \in A$
apply (*drule msize-eq-succ-imp-elem*, *auto*)
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = M -\#\ \{\#a\#\}$ **in** *exI*)
apply (*frule Mult-into-multiset*)
apply (*simp* (*no-asm-simp*))
apply (*auto simp add*: *Mult-iff-multiset*)
done

lemma *one-step-implies-multirel-lemma* [*rule-format* (*no-asm*)]:
$n \in nat ==>$
$(\forall\, I\ J\ K.$
$I \in Mult(A)\ \&\ J \in Mult(A)\ \&\ K \in Mult(A)\ \&$
$(msize(J) = \$\#\ n\ \&\ J \neq 0\ \&\ \ (\forall\, k \in mset\text{-}of(K).\ \ \exists\, j \in mset\text{-}of(J).\ <k, j> \in$
$r))$
$\longrightarrow <I +\#\ K,\ I +\#\ J> \in multirel(A, r))$

apply (*simp add*: *Mult-iff-multiset*)
apply (*erule nat-induct*, *clarify*)
apply (*drule-tac M* = *J* **in** *msize-eq-0-iff*, *auto*)

apply (*subgoal-tac msize* (*J*) =$# *succ* (*x*))
prefer *2* **apply** *simp*
apply (*frule-tac A* = *A* **in** *msize-eq-succ-imp-eq-union*)
apply (*simp-all add*: *Mult-iff-multiset*, *clarify*)
apply (*rename-tac J′*, *simp*)
apply (*case-tac J′* = *0*)
apply (*simp add*: *multirel-def*)
apply (*rule r-into-trancl*, *clarify*)
apply (*simp add*: *multirel1-iff Mult-iff-multiset*, *force*)

apply (*drule sym*, *rotate-tac* −*1*, *simp*)
apply (*erule-tac V* = $# *x* = *msize* (*J′*) **in** *thin-rl*)
apply (*frule-tac M* = *K* **and** *P* = %*x*. <*x*,*a*> ∈ *r* **in** *multiset-partition*)
apply (*erule-tac P* = ∀ *k* ∈ *mset-of* (*K*) . *P*(*k*) **for** *P* **in** *rev-mp*)
apply (*erule ssubst*)
apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac* < (*I* +# {# *x* ∈ *K*. <*x*, *a*> ∈ *r*#}) +# {# *x* ∈ *K*. <*x*, *a*> ∉ *r*#}, (*I* +# {# *x* ∈ *K*. <*x*, *a*> ∈ *r*#}) +# *J′*> ∈ *multirel*(*A*, *r*))
prefer *2*
apply (*drule-tac x* = *I* +# {# *x* ∈ *K*. <*x*, *a*> ∈ *r*#} **in** *spec*)
apply (*rotate-tac* −*1*)
apply (*drule-tac x* = *J′* **in** *spec*)
apply (*rotate-tac* −*1*)
apply (*drule-tac x* = {# *x* ∈ *K*. <*x*, *a*> ∉ *r*#} **in** *spec*, *simp*) **apply** *blast*
apply (*simp add*: *munion-assoc* [*symmetric*] *multirel-def*)
apply (*rule-tac b* = *I* +# {# *x* ∈ *K*. <*x*, *a*> ∈ *r*#} +# *J′* **in** *trancl-trans*, *blast*)
apply (*rule r-into-trancl*)
apply (*simp add*: *multirel1-iff Mult-iff-multiset*)
apply (*rule-tac x* = *a* **in** *exI*)
apply (*simp* (*no-asm-simp*))
apply (*rule-tac x* = *I* +# *J′* **in** *exI*)
apply (*auto simp add*: *munion-ac Un-subset-iff*)
done

lemma *one-step-implies-multirel*:
[| *J* ≠ *0*; ∀ *k* ∈ *mset-of*(*K*). ∃ *j* ∈ *mset-of*(*J*). <*k*,*j*> ∈ *r*;
I ∈ *Mult*(*A*); *J* ∈ *Mult*(*A*); *K* ∈ *Mult*(*A*) |]
==> <*I*+#*K*, *I*+#*J*> ∈ *multirel*(*A*, *r*)
apply (*subgoal-tac multiset* (*J*))
prefer *2* **apply** (*simp add*: *Mult-iff-multiset*)
apply (*frule-tac M* = *J* **in** *msize-int-of-nat*)
apply (*auto intro*: *one-step-implies-multirel-lemma*)
done

lemma *multirel-irrefl-lemma*:
Finite(*A*) ==> *part-ord*(*A*, *r*) ⟶ (∀ *x* ∈ *A*. ∃ *y* ∈ *A*. <*x*,*y*> ∈ *r*) ⟶*A*=*0*
apply (*erule Finite-induct*)
apply (*auto dest*: *subset-consI* [*THEN* [*2*] *part-ord-subset*])
apply (*auto simp add*: *part-ord-def irrefl-def*)
apply (*drule-tac x* = *xa* **in** *bspec*)
apply (*drule-tac* [*2*] *a* = *xa* **and** *b* = *x* **in** *trans-onD*, *auto*)
done

lemma *irrefl-on-multirel*:
part-ord(*A*, *r*) ==> *irrefl*(*Mult*(*A*), *multirel*(*A*, *r*))
apply (*simp add*: *irrefl-def*)
apply (*subgoal-tac trans*[*A*](*r*))
prefer *2* **apply** (*simp add*: *part-ord-def*, *clarify*)
apply (*drule multirel-implies-one-step*, *clarify*)
apply (*simp add*: *Mult-iff-multiset*, *clarify*)
apply (*subgoal-tac Finite* (*mset-of* (*K*)))
apply (*frule-tac r* = *r* **in** *multirel-irrefl-lemma*)
apply (*frule-tac B* = *mset-of* (*K*) **in** *part-ord-subset*)
apply *simp-all*
apply (*auto simp add*: *multiset-def mset-of-def*)
done

lemma *trans-on-multirel*: *trans*[*Mult*(*A*)](*multirel*(*A*, *r*))
apply (*simp add*: *multirel-def trans-on-def*)
apply (*blast intro*: *trancl-trans*)
done

lemma *multirel-trans*:
[| <*M*, *N*> ∈ *multirel*(*A*, *r*); <*N*, *K*> ∈ *multirel*(*A*, *r*) |] ==> <*M*, *K*> ∈ *multirel*(*A*,*r*)
apply (*simp add*: *multirel-def*)
apply (*blast intro*: *trancl-trans*)
done

lemma *trans-multirel*: *trans*(*multirel*(*A*,*r*))
apply (*simp add*: *multirel-def*)
apply (*rule trans-trancl*)
done

lemma *part-ord-multirel*: *part-ord*(*A*,*r*) ==> *part-ord*(*Mult*(*A*), *multirel*(*A*, *r*))
apply (*simp* (*no-asm*) *add*: *part-ord-def*)
apply (*blast intro*: *irrefl-on-multirel trans-on-multirel*)
done

lemma *munion-multirel1-mono*:
[|<M,N> ∈ multirel1(A, r); K ∈ Mult(A) |] ==> <K +# M, K +# N> ∈ multirel1(A, r)
apply (*frule multirel1-type* [*THEN subsetD*])
apply (*auto simp add*: *multirel1-iff Mult-iff-multiset*)
apply (*rule-tac x* = *a* **in** *exI*)
apply (*simp* (*no-asm-simp*))
apply (*rule-tac x* = *K*+#*M0* **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *Un-subset-iff*)
apply (*rule-tac x* = *Ka* **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *munion-assoc*)
done

lemma *munion-multirel-mono2*:
[| <M, N> ∈ multirel(A, r); K ∈ Mult(A) |]==><K +# M, K +# N> ∈ multirel(A, r)
apply (*frule multirel-type* [*THEN subsetD*])
apply (*simp* (*no-asm-use*) *add*: *multirel-def*)
apply *clarify*
apply (*drule-tac psi* = <M,N> ∈ *multirel1* (A, r) ^+ **in** *asm-rl*)
apply (*erule rev-mp*)
apply (*erule rev-mp*)
apply (*erule rev-mp*)
apply (*erule trancl-induct*, *clarify*)
apply (*blast intro*: *munion-multirel1-mono r-into-trancl*, *clarify*)
apply (*subgoal-tac y* ∈ *Mult*(A))
prefer *2*
apply (*blast dest*: *multirel-type* [*unfolded multirel-def*, *THEN subsetD*])
apply (*subgoal-tac* <K +# y, K +# z> ∈ *multirel1* (A, r))
prefer *2* **apply** (*blast intro*: *munion-multirel1-mono*)
apply (*blast intro*: *r-into-trancl trancl-trans*)
done

