Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

June 21, 2010

Contents

1 Sample datatype definitions

theory *Datatypes* **imports** *Main* **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** *Main* **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*: $Br(a, l, r) = Br(a', l', r') \leftrightarrow a = a' \,\&\, l = l' \,\&\, r = r'$
— 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*:
$[\!|\ t \in bt(A);$
$\quad c \in C(Lf);$
$\quad !!x\ y\ z\ r\ s.\ [\!|\ x \in A;\ \ y \in bt(A);\ \ z \in bt(A);\ \ r \in C(y);\ \ s \in C(z)\ |\!] \Longrightarrow$
$\quad h(x, y, z, r, s) \in C(Br(x, y, z))$
$|\!] \Longrightarrow bt\text{-}rec(c, h, t) \in C(t)$
— 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-nodes-aux* :: $i \Rightarrow i$
primrec
n-nodes-aux(*Lf*) = ($\lambda k \in nat.\ k$)
n-nodes-aux(*Br*(*a*, *l*, *r*)) =
($\lambda k \in nat.$ *n-nodes-aux*(*r*) ‘ (*n-nodes-aux*(*l*) ‘ *succ*(*k*)))

lemma *n-nodes-aux-eq*:
$t \in bt(A) \Longrightarrow k \in nat \Longrightarrow$ *n-nodes-aux*(*t*)‘*k* = *n-nodes*(*t*) #+ *k*
⟨*proof*⟩

definition
n-nodes-tail :: $i \Rightarrow i$ **where**
n-nodes-tail(*t*) == *n-nodes-aux*(*t*) ‘ *0*

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

2.3 Number of leaves

consts
n-leaves :: $i \Rightarrow i$
primrec
n-leaves(*Lf*) = *1*
n-leaves(*Br*(*a*, *l*, *r*)) = *n-leaves*(*l*) #+ *n-leaves*(*r*)

lemma *n-leaves-type* [*simp*]: $t \in bt(A) \Longrightarrow$ *n-leaves*(*t*) $\in nat$
⟨*proof*⟩

2.4 Reflecting trees

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

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

Theorems about *n-leaves*.

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

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

Theorems about *bt-reflect*.

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

end

3 Terms over an alphabet

theory *Term* **imports** *Main* **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*));

!!$x\ z\ zs$. [| $x \in A$; $z \in term(A)$; 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*.
⟨*proof*⟩

lemma *term-induct-eqn* [*consumes 1*, *case-names Apply*]:
[| $t \in term(A)$;
!!$x\ zs$. [| $x \in A$; 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.
⟨*proof*⟩

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

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

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

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

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

term-rec – by *Vset* recursion.

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

lemma *term-rec* [*simp*]: $ts \in list(A)$ ==>
term-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: !!$x\ zs\ r$. [| $x \in A$; zs: $list(term(A))$;
$r \in list(\bigcup t \in term(A).\ C(t))$ |]
==> $d(x,\ zs,\ r)$: $C(Apply(x,zs))$
shows *term-rec*$(t,d) \in C(t)$
— Slightly odd typing condition on r in the second premise!
⟨*proof*⟩

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))
⟨*proof*⟩

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
⟨*proof*⟩

term-map.

lemma *term-map* [*simp*]:
ts ∈ *list*(A) ==>
term-map(f, *Apply*(a, ts)) = *Apply*(f(a), *map*(*term-map*(f), ts))
⟨*proof*⟩

lemma *term-map-type* [TC]:
[| t ∈ *term*(A); !!x. x ∈ A ==> f(x): B |] ==> *term-map*(f,t) ∈ *term*(B)
⟨*proof*⟩

lemma *term-map-type2* [TC]:
t ∈ *term*(A) ==> *term-map*(f,t) ∈ *term*({f(u). u ∈ A})
⟨*proof*⟩

term-size.

lemma *term-size* [*simp*]:
ts ∈ *list*(A) ==> *term-size*(*Apply*(a, ts)) = *succ*(*list-add*(*map*(*term-size*, ts)))
⟨*proof*⟩

lemma *term-size-type* [TC]: t ∈ *term*(A) ==> *term-size*(t) ∈ *nat*
⟨*proof*⟩

reflect.

lemma *reflect* [*simp*]:
ts ∈ *list*(A) ==> *reflect*(*Apply*(a, ts)) = *Apply*(a, *rev*(*map*(*reflect*, ts)))
⟨*proof*⟩

lemma *reflect-type* [TC]: t ∈ *term*(A) ==> *reflect*(t) ∈ *term*(A)
⟨*proof*⟩

preorder.

lemma *preorder* [*simp*]:
ts ∈ *list*(A) ==> *preorder*(*Apply*(a, ts)) = *Cons*(a, *flat*(*map*(*preorder*, ts)))
⟨*proof*⟩

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

postorder.

