Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

September 11, 2023

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

datatype $data(A, B) =$
  Con0
  | *Con1* $(a \in A)$
  | *Con2* $(a \in A, b \in B)$
  | *Con3* $(a \in A, b \in B, d \in data(A, B))$

lemma *data-unfold*: $data(A, B) = (\{0\} + A) + (A \times B + A \times B \times data(A, B))$
  ⟨*proof*⟩

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

lemma *data-mono*: $[\![A \subseteq C; B \subseteq D]\!] \Longrightarrow data(A, B) \subseteq data(C, D)$
  ⟨*proof*⟩

lemma *data-univ*: $data(univ(A), univ(A)) \subseteq univ(A)$
  ⟨*proof*⟩

lemma *data-subset-univ*:
  $[\![A \subseteq univ(C); B \subseteq univ(C)]\!] \Longrightarrow data(A, B) \subseteq univ(C)$
  ⟨*proof*⟩

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
$bt :: i \Rightarrow i$

datatype $bt(A) =$
$Lf \mid Br\ (a \in A,\ t1 \in bt(A),\ t2 \in bt(A))$

declare *bt.intros* [*simp*]

lemma *Br-neq-left*: $l \in bt(A) \Longrightarrow Br(x,\ l,\ r) \neq l$
⟨*proof*⟩

lemma *Br-iff*: $Br(a,\ l,\ r) = Br(a',\ l',\ r') \longleftrightarrow a = a' \wedge l = l' \wedge r = r'$
— Proving a freeness theorem.
⟨*proof*⟩

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

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

lemma *bt-mono*: $A \subseteq B \Longrightarrow bt(A) \subseteq bt(B)$
⟨*proof*⟩

lemma *bt-univ*: $bt(univ(A)) \subseteq univ(A)$
⟨*proof*⟩

lemma *bt-subset-univ*: $A \subseteq univ(B) \Longrightarrow bt(A) \subseteq univ(B)$
⟨*proof*⟩

lemma *bt-rec-type*:
$[\![t \in bt(A);$
$c \in C(Lf);$
$\bigwedge x\ y\ z\ r\ s.\ [\![x \in A;\ \ y \in bt(A);\ \ z \in bt(A);\ \ r \in C(y);\ \ s \in C(z)]\!] \Longrightarrow$
$h(x,\ y,\ z,\ r,\ s) \in C(Br(x,\ y,\ z))$
$]\!] \Longrightarrow bt\text{-}rec(c,\ h,\ t) \in C(t)$
— Type checking for recursor – example only; not really needed.
⟨*proof*⟩

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

consts $n\text{-}nodes :: i \Rightarrow i$
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*]: $t \in bt(A) \Longrightarrow n\text{-}nodes(t) \in nat$
⟨*proof*⟩

consts $n\text{-}nodes\text{-}aux :: i \Rightarrow i$
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*:
$t \in bt(A) \Longrightarrow k \in nat \Longrightarrow n\text{-}nodes\text{-}aux(t)\text{`}k = n\text{-}nodes(t)\ \#+\ k$
⟨*proof*⟩

definition
$n\text{-}nodes\text{-}tail :: i \Rightarrow i$ **where**
$n\text{-}nodes\text{-}tail(t) \equiv n\text{-}nodes\text{-}aux(t)\ \text{`}\ 0$

lemma $t \in bt(A) \Longrightarrow n\text{-}nodes\text{-}tail(t) = n\text{-}nodes(t)$
⟨*proof*⟩

2.3 Number of leaves

consts
$n\text{-}leaves :: i \Rightarrow i$
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*]: $t \in bt(A) \Longrightarrow n\text{-}leaves(t) \in nat$
⟨*proof*⟩

2.4 Reflecting trees

consts
$bt\text{-}reflect :: i \Rightarrow i$
primrec
$bt\text{-}reflect(Lf) = Lf$
$bt\text{-}reflect(Br(a, l, r)) = Br(a, bt\text{-}reflect(r), bt\text{-}reflect(l))$

lemma *bt-reflect-type* [*simp*]: $t \in bt(A) \Longrightarrow bt\text{-}reflect(t) \in bt(A)$
⟨*proof*⟩

Theorems about *n-leaves*.

lemma *n-leaves-reflect*: $t \in bt(A) \Longrightarrow n\text{-}leaves(bt\text{-}reflect(t)) = n\text{-}leaves(t)$
⟨*proof*⟩

lemma *n-leaves-nodes*: $t \in bt(A) \Longrightarrow n\text{-}leaves(t) = succ(n\text{-}nodes(t))$
⟨*proof*⟩

Theorems about *bt-reflect*.

lemma *bt-reflect-bt-reflect-ident*: $t \in bt(A) \Longrightarrow bt\text{-}reflect(bt\text{-}reflect(t)) = t$
⟨*proof*⟩

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* ∈ *A*, *l* ∈ *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*, λ*t g*. *term-case*(λ*x zs*. *d*(*x*, *zs*, *map*(λ*z*. *g*'*z*, *zs*)), *t*))

definition
  term-map :: [*i* ⇒ *i*, *i*] ⇒ *i* **where**
  term-map(*f*,*t*) ≡ *term-rec*(*t*, λ*x zs rs*. *Apply*(*f*(*x*), *rs*))

definition
  term-size :: *i* ⇒ *i* **where**
  term-size(*t*) ≡ *term-rec*(*t*, λ*x zs rs*. *succ*(*list-add*(*rs*)))

definition
  reflect :: *i* ⇒ *i* **where**
  reflect(*t*) ≡ *term-rec*(*t*, λ*x zs rs*. *Apply*(*x*, *rev*(*rs*)))

definition
  preorder :: *i* ⇒ *i* **where**
  preorder(*t*) ≡ *term-rec*(*t*, λ*x zs rs*. *Cons*(*x*, *flat*(*rs*)))

definition
  postorder :: *i* ⇒ *i* **where**
  postorder(*t*) ≡ *term-rec*(*t*, λ*x zs rs*. *flat*(*rs*) @ [*x*])

lemma *term-unfold*: *term*(*A*) = *A* ∗ *list*(*term*(*A*))
  ⟨*proof*⟩

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*))

$]\!] \Longrightarrow P(Apply(x,\ Cons(z,zs)))$
$]\!] \Longrightarrow P(t)$
— Induction on $term(A)$ followed by induction on *list*.
⟨*proof*⟩

lemma *term-induct-eqn* [*consumes 1*, *case-names Apply*]:
$[\![t \in term(A);$
$\bigwedge x\ zs.\ [\![x \in A;\ \ zs: list(term(A));\ \ map(f,zs) = map(g,zs)]\!] \Longrightarrow$
$f(Apply(x,zs)) = g(Apply(x,zs))$
$]\!] \Longrightarrow f(t) = g(t)$
— Induction on $term(A)$ to prove an equation.
⟨*proof*⟩

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

lemma *term-mono*: $A \subseteq B \Longrightarrow term(A) \subseteq term(B)$
⟨*proof*⟩

lemma *term-univ*: $term(univ(A)) \subseteq univ(A)$
— Easily provable by induction also
⟨*proof*⟩

lemma *term-subset-univ*: $A \subseteq univ(B) \Longrightarrow term(A) \subseteq univ(B)$
⟨*proof*⟩

lemma *term-into-univ*: $[\![t \in term(A);\ \ A \subseteq univ(B)]\!] \Longrightarrow t \in univ(B)$
⟨*proof*⟩

term-rec – by *Vset* recursion.

lemma *map-lemma*: $[\![l \in list(A);\ \ Ord(i);\ \ rank(l)<i]\!]$
$\Longrightarrow map(\lambda z.\ (\lambda x \in Vset(i).h(x))\ `\ z,\ l) = map(h,l)$
— *map* works correctly on the underlying list of terms.
⟨*proof*⟩

lemma *term-rec* [*simp*]: $ts \in list(A) \Longrightarrow$
$term\text{-}rec(Apply(a,ts),\ d) = d(a,\ ts,\ map\ (\lambda z.\ term\text{-}rec(z,d),\ ts))$
— Typing premise is necessary to invoke *map-lemma*.
⟨*proof*⟩

lemma *term-rec-type*:
assumes t: $t \in term(A)$
and a: $\bigwedge x\ zs\ r.\ [\![x \in A;\ \ zs: list(term(A));$
$r \in list(\bigcup t \in term(A).\ C(t))]\!]$
$\Longrightarrow d(x,\ zs,\ r): C(Apply(x,zs))$
shows $term\text{-}rec(t,d) \in C(t)$
— Slightly odd typing condition on r in the second premise!
⟨*proof*⟩

lemma *def-term-rec*:

$[\![\bigwedge t.\ j(t) \equiv \textit{term-rec}(t,d);\ \ ts\colon \textit{list}(A)]\!] \Longrightarrow$
$j(\textit{Apply}(a,ts)) = d(a,\ ts,\ \textit{map}(\lambda Z.\ j(Z),\ ts))$
$\langle \textit{proof} \rangle$

lemma *term-rec-simple-type* [*TC*]:
$[\![t \in \textit{term}(A);$
$\bigwedge x\ zs\ r.\ [\![x \in A;\ \ zs\colon \textit{list}(\textit{term}(A));\ \ r \in \textit{list}(C)]\!]$
$\Longrightarrow d(x,\ zs,\ r)\colon C$
$]\!] \Longrightarrow \textit{term-rec}(t,d) \in C$
$\langle \textit{proof} \rangle$

term-map.

lemma *term-map* [*simp*]:
$ts \in \textit{list}(A) \Longrightarrow$
$\textit{term-map}(f,\ \textit{Apply}(a,\ ts)) = \textit{Apply}(f(a),\ \textit{map}(\textit{term-map}(f),\ ts))$
$\langle \textit{proof} \rangle$

lemma *term-map-type* [*TC*]:
$[\![t \in \textit{term}(A);\ \ \bigwedge x.\ x \in A \Longrightarrow f(x)\colon B]\!] \Longrightarrow \textit{term-map}(f,t) \in \textit{term}(B)$
$\langle \textit{proof} \rangle$

lemma *term-map-type2* [*TC*]:
$t \in \textit{term}(A) \Longrightarrow \textit{term-map}(f,t) \in \textit{term}(\{f(u).\ u \in A\})$
$\langle \textit{proof} \rangle$

term-size.

lemma *term-size* [*simp*]:
$ts \in \textit{list}(A) \Longrightarrow \textit{term-size}(\textit{Apply}(a,\ ts)) = \textit{succ}(\textit{list-add}(\textit{map}(\textit{term-size},\ ts)))$
$\langle \textit{proof} \rangle$

lemma *term-size-type* [*TC*]: $t \in \textit{term}(A) \Longrightarrow \textit{term-size}(t) \in \textit{nat}$
$\langle \textit{proof} \rangle$

reflect.

lemma *reflect* [*simp*]:
$ts \in \textit{list}(A) \Longrightarrow \textit{reflect}(\textit{Apply}(a,\ ts)) = \textit{Apply}(a,\ \textit{rev}(\textit{map}(\textit{reflect},\ ts)))$
$\langle \textit{proof} \rangle$

lemma *reflect-type* [*TC*]: $t \in \textit{term}(A) \Longrightarrow \textit{reflect}(t) \in \textit{term}(A)$
$\langle \textit{proof} \rangle$

preorder.

lemma *preorder* [*simp*]:
$ts \in \textit{list}(A) \Longrightarrow \textit{preorder}(\textit{Apply}(a,\ ts)) = \textit{Cons}(a,\ \textit{flat}(\textit{map}(\textit{preorder},\ ts)))$
$\langle \textit{proof} \rangle$

lemma *preorder-type* [*TC*]: $t \in term(A) \Longrightarrow preorder(t) \in list(A)$
⟨*proof*⟩

postorder.

lemma *postorder* [*simp*]:
$ts \in list(A) \Longrightarrow postorder(Apply(a,\ ts)) = flat(map(postorder,\ ts))\ @\ [a]$
⟨*proof*⟩

lemma *postorder-type* [*TC*]: $t \in term(A) \Longrightarrow postorder(t) \in list(A)$
⟨*proof*⟩

Theorems about *term-map*.

declare *map-compose* [*simp*]

lemma *term-map-ident*: $t \in term(A) \Longrightarrow term\text{-}map(\lambda u.\ u,\ t) = t$
⟨*proof*⟩

lemma *term-map-compose*:
$t \in term(A) \Longrightarrow term\text{-}map(f,\ term\text{-}map(g,t)) = term\text{-}map(\lambda u.\ f(g(u)),\ t)$
⟨*proof*⟩

lemma *term-map-reflect*:
$t \in term(A) \Longrightarrow term\text{-}map(f,\ reflect(t)) = reflect(term\text{-}map(f,t))$
⟨*proof*⟩

Theorems about *term-size*.

lemma *term-size-term-map*: $t \in term(A) \Longrightarrow term\text{-}size(term\text{-}map(f,t)) = term\text{-}size(t)$
⟨*proof*⟩

lemma *term-size-reflect*: $t \in term(A) \Longrightarrow term\text{-}size(reflect(t)) = term\text{-}size(t)$
⟨*proof*⟩

lemma *term-size-length*: $t \in term(A) \Longrightarrow term\text{-}size(t) = length(preorder(t))$
⟨*proof*⟩