lemma *munion-multirel-mono1*:
[|<M, N> ∈ multirel(A, r); K ∈ Mult(A)|] ==> <M +# K, N +# K> ∈ multirel(A, r)
apply (*frule multirel-type* [*THEN subsetD*])
apply (*rule-tac P* = %x. <x,u> ∈ *multirel*(A, r) **for** *u* **in** *munion-commute* [*THEN subst*])
apply (*subst munion-commute* [*of N*])
apply (*rule munion-multirel-mono2*)
apply (*auto simp add*: *Mult-iff-multiset*)
done

lemma *munion-multirel-mono*:
[|<M,K> ∈ multirel(A, r); <N,L> ∈ multirel(A, r)|]
==> <M +# N, K +# L> ∈ multirel(A, r)
apply (*subgoal-tac M* ∈ *Mult*(A) & *N* ∈ *Mult*(A) & *K* ∈ *Mult*(A) & *L* ∈ *Mult*(A)

)
prefer *2* **apply** (*blast dest*: *multirel-type* [*THEN subsetD*])
apply (*blast intro*: *munion-multirel-mono1 multirel-trans munion-multirel-mono2*)
done

8.4 Ordinal Multisets

lemmas *field-Memrel-mono* = *Memrel-mono* [*THEN field-mono*]

lemmas *multirel-Memrel-mono* = *multirel-mono* [*OF field-Memrel-mono Memrel-mono*]

lemma *omultiset-is-multiset* [*simp*]: *omultiset*(M) ==> *multiset*(M)
apply (*simp add*: *omultiset-def*)
apply (*auto simp add*: *Mult-iff-multiset*)
done

lemma *munion-omultiset* [*simp*]: [| *omultiset*(M); *omultiset*(N) |] ==> *omultiset*(M +# N)
apply (*simp add*: *omultiset-def*, *clarify*)
apply (*rule-tac* $x = i \cup ia$ **in** *exI*)
apply (*simp add*: *Mult-iff-multiset Ord-Un Un-subset-iff*)
apply (*blast intro*: *field-Memrel-mono*)
done

lemma *mdiff-omultiset* [*simp*]: *omultiset*(M) ==> *omultiset*(M −# N)
apply (*simp add*: *omultiset-def*, *clarify*)
apply (*simp add*: *Mult-iff-multiset*)
apply (*rule-tac* $x = i$ **in** *exI*)
apply (*simp* (*no-asm-simp*))
done

lemma *irrefl-Memrel*: *Ord*(i) ==> *irrefl*(*field*(*Memrel*(i)), *Memrel*(i))
apply (*rule irreflI*, *clarify*)
apply (*subgoal-tac Ord* (x))
prefer *2* **apply** (*blast intro*: *Ord-in-Ord*)
apply (*drule-tac* $i = x$ **in** *ltI* [*THEN lt-irrefl*], *auto*)
done

lemma *trans-iff-trans-on*: *trans*(r) $\longleftrightarrow$ *trans*[*field*(r)](r)
by (*simp add*: *trans-on-def trans-def*, *auto*)

lemma *part-ord-Memrel*: *Ord*(i) ==>*part-ord*(*field*(*Memrel*(i)), *Memrel*(i))
apply (*simp add*: *part-ord-def*)
apply (*simp* (*no-asm*) *add*: *trans-iff-trans-on* [*THEN iff-sym*])

apply (*blast intro*: *trans-Memrel irrefl-Memrel*)
done

lemmas *part-ord-mless* = *part-ord-Memrel* [*THEN part-ord-multirel*]

lemma *mless-not-refl*: ~(M <# M)
apply (*simp add*: *mless-def*, *clarify*)
apply (*frule multirel-type* [*THEN subsetD*])
apply (*drule part-ord-mless*)
apply (*simp add*: *part-ord-def irrefl-def*)
done

lemmas *mless-irrefl* = *mless-not-refl* [*THEN notE*, *elim*!]

lemma *mless-trans*: [| K <# M; M <# N |] ==> K <# N
apply (*simp add*: *mless-def*, *clarify*)
apply (*rule-tac* $x = i \cup ia$ **in** *exI*)
apply (*blast dest*: *multirel-Memrel-mono* [*OF Un-upper1 Un-upper1*, *THEN subsetD*]
multirel-Memrel-mono [*OF Un-upper2 Un-upper2*, *THEN subsetD*]
intro: *multirel-trans Ord-Un*)
done

lemma *mless-not-sym*: M <# N ==> ~ N <# M
apply *clarify*
apply (*rule mless-not-refl* [*THEN notE*])
apply (*erule mless-trans*, *assumption*)
done

lemma *mless-asym*: [| M <# N; ~P ==> N <# M |] ==> P
by (*blast dest*: *mless-not-sym*)

lemma *mle-refl* [*simp*]: *omultiset*(M) ==> M <#= M
by (*simp add*: *mle-def*)

lemma *mle-antisym*:
[| M <#= N; N <#= M |] ==> $M = N$
apply (*simp add*: *mle-def*)
apply (*blast dest*: *mless-not-sym*)
done

lemma *mle-trans*: [| K <#= M; M <#= N |] ==> K <#= N
apply (*simp add*: *mle-def*)
apply (*blast intro*: *mless-trans*)
done

lemma *mless-le-iff*: M <# N ⟷ (M <#= N & M ≠ N)
by (*simp add*: *mle-def*, *auto*)

lemma *munion-less-mono2*: [| M <# N; *omultiset*(K) |] ==> K +# M <# K +# N
apply (*simp add*: *mless-def omultiset-def*, *clarify*)
apply (*rule-tac x* = *i* ∪ *ia* **in** *exI*)
apply (*simp add*: *Mult-iff-multiset Ord-Un Un-subset-iff*)
apply (*rule munion-multirel-mono2*)
apply (*blast intro*: *multirel-Memrel-mono* [*THEN subsetD*])
apply (*simp add*: *Mult-iff-multiset*)
apply (*blast intro*: *field-Memrel-mono* [*THEN subsetD*])
done

lemma *munion-less-mono1*: [| M <# N; *omultiset*(K) |] ==> M +# K <# N +# K
by (*force dest*: *munion-less-mono2 simp add*: *munion-commute*)

lemma *mless-imp-omultiset*: M <# N ==> *omultiset*(M) & *omultiset*(N)
by (*auto simp add*: *mless-def omultiset-def dest*: *multirel-type* [*THEN subsetD*])

lemma *munion-less-mono*: [| M <# K; N <# L |] ==> M +# N <# K +# L
apply (*frule-tac M* = *M* **in** *mless-imp-omultiset*)
apply (*frule-tac M* = *N* **in** *mless-imp-omultiset*)
apply (*blast intro*: *munion-less-mono1 munion-less-mono2 mless-trans*)
done

lemma *mle-imp-omultiset*: M <#= N ==> *omultiset*(M) & *omultiset*(N)
by (*auto simp add*: *mle-def mless-imp-omultiset*)

lemma *mle-mono*: [| M <#= K; N <#= L |] ==> M +# N <#= K +# L
apply (*frule-tac M* = *M* **in** *mle-imp-omultiset*)
apply (*frule-tac M* = *N* **in** *mle-imp-omultiset*)
apply (*auto simp add*: *mle-def intro*: *munion-less-mono1 munion-less-mono2 munion-less-mono*)
done

lemma *omultiset-0* [*iff*]: *omultiset*(*0*)
by (*auto simp add*: *omultiset-def Mult-iff-multiset*)

lemma *empty-leI* [*simp*]: *omultiset*(*M*) ==> *0* <#= *M*
apply (*simp add*: *mle-def mless-def*)
apply (*subgoal-tac* ∃ *i*. *Ord* (*i*) & *M* ∈ *Mult*(*field*(*Memrel*(*i*))))
 prefer *2* **apply** (*simp add*: *omultiset-def*)
apply (*case-tac M*=*0*, *simp-all*, *clarify*)
apply (*subgoal-tac* <*0* +# *0*, *0* +# *M*> ∈ *multirel*(*field* (*Memrel*(*i*)), *Memrel*(*i*)))
apply (*rule-tac* [*2*] *one-step-implies-multirel*)
apply (*auto simp add*: *Mult-iff-multiset*)
done

lemma *munion-upper1*: [| *omultiset*(*M*); *omultiset*(*N*) |] ==> *M* <#= *M* +# *N*
apply (*subgoal-tac M* +# *0* <#= *M* +# *N*)
apply (*rule-tac* [*2*] *mle-mono*, *auto*)
done

end

9 An operator to "map" a relation over a list

theory *Rmap* **imports** *ZF* **begin**

consts
  rmap :: *i*=>*i*

inductive
  domains *rmap*(*r*) ⊆ *list*(*domain*(*r*)) × *list*(*range*(*r*))
  intros
    NilI: <*Nil*,*Nil*> ∈ *rmap*(*r*)