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

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

Theorems about *term-map.*

declare *map-compose* [*simp*]

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

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

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

Theorems about *term-size.*

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

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

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

Theorems about *reflect.*

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

Theorems about preorder.

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

lemma *preorder-reflect-eq-rev-postorder*:

$t \in term(A) ==> preorder(reflect(t)) = rev(postorder(t))$
⟨*proof*⟩

end

4 Datatype definition n-ary branching trees

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

datatype $maptree(A) = Sons\ (a \in A,\ h \in maptree(A) -||> maptree(A))$
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**
$ntree\text{-}rec(b) ==$
$Vrecursor(\lambda pr.\ ntree\text{-}case(\lambda x\ h.\ b(x, h, \lambda i \in domain(h).\ pr`(h`i))))$

definition
ntree-copy :: $i => i$ **where**
$ntree\text{-}copy(z) == ntree\text{-}rec(\lambda x\ h\ r.\ Branch(x,r), z)$

ntree

lemma *ntree-unfold*: $ntree(A) = A \times (\bigcup n \in nat.\ n -> ntree(A))$
⟨*proof*⟩

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

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

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

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

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

ntree recursion.

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

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

lemma *ntree-copy-is-ident*: $z \in ntree(A) ==> ntree\text{-}copy(z) = z$
⟨*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*: $!!x\ n\ h.\ [|\ x \in A;\ \ h \in maptree(A)\ -||>\ maptree(A);$
$\forall\, y \in field(h).\ P(y)$

$|]$ ==> $P(Sons(x,h))$
shows $P(t)$
— A nicer induction rule than the standard one.
⟨*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*: !!$x\ n\ h.$ $[|\ x \in A;\ \ h \in B\ -||>\ maptree2(A,B);\ \ \forall y \in range(h).\ P(y)$
$|]$ ==> $P(Sons2(x,h))$
shows $P(t)$
⟨*proof*⟩

end

5 Trees and forests, a mutually recursive type definition

theory *Tree-Forest* **imports** *Main* **begin**

5.1 Datatype definition

consts
tree :: i => i
forest :: i => i
tree-forest :: i => 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*, *standard*, *consumes 1*]
and *forest'induct* =
tree-forest.mutual-induct [*THEN conjunct2*, *THEN spec*, *THEN* [*2*] *rev-mp*, *of concl*: - *f*, *standard*, *consumes 1*]

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

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

lemma *forest-def*: $forest(A) == Part(tree\text{-}forest(A),\ Inr)$

⟨*proof*⟩

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

lemma *tree-subset-TF*: *tree*(A) ⊆ *tree-forest*(A)
⟨*proof*⟩

lemma *treeI* [*TC*]: $x \in$ *tree*(A) ==> $x \in$ *tree-forest*(A)
⟨*proof*⟩

lemma *forest-subset-TF*: *forest*(A) ⊆ *tree-forest*(A)
⟨*proof*⟩

lemma *treeI'* [*TC*]: $x \in$ *forest*(A) ==> $x \in$ *tree-forest*(A)
⟨*proof*⟩

lemma *TF-equals-Un*: *tree*(A) ∪ *forest*(A) = *tree-forest*(A)
⟨*proof*⟩

lemma
notes *rews* = *tree-forest.con-defs tree-def forest-def*
shows
tree-forest-unfold: *tree-forest*(A) =
(A × *forest*(A)) + ({*0*} + *tree*(A) × *forest*(A))
— NOT useful, but interesting ...
⟨*proof*⟩