Theorems about *reflect*.

lemma *reflect-reflect-ident*: $t \in term(A) \Longrightarrow reflect(reflect(t)) = t$
⟨*proof*⟩

Theorems about preorder.

lemma *preorder-term-map*:
$t \in term(A) \Longrightarrow preorder(term\text{-}map(f,t)) = map(f,\ preorder(t))$
⟨*proof*⟩

lemma *preorder-reflect-eq-rev-postorder*:
$t \in term(A) \Longrightarrow preorder(reflect(t)) = rev(postorder(t))$

⟨*proof*⟩

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* ⇒ *i*
  maptree :: *i* ⇒ *i*
  maptree2 :: [*i*, *i*] ⇒ *i*

datatype *ntree*(*A*) = *Branch* (*a* ∈ *A*, *h* ∈ (⋃*n* ∈ *nat*. *n* −> *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* ∈ *A*, *h* ∈ *maptree*(*A*) −||> *maptree*(*A*))
  monos *FiniteFun-mono1* — Use monotonicity in BOTH args
  type-intros *FiniteFun-univ1* [*THEN subsetD*]

datatype *maptree2*(*A*, *B*) = *Sons2* (*a* ∈ *A*, *h* ∈ *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* ∈ *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* × (⋃*n* ∈ *nat*. *n* −> *ntree*(*A*))
  ⟨*proof*⟩

lemma *ntree-induct* [*consumes 1*, *case-names Branch*, *induct set*: *ntree*]:
  assumes *t*: *t* ∈ *ntree*(*A*)
    and *step*: ⋀*x n h*. ⟦*x* ∈ *A*; *n* ∈ *nat*; *h* ∈ *n* −> *ntree*(*A*); ∀ *i* ∈ *n*. *P*(*h*‘*i*)
⟧ ⟹ *P*(*Branch*(*x*,*h*))
  shows *P*(*t*)
  — A nicer induction rule than the standard one.
  ⟨*proof*⟩

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*: $\bigwedge x\ n\ h.\ [\![x \in A;\ \ n \in nat;\ \ h \in n -> ntree(A);\ \ f\ O\ h = g\ O\ h]\!] \Longrightarrow$
$f\ `\ Branch(x,h) = g\ `\ Branch(x,h)$
shows $f`t=g`t$
— Induction on $ntree(A)$ to prove an equation
⟨*proof*⟩

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

lemma *ntree-mono*: $A \subseteq B \Longrightarrow ntree(A) \subseteq ntree(B)$
⟨*proof*⟩

lemma *ntree-univ*: $ntree(univ(A)) \subseteq univ(A)$
— Easily provable by induction also
⟨*proof*⟩

lemma *ntree-subset-univ*: $A \subseteq univ(B) \Longrightarrow ntree(A) \subseteq univ(B)$
⟨*proof*⟩

ntree recursion.

lemma *ntree-rec-Branch*:
$function(h) \Longrightarrow$
$ntree\text{-}rec(b,\ Branch(x,h)) = b(x,\ h,\ \lambda i \in domain(h).\ ntree\text{-}rec(b,\ h`i))$
⟨*proof*⟩

lemma *ntree-copy-Branch* [*simp*]:
$function(h) \Longrightarrow$
$ntree\text{-}copy\ (Branch(x,\ h)) = Branch(x,\ \lambda i \in domain(h).\ ntree\text{-}copy\ (h`i))$
⟨*proof*⟩

lemma *ntree-copy-is-ident*: $z \in ntree(A) \Longrightarrow ntree\text{-}copy(z) = z$
⟨*proof*⟩

maptree

lemma *maptree-unfold*: $maptree(A) = A \times (maptree(A) -||> maptree(A))$
⟨*proof*⟩

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

⟨*proof*⟩

maptree2

lemma *maptree2-unfold*: $maptree2(A, B) = A \times (B -||> maptree2(A, B))$
⟨*proof*⟩

lemma *maptree2-induct* [*consumes 1*, *induct set*: *maptree2*]:
assumes *t*: $t \in maptree2(A, B)$
and *step*: $\bigwedge x\ n\ h.\ [\![x \in A;\ \ h \in B -||> maptree2(A,B);\ \ \forall y \in range(h).\ P(y)$
$]\!] \Longrightarrow P(Sons2(x,h))$
shows $P(t)$
⟨*proof*⟩

end

5 Trees and forests, a mutually recursive type definition

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

5.1 Datatype definition

consts
tree :: $i \Rightarrow i$
forest :: $i \Rightarrow i$
tree-forest :: $i \Rightarrow i$

datatype $tree(A) = Tcons\ (a \in A, f \in forest(A))$
and $forest(A) = Fnil \mid Fcons\ (t \in tree(A), f \in forest(A))$

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*: $tree(A) \equiv Part(tree\text{-}forest(A), Inl)$
⟨*proof*⟩

lemma *forest-def*: $forest(A) \equiv Part(tree\text{-}forest(A), Inr)$
⟨*proof*⟩

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

lemma *tree-subset-TF*: $tree(A) \subseteq tree\text{-}forest(A)$
⟨*proof*⟩

lemma *treeI* [*TC*]: $x \in tree(A) \Longrightarrow x \in tree\text{-}forest(A)$
⟨*proof*⟩

lemma *forest-subset-TF*: $forest(A) \subseteq tree\text{-}forest(A)$
⟨*proof*⟩

lemma *treeI'* [*TC*]: $x \in forest(A) \Longrightarrow x \in tree\text{-}forest(A)$
⟨*proof*⟩

lemma *TF-equals-Un*: $tree(A) \cup forest(A) = tree\text{-}forest(A)$
⟨*proof*⟩

lemma *tree-forest-unfold*:
$tree\text{-}forest(A) = (A \times forest(A)) + (\{0\} + tree(A) \times forest(A))$
— NOT useful, but interesting ...
⟨*proof*⟩

lemma *tree-forest-unfold'*:
$tree\text{-}forest(A) =$
$A \times Part(tree\text{-}forest(A), \lambda w.\ Inr(w)) +$
$\{0\} + Part(tree\text{-}forest(A), \lambda w.\ Inl(w)) * Part(tree\text{-}forest(A), \lambda w.\ Inr(w))$
⟨*proof*⟩

lemma *tree-unfold*: $tree(A) = \{Inl(x).\ x \in A \times forest(A)\}$
⟨*proof*⟩

lemma *forest-unfold*: $forest(A) = \{Inr(x).\ x \in \{0\} + tree(A)*forest(A)\}$
⟨*proof*⟩

Type checking for recursor: Not needed; possibly interesting?

lemma *TF-rec-type*:
$[\![z \in tree\text{-}forest(A);$
$\bigwedge x\ f\ r.\ [\![x \in A;\ \ f \in forest(A);\ \ r \in C(f)$
$]\!] \Longrightarrow b(x,f,r) \in C(Tcons(x,f));$
$c \in C(Fnil);$
$\bigwedge t\ f\ r1\ r2.\ [\![t \in tree(A);\ \ f \in forest(A);\ \ r1 \in C(t);\ r2 \in C(f)$
$]\!] \Longrightarrow d(t,f,r1,r2) \in C(Fcons(t,f))$
$]\!] \Longrightarrow tree\text{-}forest\text{-}rec(b,c,d,z) \in C(z)$
⟨*proof*⟩

lemma *tree-forest-rec-type*:
$[\![\bigwedge x\ f\ r.\ [\![x \in A;\ \ f \in forest(A);\ \ r \in D(f)$
$]\!] \Longrightarrow b(x,f,r) \in C(Tcons(x,f));$
$c \in D(Fnil);$
$\bigwedge t\ f\ r1\ r2.\ [\![t \in tree(A);\ \ f \in forest(A);\ \ r1 \in C(t);\ r2 \in D(f)$
$]\!] \Longrightarrow d(t,f,r1,r2) \in D(Fcons(t,f))$

$\rrbracket \Longrightarrow (\forall\, t \in \mathit{tree}(A).\quad \mathit{tree\text{-}forest\text{-}rec}(b,c,d,t) \in C(t)) \wedge$
$(\forall\, f \in \mathit{forest}(A).\ \mathit{tree\text{-}forest\text{-}rec}(b,c,d,f) \in D(f))$

— Mutually recursive version.

⟨*proof*⟩

5.2 Operations

consts

map :: $[i \Rightarrow i,\ i] \Rightarrow i$
size :: $i \Rightarrow i$
preorder :: $i \Rightarrow i$
list-of-TF :: $i \Rightarrow i$
of-list :: $i \Rightarrow i$
reflect :: $i \Rightarrow i$

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*]:
$z \in \mathit{tree\text{-}forest}(A) \Longrightarrow \mathit{list\text{-}of\text{-}TF}(z) \in \mathit{list}(\mathit{tree}(A))$
⟨*proof*⟩

lemma *of-list-type* [*TC*]: $l \in list(tree(A)) \Longrightarrow of\text{-}list(l) \in forest(A)$
⟨*proof*⟩

map.

lemma
assumes $\bigwedge x.\ x \in A \Longrightarrow h(x)\colon B$
shows *map-tree-type*: $t \in tree(A) \Longrightarrow map(h,t) \in tree(B)$
and *map-forest-type*: $f \in forest(A) \Longrightarrow map(h,f) \in forest(B)$
⟨*proof*⟩

size.

lemma *size-type* [*TC*]: $z \in tree\text{-}forest(A) \Longrightarrow size(z) \in nat$
⟨*proof*⟩

preorder.

lemma *preorder-type* [*TC*]: $z \in tree\text{-}forest(A) \Longrightarrow preorder(z) \in list(A)$
⟨*proof*⟩

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

lemma *forest-induct* [*consumes 1*, *case-names Fnil Fcons*]:
$[\![f \in forest(A);$
$R(Fnil);$
$\bigwedge t\ f.\ [\![t \in tree(A);\ \ f \in forest(A);\ \ R(f)]\!] \Longrightarrow R(Fcons(t,f))$
$]\!] \Longrightarrow R(f)$
— Essentially the same as list induction.
⟨*proof*⟩

lemma *forest-iso*: $f \in forest(A) \Longrightarrow of\text{-}list(list\text{-}of\text{-}TF(f)) = f$
⟨*proof*⟩

lemma *tree-list-iso*: $ts\colon list(tree(A)) \Longrightarrow list\text{-}of\text{-}TF(of\text{-}list(ts)) = ts$
⟨*proof*⟩

Theorems about *map*.

lemma *map-ident*: $z \in tree\text{-}forest(A) \Longrightarrow map(\lambda u.\ u,\ z) = z$
⟨*proof*⟩

lemma *map-compose*:
$z \in tree\text{-}forest(A) \Longrightarrow map(h,\ map(j,z)) = map(\lambda u.\ h(j(u)),\ z)$
⟨*proof*⟩

Theorems about *size*.

lemma *size-map*: $z \in tree\text{-}forest(A) \Longrightarrow size(map(h,z)) = size(z)$
⟨*proof*⟩

lemma *size-length*: $z \in$ *tree-forest*$(A) \Longrightarrow$ *size*$(z) =$ *length*$($*preorder*$(z))$
⟨*proof*⟩

Theorems about *preorder*.

lemma *preorder-map*:
$z \in$ *tree-forest*$(A) \Longrightarrow$ *preorder*$($*map*$(h,z)) =$ *List.map*$(h,$ *preorder*$(z))$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *brouwer-induct2* [*consumes 1*, *case-names Zero Suc Lim*]:
assumes *b*: $b \in$ *brouwer*
and *cases*:
$P($*Zero*$)$
$\bigwedge b.\ [\![b \in$ *brouwer*$;\ P(b)]\!] \Longrightarrow P($*Suc*$(b))$
$\bigwedge h.\ [\![h \in$ *nat* −> *brouwer*$;\ \forall i \in$ *nat*$.\ P(h`i)]\!] \Longrightarrow P($*Lim*$(h))$
shows $P(b)$
— A nicer induction rule than the standard one.
⟨*proof*⟩

6.2 The Martin-Löf wellordering type

consts
Well :: $[i, i \Rightarrow i] \Rightarrow i$

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*: $Well(A, B) = (\sum x \in A.\ B(x) \;{-}{>}\; Well(A, B))$
  ⟨*proof*⟩

lemma *Well-induct2* [*consumes 1*, *case-names step*]:
  assumes *w*: $w \in Well(A, B)$
    and *step*: $\bigwedge a\ f.\ [\![a \in A;\ \ f \in B(a) \;{-}{>}\; Well(A,B);\ \ \forall y \in B(a).\ P(f`y)]\!] \Longrightarrow P(Sup(a,f))$
  shows $P(w)$
  — A nicer induction rule than the standard one.
  ⟨*proof*⟩