    ConsI: [| <*x*,*y*>: *r*; <*xs*,*ys*> ∈ *rmap*(*r*) |]
          ==> <*Cons*(*x*,*xs*), *Cons*(*y*,*ys*)> ∈ *rmap*(*r*)

  type-intros *domainI rangeI list.intros*

lemma *rmap-mono*: *r* ⊆ *s* ==> *rmap*(*r*) ⊆ *rmap*(*s*)
  apply (*unfold rmap.defs*)
  apply (*rule lfp-mono*)
    apply (*rule rmap.bnd-mono*)+
  apply (*assumption* | *rule Sigma-mono list-mono domain-mono range-mono basic-monos*)+
  done

inductive-cases
      Nil-rmap-case [*elim*!]: <*Nil*,*zs*> ∈ *rmap*(*r*)
  and *Cons-rmap-case* [*elim*!]: <*Cons*(*x*,*xs*),*zs*> ∈ *rmap*(*r*)

declare *rmap.intros* [*intro*]

lemma *rmap-rel-type*: $r \subseteq A \times B$ ==> $rmap(r) \subseteq list(A) \times list(B)$
  apply (*rule rmap.dom-subset* [*THEN subset-trans*])
  apply (*assumption* |
    rule domain-rel-subset range-rel-subset Sigma-mono list-mono)+
  done

lemma *rmap-total*: $A \subseteq domain(r)$ ==> $list(A) \subseteq domain(rmap(r))$
  apply (*rule subsetI*)
  apply (*erule list.induct*)
   apply *blast*+
  done

lemma *rmap-functional*: *function*(r) ==> *function*(*rmap*(r))
  apply (*unfold function-def*)
  apply (*rule impI* [*THEN allI*, *THEN allI*])
  apply (*erule rmap.induct*)
   apply *blast*+
  done

If f is a function then *rmap*(f) behaves as expected.

lemma *rmap-fun-type*: $f \in A$−>B ==> *rmap*(f): *list*(A)−>*list*(B)
  by (*simp add*: *Pi-iff rmap-rel-type rmap-functional rmap-total*)

lemma *rmap-Nil*: *rmap*(f)‘*Nil* = *Nil*
  by (*unfold apply-def*) *blast*

lemma *rmap-Cons*: [| $f \in A$−>B; $x \in A$; *xs*: *list*(A) |]
      ==> *rmap*(f) ‘ *Cons*(x,*xs*) = *Cons*(f‘x, *rmap*(f)‘*xs*)
  by (*blast intro*: *apply-equality apply-Pair rmap-fun-type rmap.intros*)

end

10 Meta-theory of propositional logic

theory *PropLog* **imports** *ZF* **begin**

Datatype definition of propositional logic formulae and inductive definition of the propositional tautologies.

Inductive definition of propositional logic. Soundness and completeness w.r.t. truth-tables.

Prove: If H |= p then G |= p where $G \in Fin(H)$

10.1 The datatype of propositions

consts
  propn :: *i*

```
datatype propn =
    Fls
  | Var (n ∈ nat)    (‹#-› [100] 100)
  | Imp (p ∈ propn, q ∈ propn)    (infixr ‹=>› 90)
```

10.2 The proof system

```
consts thms    :: i => i

abbreviation
  thms-syntax :: [i,i] => o    (infixl ‹|-› 50)
  where H |- p == p ∈ thms(H)

inductive
  domains thms(H) ⊆ propn
  intros
    H:  [| p ∈ H;  p ∈ propn |] ==> H |- p
    K:  [| p ∈ propn;  q ∈ propn |] ==> H |- p=>q=>p
    S:  [| p ∈ propn;  q ∈ propn;  r ∈ propn |]
          ==> H |- (p=>q=>r) => (p=>q) => p=>r
    DN: p ∈ propn ==> H |- ((p=>Fls) => Fls) => p
    MP: [| H |- p=>q;  H |- p;  p ∈ propn;  q ∈ propn |] ==> H |- q
  type-intros propn.intros

declare propn.intros [simp]
```

10.3 The semantics

10.3.1 Semantics of propositional logic.

```
consts
  is-true-fun :: [i,i] => i
primrec
  is-true-fun(Fls, t) = 0
  is-true-fun(Var(v), t) = (if v ∈ t then 1 else 0)
  is-true-fun(p=>q, t) = (if is-true-fun(p,t) = 1 then is-true-fun(q,t) else 1)

definition
  is-true :: [i,i] => o  where
  is-true(p,t) == is-true-fun(p,t) = 1
  — this definition is required since predicates can't be recursive

lemma is-true-Fls [simp]: is-true(Fls,t) ⟷ False
  by (simp add: is-true-def)

lemma is-true-Var [simp]: is-true(#v,t) ⟷ v ∈ t
  by (simp add: is-true-def)

lemma is-true-Imp [simp]: is-true(p=>q,t) ⟷ (is-true(p,t)⟶is-true(q,t))
  by (simp add: is-true-def)
```

10.3.2 Logical consequence

For every valuation, if all elements of H are true then so is p.

definition
  logcon :: [*i*,*i*] => *o*   (**infixl** ‹|=› *50*) **where**
  H |= *p* == ∀ *t*. (∀ *q* ∈ *H*. *is-true*(*q*,*t*)) ⟶ *is-true*(*p*,*t*)

A finite set of hypotheses from t and the *Var*s in p.

consts
  hyps :: [*i*,*i*] => *i*
primrec
  hyps(*Fls*, *t*) = *0*
  hyps(*Var*(*v*), *t*) = (*if* *v* ∈ *t* *then* {#*v*} *else* {#*v*=>*Fls*})
  hyps(*p*=>*q*, *t*) = *hyps*(*p*,*t*) ∪ *hyps*(*q*,*t*)

10.4 Proof theory of propositional logic

lemma *thms-mono*: *G* ⊆ *H* ==> *thms*(*G*) ⊆ *thms*(*H*)
  apply (*unfold thms.defs*)
  apply (*rule lfp-mono*)
   apply (*rule thms.bnd-mono*)+
  apply (*assumption* | *rule univ-mono basic-monos*)+
  done

lemmas *thms-in-pl* = *thms.dom-subset* [*THEN subsetD*]

inductive-cases *ImpE*: *p*=>*q* ∈ *propn*

lemma *thms-MP*: [| *H* |− *p*=>*q*; *H* |− *p* |] ==> *H* |− *q*
  — Stronger Modus Ponens rule: no typechecking!
  apply (*rule thms.MP*)
    apply (*erule asm-rl thms-in-pl thms-in-pl* [*THEN ImpE*])+
  done

lemma *thms-I*: *p* ∈ *propn* ==> *H* |− *p*=>*p*
  — Rule is called I for Identity Combinator, not for Introduction.
  apply (*rule thms.S* [*THEN thms-MP*, *THEN thms-MP*])
     apply (*rule-tac* [*5*] *thms.K*)
     apply (*rule-tac* [*4*] *thms.K*)
      apply *simp-all*
  done