lemma *tree-forest-unfold'*:
tree-forest(A) =
A × *Part*(*tree-forest*(A), λ*w*. *Inr*(*w*)) +
{*0*} + *Part*(*tree-forest*(A), λ*w*. *Inl*(*w*)) ∗ *Part*(*tree-forest*(A), λ*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-forest*(A);
!!$x\ f\ r$. [| $x \in A$; $f \in$ *forest*(A); $r \in C(f)$
|] ==> $b(x,f,r) \in C(Tcons(x,f))$;
$c \in C(Fnil)$;
!!$t\ f\ r1\ r2$. [| $t \in$ *tree*(A); $f \in$ *forest*(A); $r1 \in C(t)$; $r2 \in C(f)$
|] ==> $d(t,f,r1,r2) \in C(Fcons(t,f))$
|] ==> *tree-forest-rec*(b,c,d,z) $\in C(z)$
⟨*proof*⟩

lemma *tree-forest-rec-type*:
[| !!$x\ f\ r$. [| $x \in A$; $f \in forest(A)$; $r \in D(f)$
|] ==> $b(x,f,r) \in C(Tcons(x,f))$;
$c \in D(Fnil)$;
!!$t\ f\ r1\ r2$. [| $t \in tree(A)$; $f \in forest(A)$; $r1 \in C(t)$; $r2 \in D(f)$
|] ==> $d(t,f,r1,r2) \in D(Fcons(t,f))$
|] ==> $(\forall t \in tree(A).\quad tree\text{-}forest\text{-}rec(b,c,d,t) \in C(t)) \wedge$
$(\forall f \in forest(A).\ tree\text{-}forest\text{-}rec(b,c,d,f) \in D(f))$
— Mutually recursive version.
⟨*proof*⟩

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 $(Fcons(t,tf)) = Cons\ (t,\ list\text{-}of\text{-}TF(tf))$

primrec
of-list$([]) = Fnil$
of-list$(Cons(t,l)) = Fcons(t,\ of\text{-}list(l))$

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*))
⟨*proof*⟩

lemma *of-list-type* [*TC*]: *l* ∈ *list*(*tree*(*A*)) ==> *of-list*(*l*) ∈ *forest*(*A*)
⟨*proof*⟩

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*)
⟨*proof*⟩

size.

lemma *size-type* [*TC*]: *z* ∈ *tree-forest*(*A*) ==> *size*(*z*) ∈ *nat*
⟨*proof*⟩

preorder.

lemma *preorder-type* [*TC*]: *z* ∈ *tree-forest*(*A*) ==> *preorder*(*z*) ∈ *list*(*A*)
⟨*proof*⟩

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

lemma *forest-induct* [*consumes 1*, *case-names Fnil Fcons*]:
[| *f* ∈ *forest*(*A*);
R(*Fnil*);
!!*t f*. [| *t* ∈ *tree*(*A*); *f* ∈ *forest*(*A*); *R*(*f*) |] ==> *R*(*Fcons*(*t*,*f*))
|] ==> *R*(*f*)
— Essentially the same as list induction.
⟨*proof*⟩

lemma *forest-iso*: *f* ∈ *forest*(*A*) ==> *of-list*(*list-of-TF*(*f*)) = *f*
⟨*proof*⟩

lemma *tree-list-iso*: *ts*: *list*(*tree*(*A*)) ==> *list-of-TF*(*of-list*(*ts*)) = *ts*
⟨*proof*⟩

Theorems about *map*.

lemma *map-ident*: *z* ∈ *tree-forest*(*A*) ==> *map*(λ*u*. *u*, *z*) = *z*
⟨*proof*⟩

lemma *map-compose*:
z ∈ *tree-forest*(*A*) ==> *map*(*h*, *map*(*j*,*z*)) = *map*(λ*u*. *h*(*j*(*u*)), *z*)
⟨*proof*⟩

Theorems about *size*.

lemma *size-map*: *z* ∈ *tree-forest*(*A*) ==> *size*(*map*(*h*,*z*)) = *size*(*z*)
⟨*proof*⟩

lemma *size-length*: *z* ∈ *tree-forest*(*A*) ==> *size*(*z*) = *length*(*preorder*(*z*))
⟨*proof*⟩

Theorems about *preorder*.