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.
  ⟨*proof*⟩

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 :: $i \Rightarrow i$

inductive
  domains $domino \subseteq Pow(nat \times nat)$
  intros
    horiz: $[\![i \in nat;\ \ j \in nat]\!] \Longrightarrow \{\langle i,j\rangle, <i,succ(j)>\} \in domino$
    vertl: $[\![i \in nat;\ \ j \in nat]\!] \Longrightarrow \{\langle i,j\rangle, <succ(i),j>\} \in domino$
  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: $[\![a \in A;\ \ t \in tiling(A);\ \ a \cap t = 0]\!] \Longrightarrow a \cup t \in tiling(A)$
  type-intros *empty-subsetI Union-upper Un-least PowI*
  type-elims *PowD* [*elim-format*]

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

$evnodd(A,b) \equiv \{z \in A.\ \exists\, i\ j.\ z = \langle i,j\rangle \wedge (i \#+ j)\ mod\ 2 = b\}$

7.1 Basic properties of evnodd

lemma *evnodd-iff*: $\langle i,j\rangle: evnodd(A,b) \longleftrightarrow \langle i,j\rangle: A \wedge (i\#+j)\ mod\ 2 = b$
⟨*proof*⟩

lemma *evnodd-subset*: $evnodd(A,\ b) \subseteq A$
⟨*proof*⟩

lemma *Finite-evnodd*: $Finite(X) \Longrightarrow Finite(evnodd(X,b))$
⟨*proof*⟩

lemma *evnodd-Un*: $evnodd(A \cup B,\ b) = evnodd(A,b) \cup evnodd(B,b)$
⟨*proof*⟩

lemma *evnodd-Diff*: $evnodd(A - B,\ b) = evnodd(A,b) - evnodd(B,b)$
⟨*proof*⟩

lemma *evnodd-cons* [*simp*]:
$evnodd(cons(\langle i,j\rangle,C),\ b) =$
$(if\ (i\#+j)\ mod\ 2 = b\ then\ cons(\langle i,j\rangle,\ evnodd(C,b))\ else\ evnodd(C,b))$
⟨*proof*⟩

lemma *evnodd-0* [*simp*]: $evnodd(0,\ b) = 0$
⟨*proof*⟩

7.2 Dominoes

lemma *domino-Finite*: $d \in domino \Longrightarrow Finite(d)$
⟨*proof*⟩

lemma *domino-singleton*:
$[\![d \in domino;\ b<2]\!] \Longrightarrow \exists\, i'\ j'.\ evnodd(d,b) = \{<i',j'>\}$
⟨*proof*⟩

7.3 Tilings

The union of two disjoint tilings is a tiling

lemma *tiling-UnI*:
$t \in tiling(A) \Longrightarrow u \in tiling(A) \Longrightarrow t \cap u = 0 \Longrightarrow t \cup u \in tiling(A)$
⟨*proof*⟩

lemma *tiling-domino-Finite*: $t \in tiling(domino) \Longrightarrow Finite(t)$
⟨*proof*⟩

lemma *tiling-domino-0-1*: $t \in tiling(domino) \Longrightarrow |evnodd(t,0)| = |evnodd(t,1)|$
⟨*proof*⟩

lemma *dominoes-tile-row*:
⟦*i* ∈ *nat*; *n* ∈ *nat*⟧ ⟹ {*i*} ∗ (*n* #+ *n*) ∈ *tiling*(*domino*)
⟨*proof*⟩

lemma *dominoes-tile-matrix*:
⟦*m* ∈ *nat*; *n* ∈ *nat*⟧ ⟹ *m* ∗ (*n* #+ *n*) ∈ *tiling*(*domino*)
⟨*proof*⟩

lemma *eq-lt-E*: ⟦*x*=*y*; *x*<*y*⟧ ⟹ *P*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*](λ*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*

$\langle proof \rangle$

lemma *cons-lemma2*: $[\![cons(x,\ B)=cons(y,\ C);\ x\neq y;\ x\notin B;\ y\notin C]\!]$
$\Longrightarrow\ B\ -\ \{y\}\ =\ C-\{x\}\ \wedge\ x\in C\ \wedge\ y\in B$
$\langle proof \rangle$

lemma *fold-set-mono-lemma*:
$\langle C,\ x\rangle\ \in\ fold\text{-}set(A,\ B,\ f,\ e)$
$\Longrightarrow \forall D.\ A<=D \longrightarrow \langle C,\ x\rangle\ \in\ fold\text{-}set(D,\ B,\ f,\ e)$
$\langle proof \rangle$

lemma *fold-set-mono*: $C<=A \Longrightarrow fold\text{-}set(C,\ B,\ f,\ e) \subseteq fold\text{-}set(A,\ B,\ f,\ e)$
$\langle proof \rangle$

lemma *fold-set-lemma*:
$\langle C,\ x\rangle\in fold\text{-}set(A,\ B,\ f,\ e) \Longrightarrow \langle C,\ x\rangle\in fold\text{-}set(C,\ B,\ f,\ e)\ \wedge\ C<=A$
$\langle proof \rangle$

lemma *Diff1-fold-set*:
$[\![<C-\{x\},y>\ \in\ fold\text{-}set(A,\ B,\ f,e);\ \ x\in C;\ x\in A;\ f(x,\ y):B]\!]$
$\Longrightarrow\ <C,\ f(x,\ y)>\ \in\ fold\text{-}set(A,\ B,\ f,\ e)$
$\langle proof \rangle$

locale *fold-typing* =
fixes A **and** B **and** e **and** f
assumes *ftype* [*intro,simp*]: $[\![x\ \in\ A;\ y\ \in\ B]\!] \Longrightarrow f(x,y)\ \in\ B$
and *etype* [*intro,simp*]: $e\ \in\ B$
and *fcomm*: $[\![x\ \in\ A;\ y\ \in\ A;\ z\ \in\ B]\!] \Longrightarrow f(x,\ f(y,\ z))=f(y,\ f(x,\ z))$

lemma (**in** *fold-typing*) *Fin-imp-fold-set*:
$C\in Fin(A) \Longrightarrow (\exists x.\ \langle C,\ x\rangle\ \in\ fold\text{-}set(A,\ B,\ f,e))$
$\langle proof \rangle$

lemma *Diff-sing-imp*:
$[\![C\ -\ \{b\}\ =\ D\ -\ \{a\};\ a\ \neq\ b;\ b\ \in\ C]\!] \Longrightarrow C\ =\ cons(b,D)\ -\ \{a\}$
$\langle proof \rangle$

lemma (**in** *fold-typing*) *fold-set-determ-lemma* [*rule-format*]:
$n\in nat$
$\Longrightarrow \forall C.\ |C|<n \longrightarrow$
$(\forall x.\ \langle C,\ x\rangle\ \in\ fold\text{-}set(A,\ B,\ f,e)\longrightarrow$
$(\forall y.\ \langle C,\ y\rangle\ \in\ fold\text{-}set(A,\ B,\ f,e)\ \longrightarrow\ y=x))$
$\langle proof \rangle$

lemma (**in** *fold-typing*) *fold-set-determ*:

$[\![\langle C,\ x\rangle \in \mathit{fold\text{-}set}(A,\ B,\ f,\ e);$
$\quad \langle C,\ y\rangle \in \mathit{fold\text{-}set}(A,\ B,\ f,\ e)]\!] \Longrightarrow y{=}x$
$\langle \mathit{proof}\rangle$

lemma (**in** *fold-typing*) *fold-equality*:
$\langle C,y\rangle \in \mathit{fold\text{-}set}(A,B,f,e) \Longrightarrow \mathit{fold}[B](f,e,C) = y$
$\langle \mathit{proof}\rangle$

lemma *fold-0* [*simp*]: $e \in B \Longrightarrow \mathit{fold}[B](f,e,0) = e$
$\langle \mathit{proof}\rangle$

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

lemma (**in** *fold-typing*) *fold-set-imp-cons*:
$[\![\langle C,\ y\rangle \in \mathit{fold\text{-}set}(C,\ B,\ f,\ e);\ C \in \mathit{Fin}(A);\ c \in A;\ c \notin C]\!]$
$\Longrightarrow <\mathit{cons}(c,\ C),\ f(c,y)> \in \mathit{fold\text{-}set}(\mathit{cons}(c,\ C),\ B,\ f,\ e)$
$\langle \mathit{proof}\rangle$

lemma (**in** *fold-typing*) *fold-cons-lemma* [*rule-format*]:
$[\![C \in \mathit{Fin}(A);\ c \in A;\ c \notin C]\!]$
$\Longrightarrow <\mathit{cons}(c,\ C),\ v> \in \mathit{fold\text{-}set}(\mathit{cons}(c,\ C),\ B,\ f,\ e) \longleftrightarrow$
$(\exists\, y.\ \langle C,\ y\rangle \in \mathit{fold\text{-}set}(C,\ B,\ f,\ e) \wedge v = f(c,\ y))$
$\langle \mathit{proof}\rangle$

lemma (**in** *fold-typing*) *fold-cons*:
$[\![C \in \mathit{Fin}(A);\ c \in A;\ c \notin C]\!]$
$\Longrightarrow \mathit{fold}[B](f,\ e,\ \mathit{cons}(c,\ C)) = f(c,\ \mathit{fold}[B](f,\ e,\ C))$
$\langle \mathit{proof}\rangle$

lemma (**in** *fold-typing*) *fold-type* [*simp*,*TC*]:
$C \in \mathit{Fin}(A) \Longrightarrow \mathit{fold}[B](f,e,C){:}B$
$\langle \mathit{proof}\rangle$

lemma (**in** *fold-typing*) *fold-commute* [*rule-format*]:
$[\![C \in \mathit{Fin}(A);\ c \in A]\!]$
$\Longrightarrow (\forall\, y \in B.\ f(c,\ \mathit{fold}[B](f,\ y,\ C)) = \mathit{fold}[B](f,\ f(c,\ y),\ C))$
$\langle \mathit{proof}\rangle$

lemma (**in** *fold-typing*) *fold-nest-Un-Int*:
$[\![C \in \mathit{Fin}(A);\ D \in \mathit{Fin}(A)]\!]$
$\Longrightarrow \mathit{fold}[B](f,\ \mathit{fold}[B](f,\ e,\ D),\ C) =$
$\mathit{fold}[B](f,\ \mathit{fold}[B](f,\ e,\ (C \cap D)),\ C \cup D)$
$\langle \mathit{proof}\rangle$

lemma (**in** *fold-typing*) *fold-nest-Un-disjoint*:
$[\![C \in \mathit{Fin}(A);\ D \in \mathit{Fin}(A);\ C \cap D = 0]\!]$
$\Longrightarrow \mathit{fold}[B](f,e,C \cup D) = \mathit{fold}[B](f,\ \mathit{fold}[B](f,e,D),\ C)$
$\langle \mathit{proof}\rangle$

lemma *Finite-cons-lemma*: $Finite(C) \Longrightarrow C \in Fin(cons(c,\ C))$
⟨*proof*⟩

7.4 The Operator *setsum*

lemma *setsum-0* [*simp*]: $setsum(g,\ 0) = \#0$
⟨*proof*⟩

lemma *setsum-cons* [*simp*]:
$Finite(C) \Longrightarrow$
$setsum(g,\ cons(c,C)) =$
$(if\ c \in C\ then\ setsum(g,C)\ else\ g(c)\ \$+\ setsum(g,C))$
⟨*proof*⟩

lemma *setsum-K0*: $setsum((\lambda i.\ \#0),\ C) = \#0$
⟨*proof*⟩

lemma *setsum-Un-Int*:
$[\![Finite(C);\ Finite(D)]\!]$
$\Longrightarrow setsum(g,\ C \cup D)\ \$+\ setsum(g,\ C \cap D)$
$= setsum(g,\ C)\ \$+\ setsum(g,\ D)$
⟨*proof*⟩

lemma *setsum-type* [*simp,TC*]: $setsum(g,\ C):int$
⟨*proof*⟩

lemma *setsum-Un-disjoint*:
$[\![Finite(C);\ Finite(D);\ C \cap D = 0]\!]$
$\Longrightarrow setsum(g,\ C \cup D) = setsum(g,\ C)\ \$+\ setsum(g,D)$
⟨*proof*⟩

lemma *Finite-RepFun* [*rule-format* (*no-asm*)]:
$Finite(I) \Longrightarrow (\forall i \in I.\ Finite(C(i))) \longrightarrow Finite(RepFun(I,\ C))$
⟨*proof*⟩

lemma *setsum-UN-disjoint* [*rule-format* (*no-asm*)]:
$Finite(I)$
$\Longrightarrow (\forall i \in I.\ Finite(C(i))) \longrightarrow$
$(\forall i \in I.\ \forall j \in I.\ i \neq j \longrightarrow C(i) \cap C(j) = 0) \longrightarrow$
$setsum(f,\ \bigcup i \in I.\ C(i)) = setsum\ (\lambda i.\ setsum(f,\ C(i)),\ I)$
⟨*proof*⟩

lemma *setsum-addf*: $setsum(\lambda x.\ f(x)\ \$+\ g(x),C) = setsum(f,\ C)\ \$+\ setsum(g,\ C)$
⟨*proof*⟩