10.4.1 Weakening, left and right

lemma *weaken-left*: [| *G* ⊆ *H*; *G*|−*p* |] ==> *H*|−*p*
  — Order of premises is convenient with *THEN*
  by (*erule thms-mono* [*THEN subsetD*])

lemma *weaken-left-cons*: *H* |− *p* ==> *cons*(*a*,*H*) |− *p*

by (*erule subset-consI* [*THEN weaken-left*])

lemmas *weaken-left-Un1* = *Un-upper1* [*THEN weaken-left*]
lemmas *weaken-left-Un2* = *Un-upper2* [*THEN weaken-left*]

lemma *weaken-right*: [| H |− q; $p \in propn$ |] ==> H |− p=>q
by (*simp-all add*: *thms.K* [*THEN thms-MP*] *thms-in-pl*)

10.4.2 The deduction theorem

theorem *deduction*: [| *cons*(p,H) |− q; $p \in propn$ |] ==> H |− p=>q
apply (*erule thms.induct*)
apply (*blast intro*: *thms-I thms.H* [*THEN weaken-right*])
apply (*blast intro*: *thms.K* [*THEN weaken-right*])
apply (*blast intro*: *thms.S* [*THEN weaken-right*])
apply (*blast intro*: *thms.DN* [*THEN weaken-right*])
apply (*blast intro*: *thms.S* [*THEN thms-MP* [*THEN thms-MP*]])
done

10.4.3 The cut rule

lemma *cut*: [| H|−p; *cons*(p,H) |− q |] ==> H |− q
apply (*rule deduction* [*THEN thms-MP*])
apply (*simp-all add*: *thms-in-pl*)
done

lemma *thms-FlsE*: [| H |− *Fls*; $p \in propn$ |] ==> H |− p
apply (*rule thms.DN* [*THEN thms-MP*])
apply (*rule-tac* [*2*] *weaken-right*)
apply (*simp-all add*: *propn.intros*)
done

lemma *thms-notE*: [| H |− p=>*Fls*; H |− p; $q \in propn$ |] ==> H |− q
by (*erule thms-MP* [*THEN thms-FlsE*])

10.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: H |− p ==> H |= p
apply (*unfold logcon-def*)
apply (*induct set*: *thms*)
apply *auto*
done

10.5 Completeness

10.5.1 Towards the completeness proof

lemma *Fls-Imp*: [| H |− p=>*Fls*; $q \in propn$ |] ==> H |− p=>q
apply (*frule thms-in-pl*)
apply (*rule deduction*)

```
  apply (rule weaken-left-cons [THEN thms-notE])
   apply (blast intro: thms.H elim: ImpE)+
 done

lemma Imp-Fls: [| H |- p;  H |- q=>Fls |] ==> H |- (p=>q)=>Fls
 apply (frule thms-in-pl)
 apply (frule thms-in-pl [of concl: q=>Fls])
 apply (rule deduction)
  apply (erule weaken-left-cons [THEN thms-MP])
  apply (rule consI1 [THEN thms.H, THEN thms-MP])
   apply (blast intro: weaken-left-cons elim: ImpE)+
 done

lemma hyps-thms-if:
    p ∈ propn ==> hyps(p,t) |- (if is-true(p,t) then p else p=>Fls)
 — Typical example of strengthening the induction statement.
 apply simp
 apply (induct-tac p)
   apply (simp-all add: thms-I thms.H)
 apply (safe elim!: Fls-Imp [THEN weaken-left-Un1] Fls-Imp [THEN weaken-left-Un2])
 apply (blast intro: weaken-left-Un1 weaken-left-Un2 weaken-right Imp-Fls)+
 done

lemma logcon-thms-p: [| p ∈ propn;  0 |= p |] ==> hyps(p,t) |- p
 — Key lemma for completeness; yields a set of assumptions satisfying p
 apply (drule hyps-thms-if)
 apply (simp add: logcon-def)
 done
```

For proving certain theorems in our new propositional logic.

```
lemmas propn-SIs = propn.intros deduction
 and propn-Is = thms-in-pl thms.H thms.H [THEN thms-MP]
```

The excluded middle in the form of an elimination rule.

```
lemma thms-excluded-middle:
    [| p ∈ propn;  q ∈ propn |] ==> H |- (p=>q) => ((p=>Fls)=>q) => q
 apply (rule deduction [THEN deduction])
   apply (rule thms.DN [THEN thms-MP])
    apply (best intro!: propn-SIs intro: propn-Is)+
 done

lemma thms-excluded-middle-rule:
 [| cons(p,H) |- q;  cons(p=>Fls,H) |- q;  p ∈ propn |] ==> H |- q
 — Hard to prove directly because it requires cuts
 apply (rule thms-excluded-middle [THEN thms-MP, THEN thms-MP])
    apply (blast intro!: propn-SIs intro: propn-Is)+
 done
```

10.5.2 Completeness – lemmas for reducing the set of assumptions

For the case $hyps(p,\ t)\ -\ cons(\#v,\ Y)\ |{-}\ p$ we also have $hyps(p,\ t)\ -\ \{\#v\}\subseteq hyps(p,\ t\ -\ \{v\})$.

lemma *hyps-Diff*:
  $p \in propn \;{=}{=}{>}\; hyps(p,\ t{-}\{v\}) \subseteq cons(\#v{=}{>}Fls,\ hyps(p,t){-}\{\#v\})$
 by (*induct set*: *propn*) *auto*

For the case $hyps(p,\ t)\ -\ cons(\#v\ {=}{>}\ Fls,\ Y)\ |{-}\ p$ we also have $hyps(p,\ t)\ -\ \{\#v\ {=}{>}\ Fls\} \subseteq hyps(p,\ cons(v,\ t))$.

lemma *hyps-cons*:
  $p \in propn \;{=}{=}{>}\; hyps(p,\ cons(v,t)) \subseteq cons(\#v,\ hyps(p,t){-}\{\#v{=}{>}Fls\})$
 by (*induct set*: *propn*) *auto*

Two lemmas for use with *weaken-left*

lemma *cons-Diff-same*: $B{-}C \subseteq cons(a,\ B{-}cons(a,C))$
 by *blast*

lemma *cons-Diff-subset2*: $cons(a,\ B{-}\{c\})\ -\ D \subseteq cons(a,\ B{-}cons(c,D))$
 by *blast*

The set $hyps(p,\ t)$ is finite, and elements have the form $\#v$ or $\#v\ {=}{>}\ Fls$; could probably prove the stronger $hyps(p,\ t) \in Fin(hyps(p,\ 0) \cup hyps(p,\ nat))$.

lemma *hyps-finite*: $p \in propn \;{=}{=}{>}\; hyps(p,t) \in Fin(\bigcup v \in nat.\ \{\#v,\ \#v{=}{>}Fls\})$
 by (*induct set*: *propn*) *auto*

lemmas *Diff-weaken-left* = *Diff-mono* [*OF - subset-refl*, *THEN weaken-left*]

Induction on the finite set of assumptions $hyps(p,\ t0)$. We may repeatedly subtract assumptions until none are left!

lemma *completeness-0-lemma* [*rule-format*]:
  $[|\ p \in propn;\ \ 0\ |{=}\ p\ |] \;{=}{=}{>}\; \forall t.\ hyps(p,t)\ -\ hyps(p,t0)\ |{-}\ p$
 apply (*frule hyps-finite*)
 apply (*erule Fin-induct*)
  apply (*simp add*: *logcon-thms-p Diff-0*)

inductive step

 apply *safe*

Case $hyps(p,\ t)\ -\ cons(\#v,\ Y)\ |{-}\ p$

  apply (*rule thms-excluded-middle-rule*)
   apply (*erule-tac* [*3*] *propn.intros*)
  apply (*blast intro*: *cons-Diff-same* [*THEN weaken-left*])
  apply (*blast intro*: *cons-Diff-subset2* [*THEN weaken-left*]
   hyps-Diff [*THEN Diff-weaken-left*])

Case *hyps(p, t) − cons(#v => Fls, Y) |− p*

apply (*rule thms-excluded-middle-rule*)
apply (*erule-tac [3] propn.intros*)
apply (*blast intro: cons-Diff-subset2 [THEN weaken-left] hyps-cons [THEN Diff-weaken-left]*)
apply (*blast intro: cons-Diff-same [THEN weaken-left]*)
done