lemma *preorder-map*:
z ∈ *tree-forest*(*A*) ==> *preorder*(*map*(*h*,*z*)) = *List-ZF*.*map*(*h*, *preorder*(*z*))
⟨*proof*⟩

end

6 Infinite branching datatype definitions

theory *Brouwer* **imports** *Main-ZFC* **begin**

6.1 The Brouwer ordinals

consts
brouwer :: *i*

datatype ⊆ *Vfrom*(*0*, *csucc*(*nat*))
brouwer = *Zero* | *Suc* (*b* ∈ *brouwer*) | *Lim* (*h* ∈ *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* ∈ *brouwer*
and *cases*:
P(*Zero*)
!!*b*. [| *b* ∈ *brouwer*; *P*(*b*) |] ==> *P*(*Suc*(*b*))
!!*h*. [| *h* ∈ *nat* −> *brouwer*; ∀ *i* ∈ *nat*. *P*(*h*‘*i*) |] ==> *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* => *i*] => *i*

datatype ⊆ *Vfrom*(*A* ∪ (⋃ *x* ∈ *A*. *B*(*x*)), *csucc*(*nat* ∪ |⋃ *x* ∈ *A*. *B*(*x*)|))
— The union with *nat* ensures that the cardinal is infinite.
Well(*A*, *B*) = *Sup* (*a* ∈ *A*, *f* ∈ *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*) = (Σ *x* ∈ *A*. *B*(*x*) -> *Well*(*A*, *B*))
⟨*proof*⟩

lemma *Well-induct2* [*consumes 1*, *case-names step*]:
assumes *w*: *w* ∈ *Well*(*A*, *B*)
and *step*: !!*a f*. [| *a* ∈ *A*; *f* ∈ *B*(*a*) -> *Well*(*A*,*B*); ∀ *y* ∈ *B*(*a*). *P*(*f*‘*y*) |] ==> *P*(*Sup*(*a*,*f*))
shows *P*(*w*)
— A nicer induction rule than the standard one.
⟨*proof*⟩

lemma *Well-bool-unfold*: *Well*(*bool*, λ*x*. *x*) = *1* + (*1* -> *Well*(*bool*, λ*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** *Main* **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* ⊆ *Pow*(*nat* × *nat*)
intros
horiz: [| *i* ∈ *nat*; *j* ∈ *nat* |] ==> {<*i*,*j*>, <*i*,*succ*(*j*)>} ∈ *domino*
vertl: [| *i* ∈ *nat*; *j* ∈ *nat* |] ==> {<*i*,*j*>, <*succ*(*i*),*j*>} ∈ *domino*
type-intros *empty-subsetI cons-subsetI PowI SigmaI nat-succI*

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

definition
$evnodd :: [i,\ i] => i$ **where**
$evnodd(A,b) == \{z \in A.\ \exists\, i\ j.\ z = <i,j> \wedge (i\ \#+\ j)\ mod\ 2 = b\}$

7.1 Basic properties of evnodd

lemma *evnodd-iff*: $<i,j>: evnodd(A,b) <-> <i,j>: A\ \&\ (i\#+j)\ mod\ 2 = b$
⟨*proof*⟩

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

lemma *Finite-evnodd*: $Finite(X) ==> Finite(evnodd(X,b))$
⟨*proof*⟩

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

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

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

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

7.2 Dominoes

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

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

7.3 Tilings

The union of two disjoint tilings is a tiling

lemma *tiling-UnI*:
  $t \in tiling(A) ==> u \in tiling(A) ==> t\ Int\ u = 0 ==> t\ Un\ u \in tiling(A)$
 ⟨*proof*⟩