lemma *fold-set-cong*:
$[\![A=A'; B=B'; e=e'; (\forall x\in A'. \forall y\in B'. f(x,y) = f'(x,y))]\!]$
$\Longrightarrow$ *fold-set*(A,B,f,e) = *fold-set*(A',B',f',e')
⟨*proof*⟩

lemma *fold-cong*:
$[\![B=B'; A=A'; e=e';$
$\bigwedge x\ y.\ [\![x\in A'; y\in B']\!] \Longrightarrow f(x,y) = f'(x,y)]\!] \Longrightarrow$
fold$[B](f,e,A)$ = *fold*$[B'](f', e', A')$
⟨*proof*⟩

lemma *setsum-cong*:
$[\![A=B; \bigwedge x.\ x\in B \Longrightarrow f(x) = g(x)]\!] \Longrightarrow$
setsum(f, A) = *setsum*(g, B)
⟨*proof*⟩

lemma *setsum-Un*:
$[\![$*Finite*(A); *Finite*$(B)]\!]$
$\Longrightarrow$ *setsum*$(f, A \cup B)$ =
setsum(f, A) \$+ *setsum*(f, B) \$− *setsum*$(f, A \cap B)$
⟨*proof*⟩

lemma *setsum-zneg-or-0* [*rule-format* (*no-asm*)]:
Finite$(A) \Longrightarrow (\forall x\in A.\ g(x)$ \$≤ #*0*$) \longrightarrow$ *setsum*(g, A) \$≤ #*0*
⟨*proof*⟩

lemma *setsum-succD-lemma* [*rule-format*]:
Finite(A)
$\Longrightarrow \forall n\in nat.$ *setsum*(f,A) = \$# *succ*$(n) \longrightarrow (\exists a\in A.$ #*0* \$< $f(a))$
⟨*proof*⟩

lemma *setsum-succD*:
$[\![$*setsum*(f, A) = \$# *succ*(n); $n\in nat]\!] \Longrightarrow \exists a\in A.$ #*0* \$< $f(a)$
⟨*proof*⟩

lemma *g-zpos-imp-setsum-zpos* [*rule-format*]:
Finite$(A) \Longrightarrow (\forall x\in A.$ #*0* \$≤ $g(x)) \longrightarrow$ #*0* \$≤ *setsum*(g, A)
⟨*proof*⟩

lemma *g-zpos-imp-setsum-zpos2* [*rule-format*]:
$[\![$*Finite*(A); $\forall x.$ #*0* \$≤ $g(x)]\!] \Longrightarrow$ #*0* \$≤ *setsum*(g, A)
⟨*proof*⟩

lemma *g-zspos-imp-setsum-zspos* [*rule-format*]:
Finite(A)
$\Longrightarrow (\forall x\in A.$ #*0* \$< $g(x)) \longrightarrow A \neq 0 \longrightarrow$ (#*0* \$< *setsum*(g, A))
⟨*proof*⟩

lemma *setsum-Diff* [*rule-format*]:
$Finite(A) \Longrightarrow \forall a.\ M(a) = \#0 \longrightarrow setsum(M,\ A) = setsum(M,\ A-\{a\})$
⟨*proof*⟩

end

8 The accessible part of a relation

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

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

consts
$acc :: i \Rightarrow i$

inductive
domains $acc(r) \subseteq field(r)$
intros
vimage: $[\![r-``\{a\}: Pow(acc(r));\ a \in field(r)]\!] \Longrightarrow a \in acc(r)$
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*: $[\![\bigwedge b.\ \langle b,a\rangle{:}r \Longrightarrow b \in acc(r);\ \ a \in field(r)]\!] \Longrightarrow a \in acc(r)$
⟨*proof*⟩

lemma *acc-downward*: $[\![b \in acc(r);\ \ \langle a,b\rangle{:}\ r]\!] \Longrightarrow a \in acc(r)$
⟨*proof*⟩

lemma *acc-induct* [*consumes 1*, *case-names vimage*, *induct set*: *acc*]:
$[\![a \in acc(r);$
$\bigwedge x.\ [\![x \in acc(r);\ \ \forall y.\ \langle y,x\rangle{:}r \longrightarrow P(y)]\!] \Longrightarrow P(x)$
$]\!] \Longrightarrow P(a)$
⟨*proof*⟩

lemma *wf-on-acc*: $wf[acc(r)](r)$
⟨*proof*⟩

lemma *acc-wfI*: $field(r) \subseteq acc(r) \Longrightarrow wf(r)$
⟨*proof*⟩

lemma *acc-wfD*: $wf(r) \Longrightarrow field(r) \subseteq acc(r)$
⟨*proof*⟩

lemma *wf-acc-iff*: $wf(r) \longleftrightarrow field(r) \subseteq acc(r)$
⟨*proof*⟩

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**

$\{\#a\#\} \equiv \{\langle a,\ 1\rangle\}$

definition
$MCollect :: [i,\ i{\Rightarrow}o] \Rightarrow i$ **where**
$MCollect(M,\ P) \equiv funrestrict(M,\ \{x \in mset\text{-}of(M).\ P(x)\})$

definition

$mcount :: [i,\ i] \Rightarrow i$ **where**
$mcount(M,\ a) \equiv if\ a \in mset\text{-}of(M)\ then\ \ M`a\ else\ 0$

definition
$msize :: i \Rightarrow i$ **where**
$msize(M) \equiv setsum(\lambda a.\ \$\#\ mcount(M{,}a),\ mset\text{-}of(M))$

abbreviation
$melem :: [i{,}i] \Rightarrow o$ (‹(-/ :# -)› [50, 51] 50) **where**
$a :\#\ M \equiv a \in mset\text{-}of(M)$

syntax
$\text{-}MColl :: [pttrn,\ i,\ o] \Rightarrow i$ (‹(1{# - ∈ -./ -#})›)
translations
$\{\#x \in M.\ P\#\} == CONST\ MCollect(M,\ \lambda x.\ P)$

definition

$multirel1 :: [i{,}i]{\Rightarrow}i$ **where**
$multirel1(A,\ r) \equiv$
$\{\langle M,\ N\rangle \in Mult(A){*}Mult(A).$
$\exists\, a \in A.\ \exists\, M0 \in Mult(A).\ \exists\, K \in Mult(A).$
$N{=}M0\ {+}\#\ \{\#a\#\} \wedge M{=}M0\ {+}\#\ K \wedge (\forall\, b \in mset\text{-}of(K).\ \langle b{,}a\rangle \in r)\}$

definition
$multirel :: [i,\ i] \Rightarrow i$ **where**
$multirel(A,\ r) \equiv multirel1(A,\ r)\hat{}+$

definition
$omultiset :: i \Rightarrow o$ **where**
$omultiset(M) \equiv \exists\, i.\ Ord(i) \wedge M \in Mult(field(Memrel(i)))$

definition
$mless :: [i,\ i] \Rightarrow o$ (**infixl** ‹<#› 50) **where**
$M <\#\ N \equiv\ \exists\, i.\ Ord(i) \wedge \langle M,\ N\rangle \in multirel(field(Memrel(i)),\ Memrel(i))$

definition

mle :: [*i*, *i*] ⇒ *o* (**infixl** ‹<#=› *50*) **where**
$M <\#= N \equiv (omultiset(M) \wedge M = N) \mid M <\# N$

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

lemma *funrestrict-subset*: $[\![f \in Pi(C,B);\ A \subseteq C]\!] \Longrightarrow funrestrict(f,A) \subseteq f$
⟨*proof*⟩

lemma *funrestrict-type*:
$[\![\bigwedge x.\ x \in A \Longrightarrow f`x \in B(x)]\!] \Longrightarrow funrestrict(f,A) \in Pi(A,B)$
⟨*proof*⟩

lemma *funrestrict-type2*: $[\![f \in Pi(C,B);\ A \subseteq C]\!] \Longrightarrow funrestrict(f,A) \in Pi(A,B)$
⟨*proof*⟩

lemma *funrestrict* [*simp*]: $a \in A \Longrightarrow funrestrict(f,A)\ `\ a = f`a$
⟨*proof*⟩

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

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

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

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

A useful simplification rule

lemma *multiset-fun-iff*:
$(f \in A -> nat-\{0\}) \longleftrightarrow f \in A->nat \wedge (\forall a \in A.\ f`a \in nat \wedge 0 < f`a)$
⟨*proof*⟩

lemma *multiset-into-Mult*: $[\![multiset(M);\ mset\text{-}of(M) \subseteq A]\!] \Longrightarrow M \in Mult(A)$
⟨*proof*⟩

lemma *Mult-into-multiset*: $M \in Mult(A) \Longrightarrow multiset(M) \wedge mset\text{-}of(M) \subseteq A$
⟨*proof*⟩

lemma *Mult-iff-multiset*: $M \in Mult(A) \longleftrightarrow multiset(M) \wedge mset\text{-}of(M) \subseteq A$
⟨*proof*⟩

lemma *multiset-iff-Mult-mset-of*: $multiset(M) \longleftrightarrow M \in Mult(mset\text{-}of(M))$
⟨*proof*⟩

The *multiset* operator

lemma *multiset-0* [*simp*]: *multiset*(*0*)
⟨*proof*⟩

The *mset-of* operator

lemma *multiset-set-of-Finite* [*simp*]: *multiset*(*M*) $\Longrightarrow$ *Finite*(*mset-of*(*M*))
⟨*proof*⟩

lemma *mset-of-0* [*iff*]: *mset-of*(*0*) = *0*
⟨*proof*⟩

lemma *mset-is-0-iff*: *multiset*(*M*) $\Longrightarrow$ *mset-of*(*M*)=*0* $\longleftrightarrow$ *M*=*0*
⟨*proof*⟩

lemma *mset-of-single* [*iff*]: *mset-of*({#*a*#}) = {*a*}
⟨*proof*⟩

lemma *mset-of-union* [*iff*]: *mset-of*(*M* +# *N*) = *mset-of*(*M*) ∪ *mset-of*(*N*)
⟨*proof*⟩

lemma *mset-of-diff* [*simp*]: *mset-of*(*M*)⊆*A* $\Longrightarrow$ *mset-of*(*M* −# *N*) ⊆ *A*
⟨*proof*⟩

lemma *msingle-not-0* [*iff*]: {#*a*#} ≠ *0* ∧ *0* ≠ {#*a*#}
⟨*proof*⟩

lemma *msingle-eq-iff* [*iff*]: ({#*a*#} = {#*b*#}) $\longleftrightarrow$ (*a* = *b*)
⟨*proof*⟩

lemma *msingle-multiset* [*iff*,*TC*]: *multiset*({#*a*#})
⟨*proof*⟩

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

lemma *normalize-idem* [*simp*]: *normalize*(*normalize*(*f*)) = *normalize*(*f*)
⟨*proof*⟩

lemma *normalize-multiset* [*simp*]: *multiset*(*M*) $\Longrightarrow$ *normalize*(*M*) = *M*
⟨*proof*⟩

lemma *multiset-normalize* [*simp*]: *multiset*(*normalize*(*f*))
⟨*proof*⟩

lemma *munion-multiset* [*simp*]: $[\![multiset(M);\ multiset(N)]\!] \Longrightarrow multiset(M +\# N)$
$\langle proof \rangle$

lemma *mdiff-multiset* [*simp*]: $multiset(M -\# N)$
$\langle proof \rangle$

lemma *munion-0* [*simp*]: $multiset(M) \Longrightarrow M +\# 0 = M \wedge 0 +\# M = M$
$\langle proof \rangle$

lemma *munion-commute*: $M +\# N = N +\# M$
$\langle proof \rangle$

lemma *munion-assoc*: $(M +\# N) +\# K = M +\# (N +\# K)$
$\langle proof \rangle$

lemma *munion-lcommute*: $M +\# (N +\# K) = N +\# (M +\# K)$
$\langle proof \rangle$