10.5.3 Completeness theorem

lemma *completeness-0*: [| *p* ∈ *propn*; *0* |= *p* |] ==> *0* |− *p*
— The base case for completeness
apply (*rule Diff-cancel [THEN subst]*)
apply (*blast intro: completeness-0-lemma*)
done

lemma *logcon-Imp*: [| *cons(p,H)* |= *q* |] ==> *H* |= *p=>q*
— A semantic analogue of the Deduction Theorem
by (*simp add: logcon-def*)

lemma *completeness*:
H ∈ *Fin(propn)* ==> *p* ∈ *propn* ⟹ *H* |= *p* ⟹ *H* |− *p*
apply (*induct arbitrary: p set: Fin*)
apply (*safe intro!: completeness-0*)
apply (*rule weaken-left-cons [THEN thms-MP]*)
apply (*blast intro!: logcon-Imp propn.intros*)
apply (*blast intro: propn-Is*)
done

theorem *thms-iff*: *H* ∈ *Fin(propn)* ==> *H* |− *p* ⟷ *H* |= *p* ∧ *p* ∈ *propn*
by (*blast intro: soundness completeness thms-in-pl*)

end

11 Lists of n elements

theory *ListN* **imports** *ZF* **begin**

Inductive definition of lists of *n* elements; see [3].

consts *listn* :: *i=>i*
inductive
domains *listn(A)* ⊆ *nat* × *list(A)*
intros
NilI: <*0,Nil*> ∈ *listn(A)*
ConsI: [| *a* ∈ *A*; <*n,l*> ∈ *listn(A)* |] ==> <*succ(n), Cons(a,l)*> ∈ *listn(A)*
type-intros *nat-typechecks list.intros*

```
lemma list-into-listn: l ∈ list(A) ==> <length(l),l> ∈ listn(A)
  by (induct set: list) (simp-all add: listn.intros)

lemma listn-iff: <n,l> ∈ listn(A) ⟷ l ∈ list(A) & length(l)=n
  apply (rule iffI)
   apply (erule listn.induct)
    apply auto
  apply (blast intro: list-into-listn)
  done

lemma listn-image-eq: listn(A)``{n} = {l ∈ list(A). length(l)=n}
  apply (rule equality-iffI)
  apply (simp add: listn-iff separation image-singleton-iff)
  done

lemma listn-mono: A ⊆ B ==> listn(A) ⊆ listn(B)
  apply (unfold listn.defs)
  apply (rule lfp-mono)
    apply (rule listn.bnd-mono)+
  apply (assumption | rule univ-mono Sigma-mono list-mono basic-monos)+
  done

lemma listn-append:
    [| <n,l> ∈ listn(A); <n',l'> ∈ listn(A) |] ==> <n#+n', l@l'> ∈ listn(A)
  apply (erule listn.induct)
   apply (frule listn.dom-subset [THEN subsetD])
   apply (simp-all add: listn.intros)
  done

inductive-cases
      Nil-listn-case: <i,Nil> ∈ listn(A)
  and Cons-listn-case: <i,Cons(x,l)> ∈ listn(A)

inductive-cases
      zero-listn-case: <0,l> ∈ listn(A)
  and succ-listn-case: <succ(i),l> ∈ listn(A)

end
```

12 Combinatory Logic example: the Church-Rosser Theorem

```
theory Comb
imports ZF
begin
```

Curiously, combinators do not include free variables.

Example taken from [1].

12.1 Definitions

Datatype definition of combinators S and K.

```
consts comb :: i
datatype comb =
  K
| S
| app (p ∈ comb, q ∈ comb)  (infixl ‹•› 90)
```

Inductive definition of contractions, $\rightarrow^1$ and (multi-step) reductions, $\rightarrow$.

```
consts contract  :: i
abbreviation contract-syntax :: [i,i] ⇒ o  (infixl ‹→¹› 50)
  where p →¹ q ≡ <p,q> ∈ contract

abbreviation contract-multi :: [i,i] ⇒ o  (infixl ‹→› 50)
  where p → q ≡ <p,q> ∈ contract^*

inductive
  domains contract ⊆ comb × comb
  intros
    K:   [| p ∈ comb;  q ∈ comb |] ==> K•p•q →¹ p
    S:   [| p ∈ comb;  q ∈ comb;  r ∈ comb |] ==> S•p•q•r →¹ (p•r)•(q•r)
    Ap1: [| p→¹q;  r ∈ comb |] ==> p•r →¹ q•r
    Ap2: [| p→¹q;  r ∈ comb |] ==> r•p →¹ r•q
  type-intros comb.intros
```

Inductive definition of parallel contractions, $\Rrightarrow^1$ and (multi-step) parallel reductions, $\Rrightarrow$.

```
consts parcontract :: i

abbreviation parcontract-syntax :: [i,i] => o  (infixl ‹⇛¹› 50)
  where p ⇛¹ q == <p,q> ∈ parcontract

abbreviation parcontract-multi :: [i,i] => o  (infixl ‹⇛› 50)
  where p ⇛ q == <p,q> ∈ parcontract^+

inductive
  domains parcontract ⊆ comb × comb
  intros
    refl: [| p ∈ comb |] ==> p ⇛¹ p
    K:    [| p ∈ comb;  q ∈ comb |] ==> K•p•q ⇛¹ p
    S:    [| p ∈ comb;  q ∈ comb;  r ∈ comb |] ==> S•p•q•r ⇛¹ (p•r)•(q•r)
    Ap:   [| p⇛¹q;  r⇛¹s |] ==> p•r ⇛¹ q•s
  type-intros comb.intros
```

Misc definitions.

```
definition I :: i
  where I ≡ S•K•K
```

definition *diamond* :: *i* ⇒ *o*
  where *diamond*(*r*) ≡
    ∀ *x y*. <*x*,*y*>∈*r* ⟶ (∀ *y*′. <*x*,*y*′>∈*r* ⟶ (∃ *z*. <*y*,*z*>∈*r* & <*y*′,*z*> ∈ *r*))

12.2 Transitive closure preserves the Church-Rosser property

lemma *diamond-strip-lemmaD* [*rule-format*]:
  [| *diamond*(*r*); <*x*,*y*>:*r*^+ |] ==>
    ∀ *y*′. <*x*,*y*′>:*r* ⟶ (∃ *z*. <*y*′,*z*>: *r*^+ & <*y*,*z*>: *r*)
  apply (*unfold diamond-def*)
  apply (*erule trancl-induct*)
   apply (*blast intro*: *r-into-trancl*)
  apply *clarify*
  apply (*drule spec* [*THEN mp*], *assumption*)
  apply (*blast intro*: *r-into-trancl trans-trancl* [*THEN transD*])
  done

lemma *diamond-trancl*: *diamond*(*r*) ==> *diamond*(*r*^+)
  apply (*simp* (*no-asm-simp*) *add*: *diamond-def*)
  apply (*rule impI* [*THEN allI*, *THEN allI*])
  apply (*erule trancl-induct*)
   apply *auto*
   apply (*best intro*: *r-into-trancl trans-trancl* [*THEN transD*]
     dest: *diamond-strip-lemmaD*)+
  done

inductive-cases *Ap-E* [*elim*!]: *p*·*q* ∈ *comb*

12.3 Results about Contraction

For type checking: replaces $a \rightarrow^1 b$ by $a,\ b \in comb$.

lemmas *contract-combE2* = *contract.dom-subset* [*THEN subsetD*, *THEN SigmaE2*]
  and *contract-combD1* = *contract.dom-subset* [*THEN subsetD*, *THEN SigmaD1*]
  and *contract-combD2* = *contract.dom-subset* [*THEN subsetD*, *THEN SigmaD2*]

lemma *field-contract-eq*: *field*(*contract*) = *comb*
  by (*blast intro*: *contract.K elim*!: *contract-combE2*)

lemmas *reduction-refl* =
  field-contract-eq [*THEN equalityD2*, *THEN subsetD*, *THEN rtrancl-refl*]

lemmas *rtrancl-into-rtrancl2* =
  r-into-rtrancl [*THEN trans-rtrancl* [*THEN transD*]]

declare *reduction-refl* [*intro*!] *contract.K* [*intro*!] *contract.S* [*intro*!]