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

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

lemma *dominoes-tile-row*:
  $[|\ i \in nat;\ \ n \in nat\ |] ==> \{i\} * (n\ \#+\ n) \in tiling(domino)$
 ⟨*proof*⟩

lemma *dominoes-tile-matrix*:
  $[|\ m \in nat;\ \ n \in nat\ |] ==> m * (n\ \#+\ n) \in tiling(domino)$
 ⟨*proof*⟩

lemma *eq-lt-E*: $[|\ x=y;\ x<y\ |] ==> P$
 ⟨*proof*⟩

theorem *mutil-not-tiling*: $[|\ m \in nat;\ \ n \in nat;$
      $t = (succ(m)\#+succ(m))*(succ(n)\#+succ(n));$
      $t' = t - \{<0,0>\} - \{<succ(m\#+m),\ succ(n\#+n)>\}\ |]$
    $==> t' \notin tiling(domino)$
 ⟨*proof*⟩

end

theory *FoldSet* **imports** *Main* **begin**

consts *fold-set* :: $[i,\ i,\ [i,i]=>i,\ i] => i$

inductive
 domains *fold-set*$(A,\ B,\ f,e) <= Fin(A)*B$
 intros
  emptyI: $e \in B ==> <0,\ e> \in$ *fold-set*$(A,\ B,\ f,e)$
  consI: $[|\ x \in A;\ x \notin C;\ \ <C,y> : fold\text{-}set(A,\ B,f,e);\ f(x,y):B\ |]$
      $==>\ \ <cons(x,C),\ f(x,y)> \in fold\text{-}set(A,\ B,\ f,\ e)$
 type-intros *Fin.intros*

definition
 fold :: $[i,\ [i,i]=>i,\ i,\ i] => i$ (*fold*[-]'(-,-,-')) **where**
  fold$[B](f,e,\ A) ==$ *THE* $x.\ <A,\ x> \in fold\text{-}set(A,\ B,\ f,e)$

definition
  setsum :: $[i=>i,\ i] => i$ **where**

$setsum(g,\ C) == \mathit{if}\ Finite(C)\ \mathit{then}$
$\quad fold[int](\%x\ y.\ g(x)\ \$+\ y,\ \#0,\ C)\ \mathit{else}\ \#0$

inductive-cases *empty-fold-setE*: $<0,\ x> : fold\text{-}set(A,\ B,\ f,e)$
inductive-cases *cons-fold-setE*: $<cons(x,C),\ y> : fold\text{-}set(A,\ B,\ f,e)$

lemma *cons-lemma1*: $[|\ x\notin C;\ x\notin B\ |] ==> cons(x,B)=cons(x,C) <-> B = C$
⟨*proof*⟩

lemma *cons-lemma2*: $[|\ cons(x,\ B)=cons(y,\ C);\ x\neq y;\ x\notin B;\ y\notin C\ |]$
$\quad ==>\ B - \{y\} = C-\{x\}\ \&\ x\in C\ \&\ y\in B$
⟨*proof*⟩

lemma *fold-set-mono-lemma*:
$\quad <C,\ x> : fold\text{-}set(A,\ B,\ f,\ e)$
$\quad ==> ALL\ D.\ A<=D --> <C,\ x> : fold\text{-}set(D,\ B,\ f,\ e)$
⟨*proof*⟩

lemma *fold-set-mono*: $C<=A ==> fold\text{-}set(C,\ B,\ f,\ e) <= fold\text{-}set(A,\ B,\ f,\ e)$
⟨*proof*⟩

lemma *fold-set-lemma*:
$\quad <C,\ x>\in fold\text{-}set(A,\ B,\ f,\ e) ==> <C,\ x>\in fold\text{-}set(C,\ B,\ f,\ e)\ \&\ C<=A$
⟨*proof*⟩

lemma *Diff1-fold-set*:
$\quad [|\ <C-\{x\},y> : fold\text{-}set(A,\ B,\ f,e);\ \ x\in C;\ x\in A;\ f(x,\ y):B\ |]$
$\quad ==> <C,\ f(x,\ y)> : fold\text{-}set(A,\ B,\ f,\ e)$
⟨*proof*⟩

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

lemma (**in** *fold-typing*) *Fin-imp-fold-set*:
$\quad C\in Fin(A) ==> (EX\ x.\ <C,\ x> : fold\text{-}set(A,\ B,\ f,e))$
⟨*proof*⟩

lemma *Diff-sing-imp*:

$[|C - \{b\} = D - \{a\};\ a \neq b;\ b \in C|] ==> C = cons(b,D) - \{a\}$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-set-determ-lemma* [*rule-format*]:
$n \in nat$
$==> ALL\ C.\ |C|<n\ -->$
$(ALL\ x.\ <C,\ x> : fold\text{-}set(A,\ B,\ f,e)-->$
$(ALL\ y.\ <C,\ y> : fold\text{-}set(A,\ B,\ f,e)\ -->\ y=x))$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-set-determ*:
$[|\ <C,\ x> \in fold\text{-}set(A,\ B,\ f,\ e);$
$<C,\ y> \in fold\text{-}set(A,\ B,\ f,\ e)|] ==> y=x$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-equality*:
$<C,y> : fold\text{-}set(A,B,f,e) ==> fold[B](f,e,C) = y$
⟨*proof*⟩

lemma *fold-0* [*simp*]: $e : B ==> fold[B](f,e,0) = e$
⟨*proof*⟩

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

lemma (**in** *fold-typing*) *fold-set-imp-cons*:
$[|\ <C,\ y> : fold\text{-}set(C,\ B,\ f,\ e);\ C : Fin(A);\ c : A;\ c \notin C\ |]$
$==> <cons(c,\ C),\ f(c,y)> : fold\text{-}set(cons(c,\ C),\ B,\ f,\ e)$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-cons-lemma* [*rule-format*]:
$[|\ C : Fin(A);\ c : A;\ c \notin C\ |]$
$==> <cons(c,\ C),\ v> : fold\text{-}set(cons(c,\ C),\ B,\ f,\ e) <->$
$(EX\ y.\ <C,\ y> : fold\text{-}set(C,\ B,\ f,\ e)\ \&\ v = f(c,\ y))$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-cons*:
$[|\ C \in Fin(A);\ c \in A;\ c \notin C|]$
$==> fold[B](f,\ e,\ cons(c,\ C)) = f(c,\ fold[B](f,\ e,\ C))$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-type* [*simp*,*TC*]:
$C \in Fin(A) ==> fold[B](f,e,C):B$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-commute* [*rule-format*]:
$[|\ C \in Fin(A);\ c \in A\ |]$
$==> (\forall\, y \in B.\ f(c,\ fold[B](f,\ y,\ C)) = fold[B](f,\ f(c,\ y),\ C))$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-nest-Un-Int*:
[| $C \in Fin(A)$; $D \in Fin(A)$ |]
==> $fold[B](f, fold[B](f, e, D), C) =$
$fold[B](f, fold[B](f, e, (C\ Int\ D)), C\ Un\ D)$
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-nest-Un-disjoint*:
[| $C \in Fin(A)$; $D \in Fin(A)$; $C\ Int\ D = 0$ |]
==> $fold[B](f,e,C\ Un\ D) = fold[B](f, fold[B](f,e,D), C)$
⟨*proof*⟩

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

7.4 The Operator *setsum*

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

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

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

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

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

lemma *setsum-Un-disjoint*:
[| $Finite(C)$; $Finite(D)$; $C\ Int\ D = 0$ |]
==> $setsum(g, C\ Un\ D) = setsum(g, C)\ \$+\ setsum(g,D)$
⟨*proof*⟩

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

lemma *setsum-UN-disjoint* [*rule-format* (*no-asm*)]:

$Finite(I)$
$\Longrightarrow (\forall i{\in}I.\ Finite(C(i))) \longrightarrow$
$(\forall i{\in}I.\ \forall j{\in}I.\ i{\neq}j \longrightarrow C(i)\ Int\ C(j) = 0) \longrightarrow$
$setsum(f, \bigcup i{\in}I.\ C(i)) = setsum\ (\%i.\ setsum(f,\ C(i)),\ I)$
⟨*proof*⟩

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

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

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

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

lemma *setsum-Un*:
$[|\ Finite(A);\ Finite(B)\ |]$
$\Longrightarrow setsum(f,\ A\ Un\ B) =$
$setsum(f,\ A)\ \$+\ setsum(f,\ B)\ \$-\ setsum(f,\ A\ Int\ B)$
⟨*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*) ==> (∀ *x*∈*A*. #0 $<= *g*(*x*)) −−> #0 $<= *setsum*(*g*, *A*)
⟨*proof*⟩

lemma *g-zpos-imp-setsum-zpos2* [*rule-format*]:
Finite(*A*); ∀ *x*. #0 $<= *g*(*x*) |] ==> #0 $<= *setsum*(*g*, *A*)
⟨*proof*⟩