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

lemma *mdiff-self-eq-0* [*simp*]: $M -\# M = 0$
$\langle proof \rangle$

lemma *mdiff-0* [*simp*]: $0 -\# M = 0$
$\langle proof \rangle$

lemma *mdiff-0-right* [*simp*]: $multiset(M) \Longrightarrow M -\# 0 = M$
$\langle proof \rangle$

lemma *mdiff-union-inverse2* [*simp*]: $multiset(M) \Longrightarrow M +\# \{\#a\#\} -\# \{\#a\#\} = M$
$\langle proof \rangle$

lemma *mcount-type* [*simp*,*TC*]: $multiset(M) \Longrightarrow mcount(M,\ a) \in nat$
$\langle proof \rangle$

lemma *mcount-0* [*simp*]: $mcount(0,\ a) = 0$

⟨*proof*⟩

lemma *mcount-single* [*simp*]: $mcount(\{\#b\#\}, a) = (if\ a{=}b\ then\ 1\ else\ 0)$
⟨*proof*⟩

lemma *mcount-union* [*simp*]: $[\![multiset(M); multiset(N)]\!]$
$\Longrightarrow mcount(M +\# N, a) = mcount(M, a) \#+ mcount\ (N, a)$
⟨*proof*⟩

lemma *mcount-diff* [*simp*]:
$multiset(M) \Longrightarrow mcount(M -\# N, a) = mcount(M, a) \#- mcount(N, a)$
⟨*proof*⟩

lemma *mcount-elem*: $[\![multiset(M); a \in mset\text{-}of(M)]\!] \Longrightarrow 0 < mcount(M, a)$
⟨*proof*⟩

lemma *msize-0* [*simp*]: $msize(0) = \#0$
⟨*proof*⟩

lemma *msize-single* [*simp*]: $msize(\{\#a\#\}) = \#1$
⟨*proof*⟩

lemma *msize-type* [*simp,TC*]: $msize(M) \in int$
⟨*proof*⟩

lemma *msize-zpositive*: $multiset(M) \Longrightarrow \#0\ \$\leq msize(M)$
⟨*proof*⟩

lemma *msize-int-of-nat*: $multiset(M) \Longrightarrow \exists n \in nat.\ msize(M) = \$\#\ n$
⟨*proof*⟩

lemma *not-empty-multiset-imp-exist*:
$[\![M \neq 0; multiset(M)]\!] \Longrightarrow \exists a \in mset\text{-}of(M).\ 0 < mcount(M, a)$
⟨*proof*⟩

lemma *msize-eq-0-iff*: $multiset(M) \Longrightarrow msize(M) = \#0 \longleftrightarrow M = 0$
⟨*proof*⟩

lemma *setsum-mcount-Int*:
$Finite(A) \Longrightarrow setsum(\lambda a.\ \$\#\ mcount(N, a), A \cap mset\text{-}of(N))$
$= setsum(\lambda a.\ \$\#\ mcount(N, a), A)$
⟨*proof*⟩

lemma *msize-union* [*simp*]:
$[\![multiset(M); multiset(N)]\!] \Longrightarrow msize(M +\# N) = msize(M)\ \$+\ msize(N)$
⟨*proof*⟩

lemma *msize-eq-succ-imp-elem*: $[\![msize(M)= \$\#\ succ(n);\ n \in nat]\!] \Longrightarrow \exists a.\ a \in mset\text{-}of(M)$
⟨*proof*⟩

lemma *equality-lemma*:
$[\![multiset(M);\ multiset(N);\ \forall a.\ mcount(M,\ a)=mcount(N,\ a)]\!]$
$\Longrightarrow mset\text{-}of(M)=mset\text{-}of(N)$
⟨*proof*⟩

lemma *multiset-equality*:
$[\![multiset(M);\ multiset(N)]\!]\Longrightarrow M=N\longleftrightarrow(\forall a.\ mcount(M,\ a)=mcount(N,\ a))$
⟨*proof*⟩

lemma *munion-eq-0-iff* [*simp*]: $[\![multiset(M);\ multiset(N)]\!]\Longrightarrow(M +\#\ N =0) \longleftrightarrow (M=0 \wedge N=0)$
⟨*proof*⟩

lemma *empty-eq-munion-iff* [*simp*]: $[\![multiset(M);\ multiset(N)]\!]\Longrightarrow(0=M +\#\ N) \longleftrightarrow (M=0 \wedge N=0)$
⟨*proof*⟩

lemma *munion-right-cancel* [*simp*]:
$[\![multiset(M);\ multiset(N);\ multiset(K)]\!]\Longrightarrow(M +\#\ K = N +\#\ K)\longleftrightarrow(M=N)$
⟨*proof*⟩

lemma *munion-left-cancel* [*simp*]:
$[\![multiset(K);\ multiset(M);\ multiset(N)]\!] \Longrightarrow(K +\#\ M = K +\#\ N) \longleftrightarrow (M = N)$
⟨*proof*⟩

lemma *nat-add-eq-1-cases*: $[\![m \in nat;\ n \in nat]\!] \Longrightarrow (m \#+\ n = 1) \longleftrightarrow (m=1 \wedge n=0) \mid (m=0 \wedge n=1)$
⟨*proof*⟩

lemma *munion-is-single*:
$[\![multiset(M);\ multiset(N)]\!]$
$\Longrightarrow (M +\#\ N = \{\#a\#\}) \longleftrightarrow (M=\{\#a\#\} \wedge N=0) \mid (M = 0 \wedge N = \{\#a\#\})$
⟨*proof*⟩

lemma *msingle-is-union*: $[\![multiset(M);\ multiset(N)]\!]$
$\Longrightarrow (\{\#a\#\} = M +\#\ N) \longleftrightarrow (\{\#a\#\} = M \wedge N=0 \mid M = 0 \wedge \{\#a\#\} = N)$
⟨*proof*⟩

lemma *setsum-decr*:
Finite(*A*)
$\Longrightarrow$ ($\forall$ *M*. *multiset*(*M*) $\longrightarrow$
($\forall$ *a* $\in$ *mset-of*(*M*). *setsum*(λ*z*. \$# *mcount*(*M*(*a*:=*M*‘*a* #− *1*), *z*), *A*) =
(*if a* $\in$ *A then setsum*(λ*z*. \$# *mcount*(*M*, *z*), *A*) \$− #*1*
else setsum(λ*z*. \$# *mcount*(*M*, *z*), *A*))))
⟨*proof*⟩

lemma *setsum-decr2*:
Finite(*A*)
$\Longrightarrow$ $\forall$ *M*. *multiset*(*M*) $\longrightarrow$ ($\forall$ *a* $\in$ *mset-of*(*M*).
setsum(λ*x*. \$# *mcount*(*funrestrict*(*M*, *mset-of*(*M*)−{*a*}), *x*), *A*) =
(*if a* $\in$ *A then setsum*(λ*x*. \$# *mcount*(*M*, *x*), *A*) \$− \$# *M*‘*a*
else setsum(λ*x*. \$# *mcount*(*M*, *x*), *A*)))
⟨*proof*⟩

lemma *setsum-decr3*: ⟦*Finite*(*A*); *multiset*(*M*); *a* $\in$ *mset-of*(*M*)⟧
$\Longrightarrow$ *setsum*(λ*x*. \$# *mcount*(*funrestrict*(*M*, *mset-of*(*M*)−{*a*}), *x*), *A* − {*a*})
=
(*if a* $\in$ *A then setsum*(λ*x*. \$# *mcount*(*M*, *x*), *A*) \$− \$# *M*‘*a*
else setsum(λ*x*. \$# *mcount*(*M*, *x*), *A*))
⟨*proof*⟩

lemma *nat-le-1-cases*: *n* $\in$ *nat* $\Longrightarrow$ *n* $\leq$ *1* $\longleftrightarrow$ (*n*=*0* | *n*=*1*)
⟨*proof*⟩

lemma *succ-pred-eq-self*: ⟦*0*<*n*; *n* $\in$ *nat*⟧ $\Longrightarrow$ *succ*(*n* #− *1*) = *n*
⟨*proof*⟩

Specialized for use in the proof below.

lemma *multiset-funrestict*:
⟦$\forall$ *a*$\in$*A*. *M* ‘ *a* $\in$ *nat* $\wedge$ *0* < *M* ‘ *a*; *Finite*(*A*)⟧
$\Longrightarrow$ *multiset*(*funrestrict*(*M*, *A* − {*a*}))
⟨*proof*⟩

lemma *multiset-induct-aux*:
assumes *prem1*: $\bigwedge$*M a*. ⟦*multiset*(*M*); *a*$\notin$*mset-of*(*M*); *P*(*M*)⟧ $\Longrightarrow$ *P*(*cons*(⟨*a*, *1*⟩, *M*))
and *prem2*: $\bigwedge$*M b*. ⟦*multiset*(*M*); *b* $\in$ *mset-of*(*M*); *P*(*M*)⟧ $\Longrightarrow$ *P*(*M*(*b*:= *M*‘*b* #+ *1*))
shows
⟦*n* $\in$ *nat*; *P*(*0*)⟧
$\Longrightarrow$ ($\forall$ *M*. *multiset*(*M*)$\longrightarrow$
(*setsum*(λ*x*. \$# *mcount*(*M*, *x*), {*x* $\in$ *mset-of*(*M*). *0* < *M*‘*x*}) = \$# *n*) $\longrightarrow$ *P*(*M*))
⟨*proof*⟩

lemma *multiset-induct2*:
⟦*multiset*(*M*); *P*(*0*);

$(\bigwedge M\ a.\ [\![multiset(M);\ a \notin mset\text{-}of(M);\ P(M)]\!] \Longrightarrow P(cons(\langle a,\ 1\rangle,\ M)));$
$(\bigwedge M\ b.\ [\![multiset(M);\ b \in mset\text{-}of(M);\ \ P(M)]\!] \Longrightarrow P(M(b{:=}\ M`b\ \#{+}\ 1)))]\!]$
$\Longrightarrow P(M)$
⟨*proof*⟩

lemma *munion-single-case1*:
$[\![multiset(M);\ a \notin mset\text{-}of(M)]\!] \Longrightarrow M +\# \{\#a\#\} = cons(\langle a,\ 1\rangle,\ M)$
⟨*proof*⟩

lemma *munion-single-case2*:
$[\![multiset(M);\ a \in mset\text{-}of(M)]\!] \Longrightarrow M +\# \{\#a\#\} = M(a{:=}M`a\ \#{+}\ 1)$
⟨*proof*⟩

lemma *multiset-induct*:
assumes *M*: $multiset(M)$
and *P0*: $P(0)$
and *step*: $\bigwedge M\ a.\ [\![multiset(M);\ P(M)]\!] \Longrightarrow P(M +\# \{\#a\#\})$
shows $P(M)$
⟨*proof*⟩

lemma *MCollect-multiset* [*simp*]:
$multiset(M) \Longrightarrow multiset(\{\#\ x \in M.\ P(x)\#\})$
⟨*proof*⟩

lemma *mset-of-MCollect* [*simp*]:
$multiset(M) \Longrightarrow mset\text{-}of(\{\#\ x \in M.\ P(x)\ \#\}) \subseteq mset\text{-}of(M)$
⟨*proof*⟩

lemma *MCollect-mem-iff* [*iff*]:
$x \in mset\text{-}of(\{\#x \in M.\ P(x)\#\}) \longleftrightarrow\ x \in mset\text{-}of(M) \wedge P(x)$
⟨*proof*⟩

lemma *mcount-MCollect* [*simp*]:
$mcount(\{\#\ x \in M.\ P(x)\ \#\},\ a) = (if\ P(a)\ then\ mcount(M,a)\ else\ 0)$
⟨*proof*⟩

lemma *multiset-partition*: $multiset(M) \Longrightarrow M = \{\#\ x \in M.\ P(x)\ \#\} +\# \{\#\ x \in M.\ \neg\ P(x)\ \#\}$
⟨*proof*⟩

lemma *natify-elem-is-self* [*simp*]:
$[\![multiset(M);\ a \in mset\text{-}of(M)]\!] \Longrightarrow natify(M`a) = M`a$
⟨*proof*⟩

lemma *munion-eq-conv-diff*: $[\![multiset(M);\ multiset(N)]\!]$
$\Longrightarrow\ (M\ +\#\ \{\#a\#\} = N\ +\#\ \{\#b\#\}) \longleftrightarrow\ (M = N \wedge a = b\ |$
$M = N\ -\#\ \{\#a\#\}\ +\#\ \{\#b\#\} \wedge N = M\ -\#\ \{\#b\#\}\ +\#\ \{\#a\#\})$
$\langle proof \rangle$

lemma *melem-diff-single*:
$multiset(M) \Longrightarrow$
$k \in mset\text{-}of(M\ -\#\ \{\#a\#\}) \longleftrightarrow (k{=}a \wedge 1 < mcount(M,a))\ |\ (k{\neq}\ a \wedge k \in mset\text{-}of(M))$
$\langle proof \rangle$

lemma *munion-eq-conv-exist*:
$[\![M \in Mult(A);\ N \in Mult(A)]\!]$
$\Longrightarrow (M\ +\#\ \{\#a\#\} = N\ +\#\ \{\#b\#\}) \longleftrightarrow$
$(M{=}N \wedge a{=}b\ |\ (\exists K \in Mult(A).\ M{=}\ K\ +\#\ \{\#b\#\} \wedge N{=}K\ +\#\ \{\#a\#\}))$
$\langle proof \rangle$

8.2 Multiset Orderings

lemma *multirel1-type*: $multirel1(A,\ r) \subseteq Mult(A){*}Mult(A)$
$\langle proof \rangle$

lemma *multirel1-0* [*simp*]: $multirel1(0,\ r) = 0$
$\langle proof \rangle$

lemma *multirel1-iff*:
$\langle N,\ M\rangle \in multirel1(A,\ r) \longleftrightarrow$
$(\exists a.\ a \in A \wedge$
$(\exists M0.\ M0 \in Mult(A) \wedge (\exists K.\ K \in Mult(A) \wedge$
$M{=}M0\ +\#\ \{\#a\#\} \wedge N{=}M0\ +\#\ K \wedge (\forall b \in mset\text{-}of(K).\ \langle b,a\rangle \in r))))$
$\langle proof \rangle$