lemmas *reduction-rls* =

contract.K [*THEN rtrancl-into-rtrancl2*]
contract.S [*THEN rtrancl-into-rtrancl2*]
contract.Ap1 [*THEN rtrancl-into-rtrancl2*]
contract.Ap2 [*THEN rtrancl-into-rtrancl2*]

lemma $p \in comb$ ==> $I{\cdot}p \to p$
— Example only: not used
unfolding *I-def* **by** (*blast intro*: *reduction-rls*)

lemma *comb-I*: $I \in comb$
unfolding *I-def* **by** *blast*

12.4 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases *K-contractE* [*elim*!]: $K \to^1 r$
and *S-contractE* [*elim*!]: $S \to^1 r$
and *Ap-contractE* [*elim*!]: $p{\cdot}q \to^1 r$

lemma *I-contract-E*: $I \to^1 r$ ==> P
by (*auto simp add*: *I-def*)

lemma *K1-contractD*: $K{\cdot}p \to^1 r$ ==> $(\exists\, q.\ r = K{\cdot}q\ \&\ p \to^1 q)$
by *auto*

lemma *Ap-reduce1*: [| $p \to q$; $r \in comb$ |] ==> $p{\cdot}r \to q{\cdot}r$
apply (*frule rtrancl-type* [*THEN subsetD*, *THEN SigmaD1*])
apply (*drule field-contract-eq* [*THEN equalityD1*, *THEN subsetD*])
apply (*erule rtrancl-induct*)
apply (*blast intro*: *reduction-rls*)
apply (*erule trans-rtrancl* [*THEN transD*])
apply (*blast intro*: *contract-combD2 reduction-rls*)
done

lemma *Ap-reduce2*: [| $p \to q$; $r \in comb$ |] ==> $r{\cdot}p \to r{\cdot}q$
apply (*frule rtrancl-type* [*THEN subsetD*, *THEN SigmaD1*])
apply (*drule field-contract-eq* [*THEN equalityD1*, *THEN subsetD*])
apply (*erule rtrancl-induct*)
apply (*blast intro*: *reduction-rls*)
apply (*blast intro*: *trans-rtrancl* [*THEN transD*]
contract-combD2 reduction-rls)
done

Counterexample to the diamond property for $\to^1$.

lemma *KIII-contract1*: $K{\cdot}I{\cdot}(I{\cdot}I) \to^1 I$
by (*blast intro*: *comb-I*)

lemma *KIII-contract2*: $K{\cdot}I{\cdot}(I{\cdot}I) \to^1 K{\cdot}I{\cdot}((K{\cdot}I){\cdot}(K{\cdot}I))$
by (*unfold I-def*) (*blast intro*: *contract.intros*)

lemma *KIII-contract3*: $K{\cdot}I{\cdot}((K{\cdot}I){\cdot}(K{\cdot}I)) \rightarrow^1 I$
by (*blast intro*: *comb-I*)

lemma *not-diamond-contract*: $\neg$ *diamond*(*contract*)
apply (*unfold diamond-def*)
apply (*blast intro*: *KIII-contract1 KIII-contract2 KIII-contract3*
elim!: *I-contract-E*)
done

12.5 Results about Parallel Contraction

For type checking: replaces $a \Rrightarrow^1 b$ by $a,\ b \in comb$

lemmas *parcontract-combE2* = *parcontract.dom-subset* [*THEN subsetD*, *THEN SigmaE2*]
and *parcontract-combD1* = *parcontract.dom-subset* [*THEN subsetD*, *THEN SigmaD1*]
and *parcontract-combD2* = *parcontract.dom-subset* [*THEN subsetD*, *THEN SigmaD2*]

lemma *field-parcontract-eq*: *field*(*parcontract*) = *comb*
by (*blast intro*: *parcontract.K elim*!: *parcontract-combE2*)

Derive a case for each combinator constructor.

inductive-cases
K-parcontractE [*elim*!]: $K \Rrightarrow^1 r$
and *S-parcontractE* [*elim*!]: $S \Rrightarrow^1 r$
and *Ap-parcontractE* [*elim*!]: $p{\cdot}q \Rrightarrow^1 r$

declare *parcontract.intros* [*intro*]

12.6 Basic properties of parallel contraction

lemma *K1-parcontractD* [*dest*!]:
$K{\cdot}p \Rrightarrow^1 r$ ==> $(\exists p'.\ r = K{\cdot}p' \ \&\ p \Rrightarrow^1 p')$
by *auto*

lemma *S1-parcontractD* [*dest*!]:
$S{\cdot}p \Rrightarrow^1 r$ ==> $(\exists p'.\ r = S{\cdot}p' \ \&\ p \Rrightarrow^1 p')$
by *auto*

lemma *S2-parcontractD* [*dest*!]:
$S{\cdot}p{\cdot}q \Rrightarrow^1 r$ ==> $(\exists p'\ q'.\ r = S{\cdot}p'{\cdot}q' \ \&\ p \Rrightarrow^1 p' \ \&\ q \Rrightarrow^1 q')$
by *auto*

lemma *diamond-parcontract*: *diamond*(*parcontract*)
— Church-Rosser property for parallel contraction
apply (*unfold diamond-def*)
apply (*rule impI* [*THEN allI*, *THEN allI*])

```
  apply (erule parcontract.induct)
     apply (blast elim!: comb.free-elims  intro: parcontract-combD2)+
  done
```

Equivalence of $p \to q$ and $p \Rrightarrow q$.

```
lemma contract-imp-parcontract: p→¹q ==> p⇛¹q
  by (induct set: contract) auto

lemma reduce-imp-parreduce: p→q ==> p⇛q
  apply (frule rtrancl-type [THEN subsetD, THEN SigmaD1])
  apply (drule field-contract-eq [THEN equalityD1, THEN subsetD])
  apply (erule rtrancl-induct)
   apply (blast intro: r-into-trancl)
  apply (blast intro: contract-imp-parcontract r-into-trancl
    trans-trancl [THEN transD])
  done

lemma parcontract-imp-reduce: p⇛¹q ==> p→q
  apply (induct set: parcontract)
     apply (blast intro: reduction-rls)
    apply (blast intro: reduction-rls)
   apply (blast intro: reduction-rls)
  apply (blast intro: trans-rtrancl [THEN transD]
    Ap-reduce1 Ap-reduce2 parcontract-combD1 parcontract-combD2)
  done

lemma parreduce-imp-reduce: p⇛q ==> p→q
  apply (frule trancl-type [THEN subsetD, THEN SigmaD1])
  apply (drule field-parcontract-eq [THEN equalityD1, THEN subsetD])
  apply (erule trancl-induct, erule parcontract-imp-reduce)
  apply (erule trans-rtrancl [THEN transD])
  apply (erule parcontract-imp-reduce)
  done

lemma parreduce-iff-reduce: p⇛q ⟷ p→q
  by (blast intro: parreduce-imp-reduce reduce-imp-parreduce)

end
```

13 Primitive Recursive Functions: the inductive definition

theory *Primrec* **imports** *ZF* **begin**

Proof adopted from [4].

See also [2, page 250, exercise 11].

13.1 Basic definitions

definition
$SC :: i$ **where**
$SC == \lambda l \in list(nat).\ list\text{-}case(0,\ \lambda x\ xs.\ succ(x),\ l)$

definition
$CONSTANT :: i{=}{>}i$ **where**
$CONSTANT(k) == \lambda l \in list(nat).\ k$

definition
$PROJ :: i{=}{>}i$ **where**
$PROJ(i) == \lambda l \in list(nat).\ list\text{-}case(0,\ \lambda x\ xs.\ x,\ drop(i,l))$

definition
$COMP :: [i,i]{=}{>}i$ **where**
$COMP(g,fs) == \lambda l \in list(nat).\ g\ `\ map(\lambda f.\ f`l,\ fs)$

definition
$PREC :: [i,i]{=}{>}i$ **where**
$PREC(f,g) ==$
$\lambda l \in list(nat).\ list\text{-}case(0,$
$\lambda x\ xs.\ rec(x,\ f`xs,\ \lambda y\ r.\ g\ `\ Cons(r,\ Cons(y,\ xs))),\ l)$
— Note that g is applied first to $PREC(f,\ g)\ `\ y$ and then to y!

consts
$ACK :: i{=}{>}i$
primrec
$ACK(0) = SC$
$ACK(succ(i)) = PREC\ (CONSTANT\ (ACK(i)\ `\ [1]),\ COMP(ACK(i),\ [PROJ(0)]))$

abbreviation
$ack :: [i,i]{=}{>}i$ **where**
$ack(x,y) == ACK(x)\ `\ [y]$

Useful special cases of evaluation.