Monotonicity of *multirel1*

lemma *multirel1-mono1*: $A{\subseteq}B \Longrightarrow multirel1(A,\ r){\subseteq}multirel1(B,\ r)$
$\langle proof \rangle$

lemma *multirel1-mono2*: $r{\subseteq}s \Longrightarrow multirel1(A,r){\subseteq}multirel1(A,\ s)$
$\langle proof \rangle$

lemma *multirel1-mono*:
$[\![A{\subseteq}B;\ r{\subseteq}s]\!] \Longrightarrow multirel1(A,\ r) \subseteq multirel1(B,\ s)$
$\langle proof \rangle$

8.3 Toward the proof of well-foundedness of multirel1

lemma *not-less-0* [*iff*]: $\langle M,0\rangle \notin multirel1(A,\ r)$
$\langle proof \rangle$

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)
⟨*proof*⟩

lemma *multirel1-base*: ⟦M ∈ *Mult*(A); a ∈ A⟧ ⟹ <M, M +# {#a#}> ∈ *multirel1*(A, r)
⟨*proof*⟩

lemma *acc-0*: *acc*(0)=0
⟨*proof*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *lemma3*: ⟦*wf*[A](r); a ∈ A⟧
    ⟹ ∀ M ∈ *acc*(*multirel1*(A, r)). M +# {#a#} ∈ *acc*(*multirel1*(A, r))
⟨*proof*⟩

lemma *lemma4*: *multiset*(M) ⟹ *mset-of*(M)⊆A ⟶
  wf[A](r) ⟶ M ∈ *field*(*multirel1*(A, r)) ⟶ M ∈ *acc*(*multirel1*(A, r))
⟨*proof*⟩

lemma *all-accessible*: ⟦*wf*[A](r); M ∈ *Mult*(A); A ≠ 0⟧ ⟹ M ∈ *acc*(*multirel1*(A, r))
⟨*proof*⟩

lemma *wf-on-multirel1*: *wf*[A](r) ⟹ *wf*[A−||>*nat*−{0}](*multirel1*(A, r))
⟨*proof*⟩

lemma *wf-multirel1*: *wf*(r) ⟹*wf*(*multirel1*(*field*(r), r))
⟨*proof*⟩

lemma *multirel-type*: *multirel*(A, r) ⊆ *Mult*(A)∗*Mult*(A)
⟨*proof*⟩

lemma *multirel-mono*:
$[\![A \subseteq B;\ r \subseteq s]\!] \Longrightarrow multirel(A,\ r) \subseteq multirel(B,s)$
⟨*proof*⟩

lemma *add-diff-eq*: $k \in nat \Longrightarrow 0 < k \longrightarrow n \ \#+\ k \ \#-\ 1 = n \ \#+\ (k \ \#-\ 1)$
⟨*proof*⟩

lemma *mdiff-union-single-conv*: $[\![a \in mset\text{-}of(J);\ multiset(I);\ multiset(J)]\!]$
$\Longrightarrow I \ +\#\ J \ -\#\ \{\#a\#\} = I \ +\#\ (J -\#\ \{\#a\#\})$
⟨*proof*⟩

lemma *diff-add-commute*: $[\![n \le m;\ \ m \in nat;\ n \in nat;\ k \in nat]\!] \Longrightarrow m \ \#-\ n \ \#+\ k = m \ \#+\ k \ \#-\ n$
⟨*proof*⟩

lemma *multirel-implies-one-step*:
$\langle M,N\rangle \in multirel(A,\ r) \Longrightarrow$
$trans[A](r) \longrightarrow$
$(\exists\, I\ J\ K.$
$I \in Mult(A) \wedge J \in Mult(A) \wedge\ K \in Mult(A) \wedge$
$N = I \ +\#\ J \wedge M = I \ +\#\ K \wedge J \neq 0 \wedge$
$(\forall\, k \in mset\text{-}of(K).\ \exists\, j \in mset\text{-}of(J).\ \langle k,j\rangle \in r))$
⟨*proof*⟩

lemma *melem-imp-eq-diff-union* [*simp*]: $[\![a \in mset\text{-}of(M);\ multiset(M)]\!] \Longrightarrow M \ -\#\ \{\#a\#\} \ +\#\ \{\#a\#\} = M$
⟨*proof*⟩

lemma *msize-eq-succ-imp-eq-union*:
$[\![msize(M) = \$\#\ succ(n);\ M \in Mult(A);\ n \in nat]\!]$
$\Longrightarrow \exists\, a\ N.\ M = N \ +\#\ \{\#a\#\} \wedge N \in Mult(A) \wedge a \in A$
⟨*proof*⟩

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

lemma *one-step-implies-multirel*:

$[\![J \neq 0;\ \forall k \in \mathit{mset\text{-}of}(K).\ \exists j \in \mathit{mset\text{-}of}(J).\ \langle k,j\rangle \in r;$
$I \in \mathit{Mult}(A);\ J \in \mathit{Mult}(A);\ K \in \mathit{Mult}(A)]\!]$
$\Longrightarrow <I{+}\#K,\ I{+}\#J> \in \mathit{multirel}(A,\ r)$
⟨*proof*⟩

lemma *multirel-irrefl-lemma*:
$\mathit{Finite}(A) \Longrightarrow \mathit{part\text{-}ord}(A,\ r) \longrightarrow (\forall x \in A.\ \exists y \in A.\ \langle x,y\rangle \in r) \longrightarrow A{=}0$
⟨*proof*⟩

lemma *irrefl-on-multirel*:
$\mathit{part\text{-}ord}(A,\ r) \Longrightarrow \mathit{irrefl}(\mathit{Mult}(A),\ \mathit{multirel}(A,\ r))$
⟨*proof*⟩

lemma *trans-on-multirel*: $\mathit{trans}[\mathit{Mult}(A)](\mathit{multirel}(A,\ r))$
⟨*proof*⟩

lemma *multirel-trans*:
$[\![\langle M,\ N\rangle \in \mathit{multirel}(A,\ r);\ \langle N,\ K\rangle \in \mathit{multirel}(A,\ r)]\!] \Longrightarrow\ \langle M,\ K\rangle \in \mathit{multirel}(A,r)$
⟨*proof*⟩

lemma *trans-multirel*: $\mathit{trans}(\mathit{multirel}(A,r))$
⟨*proof*⟩

lemma *part-ord-multirel*: $\mathit{part\text{-}ord}(A,r) \Longrightarrow \mathit{part\text{-}ord}(\mathit{Mult}(A),\ \mathit{multirel}(A,\ r))$
⟨*proof*⟩

lemma *munion-multirel1-mono*:
$[\![\langle M,N\rangle \in \mathit{multirel1}(A,\ r);\ K \in \mathit{Mult}(A)]\!] \Longrightarrow <K\ {+}\#\ M,\ K\ {+}\#\ N> \in \mathit{multirel1}(A,\ r)$
⟨*proof*⟩

lemma *munion-multirel-mono2*:
$[\![\langle M,\ N\rangle \in \mathit{multirel}(A,\ r);\ K \in \mathit{Mult}(A)]\!] \Longrightarrow <K\ {+}\#\ M,\ K\ {+}\#\ N> \in \mathit{multirel}(A,\ r)$
⟨*proof*⟩

lemma *munion-multirel-mono1*:
$[\![\langle M,\ N\rangle \in \mathit{multirel}(A,\ r);\ K \in \mathit{Mult}(A)]\!] \Longrightarrow <M\ {+}\#\ K,\ N\ {+}\#\ K> \in \mathit{multirel}(A,\ r)$
⟨*proof*⟩

lemma *munion-multirel-mono*:
$[\![\langle M,K\rangle \in \mathit{multirel}(A,\ r);\ \langle N,L\rangle \in \mathit{multirel}(A,\ r)]\!]$

$\Longrightarrow$ *<M +# N, K +# L>* $\in$ *multirel(A, r)*
⟨*proof*⟩

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)* $\Longrightarrow$ *multiset(M)*
⟨*proof*⟩

lemma *munion-omultiset* [*simp*]: ⟦*omultiset(M)*; *omultiset(N)*⟧ $\Longrightarrow$ *omultiset(M +# N)*
⟨*proof*⟩

lemma *mdiff-omultiset* [*simp*]: *omultiset(M)* $\Longrightarrow$ *omultiset(M −# N)*
⟨*proof*⟩

lemma *irrefl-Memrel*: *Ord(i)* $\Longrightarrow$ *irrefl(field(Memrel(i)), Memrel(i))*
⟨*proof*⟩

lemma *trans-iff-trans-on*: *trans(r)* $\longleftrightarrow$ *trans[field(r)](r)*
⟨*proof*⟩

lemma *part-ord-Memrel*: *Ord(i)* $\Longrightarrow$*part-ord(field(Memrel(i)), Memrel(i))*
⟨*proof*⟩

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

lemma *mless-not-refl*: ¬(*M <# M*)
⟨*proof*⟩

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

lemma *mless-trans*: ⟦*K <# M*; *M <# N*⟧ $\Longrightarrow$ *K <# N*
⟨*proof*⟩

lemma *mless-not-sym*: $M <\# N \Longrightarrow \neg N <\# M$
⟨*proof*⟩

lemma *mless-asym*: $[\![M <\# N; \neg P \Longrightarrow N <\# M]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *mle-refl* [*simp*]: $omultiset(M) \Longrightarrow M <\#= M$
⟨*proof*⟩

lemma *mle-antisym*:
$[\![M <\#= N;\ \ N <\#= M]\!] \Longrightarrow M = N$
⟨*proof*⟩

lemma *mle-trans*: $[\![K <\#= M; M <\#= N]\!] \Longrightarrow K <\#= N$
⟨*proof*⟩

lemma *mless-le-iff*: $M <\# N \longleftrightarrow (M <\#= N \wedge M \neq N)$
⟨*proof*⟩

lemma *munion-less-mono2*: $[\![M <\# N; omultiset(K)]\!] \Longrightarrow K +\# M <\# K +\# N$
⟨*proof*⟩

lemma *munion-less-mono1*: $[\![M <\# N; omultiset(K)]\!] \Longrightarrow M +\# K <\# N +\# K$
⟨*proof*⟩

lemma *mless-imp-omultiset*: $M <\# N \Longrightarrow omultiset(M) \wedge omultiset(N)$
⟨*proof*⟩

lemma *munion-less-mono*: $[\![M <\# K; N <\# L]\!] \Longrightarrow M +\# N <\# K +\# L$
⟨*proof*⟩

lemma *mle-imp-omultiset*: $M <\#= N \Longrightarrow omultiset(M) \wedge omultiset(N)$
⟨*proof*⟩

lemma *mle-mono*: $[\![M <\#= K;\ \ N <\#= L]\!] \Longrightarrow M +\# N <\#= K +\# L$
⟨*proof*⟩

lemma *omultiset-0* [*iff*]: $omultiset(0)$
⟨*proof*⟩

lemma *empty-leI* [*simp*]: *omultiset*(*M*) ⟹ *0* <#= *M*
⟨*proof*⟩

lemma *munion-upper1*: ⟦*omultiset*(*M*); *omultiset*(*N*)⟧ ⟹ *M* <#= *M* +# *N*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⊆ *A* × *B* ⟹ *rmap*(*r*) ⊆ *list*(*A*) × *list*(*B*)
⟨*proof*⟩

lemma *rmap-total*: *A* ⊆ *domain*(*r*) ⟹ *list*(*A*) ⊆ *domain*(*rmap*(*r*))
⟨*proof*⟩

lemma *rmap-functional*: *function*(*r*) ⟹ *function*(*rmap*(*r*))
⟨*proof*⟩

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

lemma *rmap-fun-type*: *f* ∈ *A*−>*B* ⟹ *rmap*(*f*): *list*(*A*)−>*list*(*B*)
⟨*proof*⟩

lemma *rmap-Nil*: *rmap*(*f*)‘*Nil* = *Nil*

⟨*proof*⟩

lemma *rmap-Cons*: $[\![f \in A{-}{>}B;\ \ x \in A;\ \ xs\colon list(A)]\!]$
$\Longrightarrow rmap(f)\ `\ Cons(x,xs) = Cons(f`x,\ rmap(f)`xs)$
⟨*proof*⟩

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 $H \mid= p$ then $G \mid= p$ where $G \in Fin(H)$

10.1 The datatype of propositions

consts
propn :: *i*

datatype *propn* =
Fls
| *Var* ($n \in nat$) (‹#-› [*100*] *100*)
| *Imp* ($p \in propn$, $q \in propn$) (**infixr** ‹⇒› *90*)

10.2 The proof system

consts *thms* :: $i \Rightarrow i$

abbreviation
thms-syntax :: $[i,i] \Rightarrow o$ (**infixl** ‹|−› *50*)
where $H \mid- p \equiv p \in thms(H)$

inductive
domains $thms(H) \subseteq propn$
intros
H: $[\![p \in H;\ \ p \in propn]\!] \Longrightarrow H \mid- p$
K: $[\![p \in propn;\ \ q \in propn]\!] \Longrightarrow H \mid- p{\Rightarrow}q{\Rightarrow}p$
S: $[\![p \in propn;\ \ q \in propn;\ \ r \in propn]\!]$
$\Longrightarrow H \mid- (p{\Rightarrow}q{\Rightarrow}r) \Rightarrow (p{\Rightarrow}q) \Rightarrow p{\Rightarrow}r$
DN: $p \in propn \Longrightarrow H \mid- ((p{\Rightarrow}Fls) \Rightarrow Fls) \Rightarrow p$
MP: $[\![H \mid- p{\Rightarrow}q;\ \ H \mid- p;\ \ p \in propn;\ \ q \in propn]\!] \Longrightarrow H \mid- q$
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*
  ⟨*proof*⟩

lemma *is-true-Var* [*simp*]: *is-true*(#*v*,*t*) ⟷ *v* ∈ *t*
  ⟨*proof*⟩

lemma *is-true-Imp* [*simp*]: *is-true*(*p*⇒*q*,*t*) ⟷ (*is-true*(*p*,*t*)⟶*is-true*(*q*,*t*))
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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

inductive-cases *ImpE*: *p*⇒*q* ∈ *propn*

lemma *thms-MP*: $[\![H \mid\!- \ p{\Rightarrow}q;\ \ H \mid\!- \ p]\!] \Longrightarrow H \mid\!- \ q$
— Stronger Modus Ponens rule: no typechecking!
⟨*proof*⟩

lemma *thms-I*: $p \in propn \Longrightarrow H \mid\!- \ p{\Rightarrow}p$
— Rule is called *I* for Identity Combinator, not for Introduction.
⟨*proof*⟩

10.4.1 Weakening, left and right

lemma *weaken-left*: $[\![G \subseteq H;\ \ G|\!-p]\!] \Longrightarrow H|\!-p$
— Order of premises is convenient with *THEN*
⟨*proof*⟩

lemma *weaken-left-cons*: $H \mid\!- \ p \Longrightarrow cons(a,H) \mid\!- \ p$
⟨*proof*⟩

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

lemma *weaken-right*: $[\![H \mid\!- \ q;\ \ p \in propn]\!] \Longrightarrow H \mid\!- \ p{\Rightarrow}q$
⟨*proof*⟩

10.4.2 The deduction theorem

theorem *deduction*: $[\![cons(p,H) \mid\!- \ q;\ \ p \in propn]\!] \Longrightarrow\ H \mid\!- \ p{\Rightarrow}q$
⟨*proof*⟩

10.4.3 The cut rule

lemma *cut*: $[\![H|\!-p;\ \ cons(p,H) \mid\!- \ q]\!] \Longrightarrow\ H \mid\!- \ q$
⟨*proof*⟩

lemma *thms-FlsE*: $[\![H \mid\!- \ Fls;\ p \in propn]\!] \Longrightarrow H \mid\!- \ p$
⟨*proof*⟩

lemma *thms-notE*: $[\![H \mid\!- \ p{\Rightarrow}Fls;\ \ H \mid\!- \ p;\ \ q \in propn]\!] \Longrightarrow H \mid\!- \ q$
⟨*proof*⟩

10.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: $H \mid\!- \ p \Longrightarrow H \mid\!= \ p$
⟨*proof*⟩

10.5 Completeness

10.5.1 Towards the completeness proof

lemma *Fls-Imp*: $[\![H \mid\!- \ p{\Rightarrow}Fls;\ q \in propn]\!] \Longrightarrow H \mid\!- \ p{\Rightarrow}q$
⟨*proof*⟩

lemma *Imp-Fls*: $[\![H \mid\!- p;\ \ H \mid\!- q{\Rightarrow}Fls]\!] \Longrightarrow H \mid\!- (p{\Rightarrow}q){\Rightarrow}Fls$
⟨*proof*⟩

lemma *hyps-thms-if*:
$p \in propn \Longrightarrow hyps(p,t) \mid\!- (if\ is\text{-}true(p,t)\ then\ p\ else\ p{\Rightarrow}Fls)$
— Typical example of strengthening the induction statement.
⟨*proof*⟩

lemma *logcon-thms-p*: $[\![p \in propn;\ \ 0 \mid= p]\!] \Longrightarrow hyps(p,t) \mid\!- p$
— Key lemma for completeness; yields a set of assumptions satisfying p
⟨*proof*⟩

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*:
$[\![p \in propn;\ \ q \in propn]\!] \Longrightarrow H \mid\!- (p{\Rightarrow}q) \Rightarrow ((p{\Rightarrow}Fls){\Rightarrow}q) \Rightarrow q$
⟨*proof*⟩

lemma *thms-excluded-middle-rule*:
$[\![cons(p,H) \mid\!- q;\ \ cons(p{\Rightarrow}Fls,H) \mid\!- q;\ \ p \in propn]\!] \Longrightarrow H \mid\!- q$
— Hard to prove directly because it requires cuts
⟨*proof*⟩

10.5.2 Completeness – lemmas for reducing the set of assumptions

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

lemma *hyps-Diff*:
$p \in propn \Longrightarrow hyps(p,\ t{-}\{v\}) \subseteq cons(\#v{\Rightarrow}Fls,\ hyps(p,t){-}\{\#v\})$
⟨*proof*⟩

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

lemma *hyps-cons*:
$p \in propn \Longrightarrow hyps(p,\ cons(v,t)) \subseteq cons(\#v,\ hyps(p,t){-}\{\#v{\Rightarrow}Fls\})$
⟨*proof*⟩

Two lemmas for use with *weaken-left*

lemma *cons-Diff-same*: $B{-}C \subseteq cons(a,\ B{-}cons(a,C))$
⟨*proof*⟩

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

⟨*proof*⟩

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

lemma *hyps-finite*: $p \in propn \Longrightarrow hyps(p,t) \in Fin(\bigcup v \in nat. \{\#v, \#v \Rightarrow Fls\})$
⟨*proof*⟩

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*]:
$[\![p \in propn;\ 0 \mid= p]\!] \Longrightarrow \forall t.\ hyps(p,t) - hyps(p,t0) \mid- p$
⟨*proof*⟩

10.5.3 Completeness theorem

lemma *completeness-0*: $[\![p \in propn;\ 0 \mid= p]\!] \Longrightarrow 0 \mid- p$
— The base case for completeness
⟨*proof*⟩

lemma *logcon-Imp*: $[\![cons(p,H) \mid= q]\!] \Longrightarrow H \mid= p \Rightarrow q$
— A semantic analogue of the Deduction Theorem
⟨*proof*⟩

lemma *completeness*:
$H \in Fin(propn) \Longrightarrow p \in propn \Longrightarrow H \mid= p \Longrightarrow H \mid- p$
⟨*proof*⟩

theorem *thms-iff*: $H \in Fin(propn) \Longrightarrow H \mid- p \longleftrightarrow H \mid= p \wedge p \in propn$
⟨*proof*⟩

end

11 Lists of n elements

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

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

consts *listn* :: $i \Rightarrow i$
inductive
  domains $listn(A) \subseteq nat \times list(A)$
  intros
    NilI: $\langle 0,Nil\rangle \in listn(A)$
    ConsI: $[\![a \in A; \langle n,l\rangle \in listn(A)]\!] \Longrightarrow <succ(n), Cons(a,l)> \in listn(A)$
  type-intros *nat-typechecks list.intros*

lemma *list-into-listn*: $l \in list(A) \Longrightarrow <length(l),l> \in listn(A)$
⟨*proof*⟩

lemma *listn-iff*: $\langle n,l\rangle \in listn(A) \longleftrightarrow l \in list(A) \wedge length(l)=n$
⟨*proof*⟩

lemma *listn-image-eq*: $listn(A)``\{n\} = \{l \in list(A).\ length(l)=n\}$
⟨*proof*⟩

lemma *listn-mono*: $A \subseteq B \Longrightarrow listn(A) \subseteq listn(B)$
⟨*proof*⟩

lemma *listn-append*:
$[\![\langle n,l\rangle \in listn(A);\ <n',l'> \in listn(A)]\!] \Longrightarrow <n\#+n',\ l@l'> \in listn(A)$
⟨*proof*⟩

inductive-cases
Nil-listn-case: $\langle i,Nil\rangle \in listn(A)$
and *Cons-listn-case*: $<i,Cons(x,l)> \in listn(A)$

inductive-cases
zero-listn-case: $\langle 0,l\rangle \in listn(A)$
and *succ-listn-case*: $<succ(i),l> \in listn(A)$

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* ($p \in comb$, $q \in comb$) (**infixl** ‹·› *90*)

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

consts *contract* :: *i*
abbreviation *contract-syntax* :: $[i,i] \Rightarrow o$ (**infixl** ‹$\rightarrow^1$› *50*)
  where $p \rightarrow^1 q \equiv \langle p,q \rangle \in contract$

abbreviation *contract-multi* :: $[i,i] \Rightarrow o$ (**infixl** ‹$\rightarrow$› *50*)
  where $p \rightarrow q \equiv \langle p,q \rangle \in contract\hat{}*$

inductive
  domains $contract \subseteq comb \times comb$
  intros
    K: $[\![p \in comb; \; q \in comb]\!] \Longrightarrow K \cdot p \cdot q \rightarrow^1 p$
    S: $[\![p \in comb; \; q \in comb; \; r \in comb]\!] \Longrightarrow S \cdot p \cdot q \cdot r \rightarrow^1 (p \cdot r) \cdot (q \cdot r)$
    Ap1: $[\![p \rightarrow^1 q; \; r \in comb]\!] \Longrightarrow p \cdot r \rightarrow^1 q \cdot r$
    Ap2: $[\![p \rightarrow^1 q; \; r \in comb]\!] \Longrightarrow r \cdot p \rightarrow^1 r \cdot q$
  type-intros *comb.intros*

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

consts *parcontract* :: *i*

abbreviation *parcontract-syntax* :: $[i,i] \Rightarrow o$ (**infixl** ‹$\Rrightarrow^1$› *50*)
  where $p \Rrightarrow^1 q \equiv \langle p,q \rangle \in parcontract$

abbreviation *parcontract-multi* :: $[i,i] \Rightarrow o$ (**infixl** ‹$\Rrightarrow$› *50*)
  where $p \Rrightarrow q \equiv \langle p,q \rangle \in parcontract\hat{}+$

inductive
  domains $parcontract \subseteq comb \times comb$
  intros
    refl: $[\![p \in comb]\!] \Longrightarrow p \Rrightarrow^1 p$
    K: $[\![p \in comb; \; q \in comb]\!] \Longrightarrow K \cdot p \cdot q \Rrightarrow^1 p$
    S: $[\![p \in comb; \; q \in comb; \; r \in comb]\!] \Longrightarrow S \cdot p \cdot q \cdot r \Rrightarrow^1 (p \cdot r) \cdot (q \cdot r)$
    Ap: $[\![p \Rrightarrow^1 q; \; r \Rrightarrow^1 s]\!] \Longrightarrow p \cdot r \Rrightarrow^1 q \cdot s$
  type-intros *comb.intros*

Misc definitions.

definition *I* :: *i*
  where $I \equiv S \cdot K \cdot K$

definition *diamond* :: $i \Rightarrow o$
  where $diamond(r) \equiv$
    $\forall x\ y.\ \langle x,y \rangle \in r \longrightarrow (\forall y'.\ <x,y'> \in r \longrightarrow (\exists z.\ \langle y,z \rangle \in r \land <y',z> \in r))$

12.2 Transitive closure preserves the Church-Rosser property

lemma *diamond-strip-lemmaD* [*rule-format*]:
  $[\![diamond(r); \; \langle x,y \rangle : r\hat{}+]\!] \Longrightarrow$
    $\forall y'.\ <x,y'> : r \longrightarrow (\exists z.\ <y',z> : r\hat{}+ \land \langle y,z \rangle : r)$

⟨*proof*⟩

lemma *diamond-trancl*: $diamond(r) \Longrightarrow diamond(r\hat{}+)$
⟨*proof*⟩

inductive-cases *Ap-E* [*elim*!]: $p \cdot q \in comb$

12.3 Results about Contraction

For type checking: replaces $a \to^1 b$ 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*
⟨*proof*⟩

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 $p \in comb \Longrightarrow I \cdot p \to p$
— Example only: not used
⟨*proof*⟩

lemma *comb-I*: $I \in comb$
⟨*proof*⟩

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*: $I \to^1 r \Longrightarrow P$
⟨*proof*⟩

lemma *K1-contractD*: $K{\cdot}p \to^1 r \Longrightarrow (\exists\, q.\ r = K{\cdot}q \wedge p \to^1 q)$
⟨*proof*⟩

lemma *Ap-reduce1*: $[\![p \to q;\ \ r \in comb]\!] \Longrightarrow p{\cdot}r \to q{\cdot}r$
⟨*proof*⟩

lemma *Ap-reduce2*: $[\![p \to q;\ \ r \in comb]\!] \Longrightarrow r{\cdot}p \to r{\cdot}q$
⟨*proof*⟩