lemma SC: $[|\ x \in nat;\ \ l \in list(nat)\ |] ==> SC\ `\ (Cons(x,l)) = succ(x)$
by $(simp\ add:\ SC\text{-}def)$

lemma $CONSTANT$: $l \in list(nat) ==> CONSTANT(k)\ `\ l = k$
by $(simp\ add:\ CONSTANT\text{-}def)$

lemma $PROJ\text{-}0$: $[|\ x \in nat;\ \ l \in list(nat)\ |] ==> PROJ(0)\ `\ (Cons(x,l)) = x$
by $(simp\ add:\ PROJ\text{-}def)$

lemma $COMP\text{-}1$: $l \in list(nat) ==> COMP(g,[f])\ `\ l = g`\ [f`l]$
by $(simp\ add:\ COMP\text{-}def)$

lemma $PREC\text{-}0$: $l \in list(nat) ==> PREC(f,g)\ `\ (Cons(0,l)) = f`l$

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

lemma *PREC-succ*:
[| $x \in nat$; $l \in list(nat)$ |]
==> $PREC(f,g)$ ‘ $(Cons(succ(x),l))$ =
g ‘ $Cons(PREC(f,g)‘(Cons(x,l)), Cons(x,l))$
by (*simp add*: *PREC-def*)

13.2 Inductive definition of the PR functions

consts
prim-rec :: *i*

inductive
domains *prim-rec* $\subseteq$ *list*(*nat*)−>*nat*
intros
$SC \in$ *prim-rec*
$k \in nat$ ==> $CONSTANT(k) \in$ *prim-rec*
$i \in nat$ ==> $PROJ(i) \in$ *prim-rec*
[| $g \in$ *prim-rec*; *fs*$\in$*list*(*prim-rec*) |] ==> $COMP(g,fs) \in$ *prim-rec*
[| $f \in$ *prim-rec*; $g \in$ *prim-rec* |] ==> $PREC(f,g) \in$ *prim-rec*
monos *list-mono*
con-defs *SC-def CONSTANT-def PROJ-def COMP-def PREC-def*
type-intros *nat-typechecks list.intros*
lam-type list-case-type drop-type map-type
apply-type rec-type

lemma *prim-rec-into-fun* [*TC*]: $c \in$ *prim-rec* ==> $c \in list(nat)$ −> *nat*
by (*erule subsetD* [*OF prim-rec.dom-subset*])

lemmas [*TC*] = *apply-type* [*OF prim-rec-into-fun*]

declare *prim-rec.intros* [*TC*]
declare *nat-into-Ord* [*TC*]
declare *rec-type* [*TC*]

lemma *ACK-in-prim-rec* [*TC*]: $i \in nat$ ==> $ACK(i) \in$ *prim-rec*
by (*induct set*: *nat*) *simp-all*

lemma *ack-type* [*TC*]: [| $i \in nat$; $j \in nat$ |] ==> $ack(i,j) \in nat$
by *auto*

13.3 Ackermann's function cases

lemma *ack-0*: $j \in nat$ ==> $ack(0,j) = succ(j)$
— PROPERTY A 1
by (*simp add*: *SC*)

lemma *ack-succ-0*: $ack(succ(i), 0) = ack(i,1)$

— PROPERTY A 2
by (*simp add*: *CONSTANT PREC-0*)

lemma *ack-succ-succ*:
[| *i*∈*nat*; *j*∈*nat* |] ==> *ack*(*succ*(*i*), *succ*(*j*)) = *ack*(*i*, *ack*(*succ*(*i*), *j*))
— PROPERTY A 3
by (*simp add*: *CONSTANT PREC-succ COMP-1 PROJ-0*)

lemmas [*simp*] = *ack-0 ack-succ-0 ack-succ-succ ack-type*
and [*simp del*] = *ACK.simps*

lemma *lt-ack2*: *i* ∈ *nat* ==> *j* ∈ *nat* ==> *j* < *ack*(*i*,*j*)
— PROPERTY A 4
apply (*induct i arbitrary*: *j set*: *nat*)
apply *simp*
apply (*induct-tac j*)
apply (*erule-tac* [*2*] *succ-leI* [*THEN lt-trans1*])
apply (*rule nat-0I* [*THEN nat-0-le*, *THEN lt-trans*])
apply *auto*
done

lemma *ack-lt-ack-succ2*: [|*i*∈*nat*; *j*∈*nat*|] ==> *ack*(*i*,*j*) < *ack*(*i*, *succ*(*j*))
— PROPERTY A 5-, the single-step lemma
by (*induct set*: *nat*) (*simp-all add*: *lt-ack2*)

lemma *ack-lt-mono2*: [| *j*<*k*; *i* ∈ *nat*; *k* ∈ *nat* |] ==> *ack*(*i*,*j*) < *ack*(*i*,*k*)
— PROPERTY A 5, monotonicity for <
apply (*frule lt-nat-in-nat*, *assumption*)
apply (*erule succ-lt-induct*)
apply *assumption*
apply (*rule-tac* [*2*] *lt-trans*)
apply (*auto intro*: *ack-lt-ack-succ2*)
done

lemma *ack-le-mono2*: [|*j*≤*k*; *i*∈*nat*; *k*∈*nat*|] ==> *ack*(*i*,*j*) ≤ *ack*(*i*,*k*)
— PROPERTY A 5', monotonicity for ≤
apply (*rule-tac f* = *λj. ack* (*i*,*j*) **in** *Ord-lt-mono-imp-le-mono*)
apply (*assumption* | *rule ack-lt-mono2 ack-type* [*THEN nat-into-Ord*])+
done

lemma *ack2-le-ack1*:
[| *i*∈*nat*; *j*∈*nat* |] ==> *ack*(*i*, *succ*(*j*)) ≤ *ack*(*succ*(*i*), *j*)
— PROPERTY A 6
apply (*induct-tac j*)
apply *simp-all*
apply (*rule ack-le-mono2*)
apply (*rule lt-ack2* [*THEN succ-leI*, *THEN le-trans*])
apply *auto*

done

lemma *ack-lt-ack-succ1*: [| $i \in nat$; $j \in nat$ |] ==> $ack(i,j) < ack(succ(i),j)$
— PROPERTY A 7-, the single-step lemma
apply (*rule ack-lt-mono2* [*THEN lt-trans2*])
apply (*rule-tac* [*4*] *ack2-le-ack1*)
apply *auto*
done

lemma *ack-lt-mono1*: [| $i<j$; $j \in nat$; $k \in nat$ |] ==> $ack(i,k) < ack(j,k)$
— PROPERTY A 7, monotonicity for <
apply (*frule lt-nat-in-nat*, *assumption*)
apply (*erule succ-lt-induct*)
apply *assumption*
apply (*rule-tac* [*2*] *lt-trans*)
apply (*auto intro*: *ack-lt-ack-succ1*)
done

lemma *ack-le-mono1*: [| $i \le j$; $j \in nat$; $k \in nat$ |] ==> $ack(i,k) \le ack(j,k)$
— PROPERTY A 7', monotonicity for $\le$
apply (*rule-tac* $f = \lambda j.\ ack\ (j,k)$ **in** *Ord-lt-mono-imp-le-mono*)
apply (*assumption* | *rule ack-lt-mono1 ack-type* [*THEN nat-into-Ord*])+
done

lemma *ack-1*: $j \in nat$ ==> $ack(1,j) = succ(succ(j))$
— PROPERTY A 8
by (*induct set*: *nat*) *simp-all*

lemma *ack-2*: $j \in nat$ ==> $ack(succ(1),j) = succ(succ(succ(j\#+j)))$
— PROPERTY A 9
by (*induct set*: *nat*) (*simp-all add*: *ack-1*)

lemma *ack-nest-bound*:
[| $i1 \in nat$; $i2 \in nat$; $j \in nat$ |]
==> $ack(i1, ack(i2,j)) < ack(succ(succ(i1\#+i2)), j)$
— PROPERTY A 10
apply (*rule lt-trans2* [*OF - ack2-le-ack1*])
apply *simp*
apply (*rule add-le-self* [*THEN ack-le-mono1*, *THEN lt-trans1*])
apply *auto*
apply (*force intro*: *add-le-self2* [*THEN ack-lt-mono1*, *THEN ack-lt-mono2*])
done

lemma *ack-add-bound*:
[| $i1 \in nat$; $i2 \in nat$; $j \in nat$ |]
==> $ack(i1,j) \#+ ack(i2,j) < ack(succ(succ(succ(succ(i1\#+i2)))), j)$
— PROPERTY A 11
apply (*rule-tac* $j = ack\ (succ\ (1),\ ack\ (i1\ \#+\ i2,\ j))$ **in** *lt-trans*)
apply (*simp add*: *ack-2*)

apply (*rule-tac* [*2*] *ack-nest-bound* [*THEN lt-trans2*])
apply (*rule add-le-mono* [*THEN leI*, *THEN leI*])
apply (*auto intro*: *add-le-self add-le-self2 ack-le-mono1*)
done

lemma *ack-add-bound2*:
[| *i* < *ack*(*k*,*j*); *j* ∈ *nat*; *k* ∈ *nat* |]
==> *i*#+*j* < *ack*(*succ*(*succ*(*succ*(*succ*(*k*)))), *j*)
— PROPERTY A 12.
— Article uses existential quantifier but the ALF proof used *k* #+ #*4*.
— Quantified version must be nested ∃ *k*′. ∀ *i*,*j* . . ..
apply (*rule-tac j* = *ack* (*k*,*j*) #+ *ack* (*0*,*j*) **in** *lt-trans*)
apply (*rule-tac* [*2*] *ack-add-bound* [*THEN lt-trans2*])
apply (*rule add-lt-mono*)
apply *auto*
done

13.4 Main result

declare *list-add-type* [*simp*]

lemma *SC-case*: *l* ∈ *list*(*nat*) ==> *SC* ‘ *l* < *ack*(*1*, *list-add*(*l*))
apply (*unfold SC-def*)
apply (*erule list.cases*)
apply (*simp add*: *succ-iff*)
apply (*simp add*: *ack-1 add-le-self*)
done

lemma *lt-ack1*: [| *i* ∈ *nat*; *j* ∈ *nat* |] ==> *i* < *ack*(*i*,*j*)
— PROPERTY A 4’? Extra lemma needed for *CONSTANT* case, constant functions.
apply (*induct-tac i*)
apply (*simp add*: *nat-0-le*)
apply (*erule lt-trans1* [*OF succ-leI ack-lt-ack-succ1*])
apply *auto*
done

lemma *CONSTANT-case*:
[| *l* ∈ *list*(*nat*); *k* ∈ *nat* |] ==> *CONSTANT*(*k*) ‘ *l* < *ack*(*k*, *list-add*(*l*))
by (*simp add*: *CONSTANT-def lt-ack1*)

lemma *PROJ-case* [*rule-format*]:
l ∈ *list*(*nat*) ==> ∀ *i* ∈ *nat*. *PROJ*(*i*) ‘ *l* < *ack*(*0*, *list-add*(*l*))
apply (*unfold PROJ-def*)
apply *simp*
apply (*erule list.induct*)
apply (*simp add*: *nat-0-le*)
apply *simp*
apply (*rule ballI*)

```
  apply (erule-tac n = i in natE)
   apply (simp add: add-le-self)
  apply simp
  apply (erule bspec [THEN lt-trans2])
   apply (rule-tac [2] add-le-self2 [THEN succ-leI])
   apply auto
  done
```

COMP case.

```
lemma COMP-map-lemma:
  fs ∈ list({f ∈ prim-rec. ∃ kf ∈ nat. ∀ l ∈ list(nat). f‘l < ack(kf, list-add(l))})
    ==> ∃ k ∈ nat. ∀ l ∈ list(nat).
      list-add(map(λf. f ‘ l, fs)) < ack(k, list-add(l))
  apply (induct set: list)
   apply (rule-tac x = 0 in bexI)
    apply (simp-all add: lt-ack1 nat-0-le)
  apply clarify
  apply (rule ballI [THEN bexI])
  apply (rule add-lt-mono [THEN lt-trans])
       apply (rule-tac [5] ack-add-bound)
         apply blast
        apply auto
  done

lemma COMP-case:
 [| kg∈nat;
     ∀ l ∈ list(nat). g‘l < ack(kg, list-add(l));
     fs ∈ list({f ∈ prim-rec .
                 ∃ kf ∈ nat. ∀ l ∈ list(nat).
                        f‘l < ack(kf, list-add(l))}) |]
   ==> ∃ k ∈ nat. ∀ l ∈ list(nat). COMP(g,fs)‘l < ack(k, list-add(l))
  apply (simp add: COMP-def)
  apply (frule list-CollectD)
  apply (erule COMP-map-lemma [THEN bexE])
  apply (rule ballI [THEN bexI])
   apply (erule bspec [THEN lt-trans])
    apply (rule-tac [2] lt-trans)
     apply (rule-tac [3] ack-nest-bound)
       apply (erule-tac [2] bspec [THEN ack-lt-mono2])
         apply auto
  done
```

PREC case.

```
lemma PREC-case-lemma:
 [| ∀ l ∈ list(nat). f‘l #+ list-add(l) < ack(kf, list-add(l));
     ∀ l ∈ list(nat). g‘l #+ list-add(l) < ack(kg, list-add(l));
     f ∈ prim-rec;  kf∈nat;
     g ∈ prim-rec;  kg∈nat;
```

$l \in list(nat)$ |]
==> $PREC(f,g)$ ‘l #+ $list\text{-}add(l) < ack(succ(kf\#+kg), list\text{-}add(l))$
apply (*unfold PREC-def*)
apply (*erule list.cases*)
apply (*simp add*: *lt-trans* [*OF nat-le-refl lt-ack2*])
apply *simp*
apply (*erule ssubst*) — get rid of the needless assumption
apply (*induct-tac a*)
apply *simp-all*

base case

apply (*rule lt-trans*, *erule bspec*, *assumption*)
apply (*simp add*: *add-le-self* [*THEN ack-lt-mono1*])

ind step

apply (*rule succ-leI* [*THEN lt-trans1*])
apply (*rule-tac j = g ‘ ll #+ mm* **for** *ll mm* **in** *lt-trans1*)
apply (*erule-tac* [2] *bspec*)
apply (*rule nat-le-refl* [*THEN add-le-mono*])
apply *typecheck*
apply (*simp add*: *add-le-self2*)

final part of the simplification

apply *simp*
apply (*rule add-le-self2* [*THEN ack-le-mono1*, *THEN lt-trans1*])
apply (*erule-tac* [4] *ack-lt-mono2*)
apply *auto*
done

lemma *PREC-case*:
[| $f \in prim\text{-}rec$; $kf \in nat$;
$g \in prim\text{-}rec$; $kg \in nat$;
$\forall l \in list(nat).\ f‘l < ack(kf, list\text{-}add(l))$;
$\forall l \in list(nat).\ g‘l < ack(kg, list\text{-}add(l))$ |]
==> $\exists k \in nat.\ \forall l \in list(nat).\ PREC(f,g)‘l < ack(k, list\text{-}add(l))$
apply (*rule ballI* [*THEN bexI*])
apply (*rule lt-trans1* [*OF add-le-self PREC-case-lemma*])
apply *typecheck*
apply (*blast intro*: *ack-add-bound2 list-add-type*)+
done

lemma *ack-bounds-prim-rec*:
$f \in prim\text{-}rec$ ==> $\exists k \in nat.\ \forall l \in list(nat).\ f‘l < ack(k, list\text{-}add(l))$
apply (*induct set*: *prim-rec*)
apply (*auto intro*: *SC-case CONSTANT-case PROJ-case COMP-case PREC-case*)
done

theorem *ack-not-prim-rec*:
$(\lambda l \in list(nat).\ list\text{-}case(0, \lambda x\ xs.\ ack(x,x), l)) \notin prim\text{-}rec$

apply (*rule notI*)
apply (*drule ack-bounds-prim-rec*)
apply *force*
done

end

References

[1] J. Camilleri and T. F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, Aug. 1992.

[2] E. Mendelson. *Introduction to Mathematical Logic.* Chapman & Hall, fourth edition, 1997.

[3] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.

[4] N. Szasz. A machine checked proof that Ackermann's function is not primitive recursive. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 317–338. Cambridge University Press, 1993.