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

lemma *KIII-contract1*: $K{\cdot}I{\cdot}(I{\cdot}I) \to^1 I$
⟨*proof*⟩

lemma *KIII-contract2*: $K{\cdot}I{\cdot}(I{\cdot}I) \to^1 K{\cdot}I{\cdot}((K{\cdot}I){\cdot}(K{\cdot}I))$
⟨*proof*⟩

lemma *KIII-contract3*: $K{\cdot}I{\cdot}((K{\cdot}I){\cdot}(K{\cdot}I)) \to^1 I$
⟨*proof*⟩

lemma *not-diamond-contract*: $\neg\ diamond(contract)$
⟨*proof*⟩

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*
⟨*proof*⟩

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*!]:
$K{\cdot}p \Rrightarrow^1 r \Longrightarrow (\exists\, p'.\ r = K{\cdot}p' \wedge p \Rrightarrow^1 p')$
⟨*proof*⟩

lemma *S1-parcontractD* [*dest*!]:
  $S{\cdot}p \Rrightarrow^1 r \Longrightarrow (\exists p'. \ r = S{\cdot}p' \wedge p \Rrightarrow^1 p')$
  ⟨*proof*⟩

lemma *S2-parcontractD* [*dest*!]:
  $S{\cdot}p{\cdot}q \Rrightarrow^1 r \Longrightarrow (\exists p' \ q'. \ r = S{\cdot}p'{\cdot}q' \wedge p \Rrightarrow^1 p' \wedge q \Rrightarrow^1 q')$
  ⟨*proof*⟩

lemma *diamond-parcontract*: *diamond*(*parcontract*)
— Church-Rosser property for parallel contraction
  ⟨*proof*⟩

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

lemma *contract-imp-parcontract*: $p{\rightarrow}^1 q \Longrightarrow p{\Rrightarrow}^1 q$
  ⟨*proof*⟩

lemma *reduce-imp-parreduce*: $p{\rightarrow}q \Longrightarrow p{\Rrightarrow}q$
  ⟨*proof*⟩

lemma *parcontract-imp-reduce*: $p{\Rrightarrow}^1 q \Longrightarrow p{\rightarrow}q$
  ⟨*proof*⟩

lemma *parreduce-imp-reduce*: $p{\Rrightarrow}q \Longrightarrow p{\rightarrow}q$
  ⟨*proof*⟩

lemma *parreduce-iff-reduce*: $p{\Rrightarrow}q \longleftrightarrow p{\rightarrow}q$
  ⟨*proof*⟩

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**
  $SC \equiv \lambda l \in list(nat). \ list\text{-}case(0, \lambda x \ xs. \ succ(x), l)$

definition
  CONSTANT :: $i{\Rightarrow}i$ **where**

$CONSTANT(k) \equiv \lambda l \in list(nat).\ k$

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

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

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

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

abbreviation
$ack :: [i,i] \Rightarrow i$ **where**
$ack(x,y) \equiv ACK(x)\ `\ [y]$

Useful special cases of evaluation.

lemma SC: $[\![x \in nat;\ \ l \in list(nat)]\!] \Longrightarrow SC\ `\ (Cons(x,l)) = succ(x)$
⟨*proof*⟩

lemma $CONSTANT$: $l \in list(nat) \Longrightarrow CONSTANT(k)\ `\ l = k$
⟨*proof*⟩

lemma $PROJ\text{-}0$: $[\![x \in nat;\ \ l \in list(nat)]\!] \Longrightarrow PROJ(0)\ `\ (Cons(x,l)) = x$
⟨*proof*⟩

lemma $COMP\text{-}1$: $l \in list(nat) \Longrightarrow COMP(g,[f])\ `\ l = g`\ [f`l]$
⟨*proof*⟩

lemma $PREC\text{-}0$: $l \in list(nat) \Longrightarrow PREC(f,g)\ `\ (Cons(0,l)) = f`l$
⟨*proof*⟩

lemma $PREC\text{-}succ$:
$[\![x \in nat;\ \ l \in list(nat)]\!]$
$\Longrightarrow PREC(f,g)\ `\ (Cons(succ(x),l)) =$
$g\ `\ Cons(PREC(f,g)`(Cons(x,l)),\ Cons(x,l))$
⟨*proof*⟩

13.2 Inductive definition of the PR functions

consts
  prim-rec :: *i*

inductive
  domains *prim-rec* ⊆ *list*(*nat*)−>*nat*
  intros
    SC ∈ *prim-rec*
    k ∈ *nat* ⟹ *CONSTANT*(*k*) ∈ *prim-rec*
    i ∈ *nat* ⟹ *PROJ*(*i*) ∈ *prim-rec*
    ⟦*g* ∈ *prim-rec*; *fs*∈*list*(*prim-rec*)⟧ ⟹ *COMP*(*g*,*fs*) ∈ *prim-rec*
    ⟦*f* ∈ *prim-rec*; *g* ∈ *prim-rec*⟧ ⟹ *PREC*(*f*,*g*) ∈ *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* ∈ *prim-rec* ⟹ *c* ∈ *list*(*nat*) −> *nat*
  ⟨*proof*⟩

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* ∈ *nat* ⟹ *ACK*(*i*) ∈ *prim-rec*
  ⟨*proof*⟩

lemma *ack-type* [*TC*]: ⟦*i* ∈ *nat*; *j* ∈ *nat*⟧ ⟹ *ack*(*i*,*j*) ∈ *nat*
  ⟨*proof*⟩

13.3 Ackermann's function cases

lemma *ack-0*: *j* ∈ *nat* ⟹ *ack*(*0*,*j*) = *succ*(*j*)
  — PROPERTY A 1
  ⟨*proof*⟩

lemma *ack-succ-0*: *ack*(*succ*(*i*), *0*) = *ack*(*i*,*1*)
  — PROPERTY A 2
  ⟨*proof*⟩

lemma *ack-succ-succ*:
  ⟦*i*∈*nat*; *j*∈*nat*⟧ ⟹ *ack*(*succ*(*i*), *succ*(*j*)) = *ack*(*i*, *ack*(*succ*(*i*), *j*))
  — PROPERTY A 3
  ⟨*proof*⟩

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

lemma *lt-ack2*: $i \in nat \Longrightarrow j \in nat \Longrightarrow j < ack(i,j)$
— PROPERTY A 4
⟨*proof*⟩

lemma *ack-lt-ack-succ2*: $[\![i{\in}nat;\ j{\in}nat]\!] \Longrightarrow ack(i,j) < ack(i,\ succ(j))$
— PROPERTY A 5-, the single-step lemma
⟨*proof*⟩

lemma *ack-lt-mono2*: $[\![j<k;\ i \in nat;\ k \in nat]\!] \Longrightarrow ack(i,j) < ack(i,k)$
— PROPERTY A 5, monotonicity for <
⟨*proof*⟩

lemma *ack-le-mono2*: $[\![j{\le}k;\ \ i{\in}nat;\ \ k{\in}nat]\!] \Longrightarrow ack(i,j) \le ack(i,k)$
— PROPERTY A 5', monotonicity for ≤
⟨*proof*⟩

lemma *ack2-le-ack1*:
$[\![i{\in}nat;\ \ j{\in}nat]\!] \Longrightarrow ack(i,\ succ(j)) \le ack(succ(i),\ j)$
— PROPERTY A 6
⟨*proof*⟩

lemma *ack-lt-ack-succ1*: $[\![i \in nat;\ j \in nat]\!] \Longrightarrow ack(i,j) < ack(succ(i),j)$
— PROPERTY A 7-, the single-step lemma
⟨*proof*⟩

lemma *ack-lt-mono1*: $[\![i<j;\ j \in nat;\ k \in nat]\!] \Longrightarrow ack(i,k) < ack(j,k)$
— PROPERTY A 7, monotonicity for <
⟨*proof*⟩

lemma *ack-le-mono1*: $[\![i{\le}j;\ j \in nat;\ k \in nat]\!] \Longrightarrow ack(i,k) \le ack(j,k)$
— PROPERTY A 7', monotonicity for ≤
⟨*proof*⟩

lemma *ack-1*: $j \in nat \Longrightarrow ack(1,j) = succ(succ(j))$
— PROPERTY A 8
⟨*proof*⟩

lemma *ack-2*: $j \in nat \Longrightarrow ack(succ(1),j) = succ(succ(succ(j\#{+}j)))$
— PROPERTY A 9
⟨*proof*⟩

lemma *ack-nest-bound*:
$[\![i1 \in nat;\ i2 \in nat;\ j \in nat]\!]$
$\Longrightarrow ack(i1,\ ack(i2,j)) < ack(succ(succ(i1\#{+}i2)),\ j)$
— PROPERTY A 10

⟨*proof*⟩

lemma *ack-add-bound*:
$[\![i1 \in nat;\ i2 \in nat;\ j \in nat]\!]$
$\Longrightarrow ack(i1,j)\ \#+\ ack(i2,j) < ack(succ(succ(succ(succ(i1\#+i2)))),\ j)$
— PROPERTY A 11
⟨*proof*⟩

lemma *ack-add-bound2*:
$[\![i < ack(k,j);\ \ j \in nat;\ \ k \in nat]\!]$
$\Longrightarrow i\#+j < ack(succ(succ(succ(succ(k)))),\ j)$
— PROPERTY A 12.
— Article uses existential quantifier but the ALF proof used $k\ \#+\ \#4$.
— Quantified version must be nested $\exists\, k'.\ \forall\, i,j\ \ldots$.
⟨*proof*⟩

13.4 Main result

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

lemma *SC-case*: $l \in list(nat) \Longrightarrow SC\ `\ l < ack(1,\ list\text{-}add(l))$
⟨*proof*⟩

lemma *lt-ack1*: $[\![i \in nat;\ j \in nat]\!] \Longrightarrow i < ack(i,j)$
— PROPERTY A 4'? Extra lemma needed for *CONSTANT* case, constant functions.
⟨*proof*⟩

lemma *CONSTANT-case*:
$[\![l \in list(nat);\ \ k \in nat]\!] \Longrightarrow CONSTANT(k)\ `\ l < ack(k,\ list\text{-}add(l))$
⟨*proof*⟩

lemma *PROJ-case* [*rule-format*]:
$l \in list(nat) \Longrightarrow \forall\, i \in nat.\ PROJ(i)\ `\ l < ack(0,\ list\text{-}add(l))$
⟨*proof*⟩

COMP case.

lemma *COMP-map-lemma*:
$fs \in list(\{f \in prim\text{-}rec.\ \exists\, kf \in nat.\ \forall\, l \in list(nat).\ f`l < ack(kf,\ list\text{-}add(l))\})$
$\Longrightarrow \exists\, k \in nat.\ \forall\, l \in list(nat).$
$list\text{-}add(map(\lambda f.\ f\ `\ l,\ fs)) < ack(k,\ list\text{-}add(l))$
⟨*proof*⟩

lemma *COMP-case*:
$[\![kg{\in}nat;$
$\forall\, l \in list(nat).\ g`l < ack(kg,\ list\text{-}add(l));$
$fs \in list(\{f \in prim\text{-}rec\ .$
$\exists\, kf \in nat.\ \forall\, l \in list(nat).$
$f`l < ack(kf,\ list\text{-}add(l))\})]\!]$

$\Longrightarrow \exists\, k \in nat.\ \forall\, l \in list(nat).\ COMP(g,fs)\text{‘}l < ack(k,\ list\text{-}add(l))$
⟨proof⟩

PREC case.

lemma *PREC-case-lemma*:
$[\![\forall\, l \in list(nat).\ f\text{‘}l \ \#+\ list\text{-}add(l) < ack(kf,\ list\text{-}add(l));$
$\forall\, l \in list(nat).\ g\text{‘}l \ \#+\ list\text{-}add(l) < ack(kg,\ list\text{-}add(l));$
$f \in prim\text{-}rec;\ \ kf{\in}nat;$
$g \in prim\text{-}rec;\ \ kg{\in}nat;$
$l \in list(nat)]\!]$
$\Longrightarrow PREC(f,g)\text{‘}l \ \#+\ list\text{-}add(l) < ack(succ(kf\#{+}kg),\ list\text{-}add(l))$
⟨proof⟩

lemma *PREC-case*:
$[\![f \in prim\text{-}rec;\ \ kf{\in}nat;$
$g \in prim\text{-}rec;\ \ kg{\in}nat;$
$\forall\, l \in list(nat).\ f\text{‘}l < ack(kf,\ list\text{-}add(l));$
$\forall\, l \in list(nat).\ g\text{‘}l < ack(kg,\ list\text{-}add(l))]\!]$
$\Longrightarrow \exists\, k \in nat.\ \forall\, l \in list(nat).\ PREC(f,g)\text{‘}l < ack(k,\ list\text{-}add(l))$
⟨proof⟩

lemma *ack-bounds-prim-rec*:
$f \in prim\text{-}rec \Longrightarrow \exists\, k \in nat.\ \forall\, l \in list(nat).\ f\text{‘}l < ack(k,\ list\text{-}add(l))$
⟨proof⟩

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

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.