lemma *g-zspos-imp-setsum-zspos* [*rule-format*]:
Finite(*A*)
==> (∀ *x*∈*A*. #0 $< *g*(*x*)) −−> *A* ≠ *0* −−> (#0 $< *setsum*(*g*, *A*))
⟨*proof*⟩

lemma *setsum-Diff* [*rule-format*]:
Finite(*A*) ==> ∀ *a*. *M*(*a*) = #0 −−> *setsum*(*M*, *A*) = *setsum*(*M*, *A*−{*a*})
⟨*proof*⟩

end

8 The accessible part of a relation

theory *Acc* **imports** *Main* **begin**

Inductive definition of *acc*(*r*); see [**?**].

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

inductive
domains *acc*(*r*) ⊆ *field*(*r*)
intros
vimage: [| *r*−‘‘{*a*}: *Pow*(*acc*(*r*)); *a* ∈ *field*(*r*) |] ==> *a* ∈ *acc*(*r*)
monos *Pow-mono*

The introduction rule must require *a* ∈ *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*)
⟨*proof*⟩

lemma *acc-downward*: [| *b* ∈ *acc*(*r*); <*a*,*b*>: *r* |] ==> *a* ∈ *acc*(*r*)
⟨*proof*⟩

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*)
⟨*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*: $wf(r) ==> field(r) \subseteq acc(r)$
⟨*proof*⟩

lemma *wf-acc-iff*: $wf(r) <-> field(r) \subseteq acc(r)$
⟨*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**
$funrestrict(f,A) == \lambda x \in A.\ f`x$

definition

$multiset :: i => o$ **where**
$multiset(M) == \exists A.\ M \in A -> nat-\{0\}\ \&\ Finite(A)$

definition
$mset\text{-}of :: i=>i$ **where**
$mset\text{-}of(M) == domain(M)$

definition
$munion \quad :: [i, i] => i$ (**infixl** $+\#\ 65$) **where**
$M +\# N == \lambda x \in mset\text{-}of(M)\ Un\ mset\text{-}of(N).$
$\quad if\ x \in mset\text{-}of(M)\ Int\ mset\text{-}of(N)\ then\ (M`x) \#+ (N`x)$
$\quad else\ (if\ x \in mset\text{-}of(M)\ then\ M`x\ else\ N`x)$

definition

$normalize :: i => i$ **where**
$normalize(f) ==$

if $(\exists A.\ f \in A \to nat\ \&\ Finite(A))$ *then*
funrestrict$(f, \{x \in mset\text{-}of(f).\ 0 < f`x\})$
else 0

definition
mdiff :: $[i, i] \Rightarrow i$ (**infixl** $-\#\ 65$) **where**
$M -\# N ==$ *normalize*$(\lambda x \in mset\text{-}of(M).$
if $x \in mset\text{-}of(N)$ *then* $M`x\ \#-\ N`x$ *else* $M`x)$

definition

msingle :: $i \Rightarrow i$ $(\{\#\text{-}\#\})$ **where**
$\{\#a\#\} == \{<a, 1>\}$

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

definition

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

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

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

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

definition

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

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*
⟨*proof*⟩

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

lemma *funrestrict-type2*: [| *f* ∈ *Pi*(*C*,*B*); *A*⊆*C* |] ==> *funrestrict*(*f*,*A*) ∈ *Pi*(*A*,*B*)
⟨*proof*⟩

lemma *funrestrict* [*simp*]: *a* ∈ *A* ==> *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 ∈ *cons*(*a*, *b*) -> *B* ==> *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* ∈ *A* -> *nat*−{*0*}) <-> *f* ∈ *A*->*nat*&(∀ *a* ∈ *A*. *f*‘*a* ∈ *nat* & *0* < *f*‘*a*)
⟨*proof*⟩

lemma *multiset-into-Mult*: [| *multiset*(M); *mset-of*(M)⊆A |] ==> $M \in$ *Mult*(A)
⟨*proof*⟩

lemma *Mult-into-multiset*: $M \in$ *Mult*(A) ==> *multiset*(M) & *mset-of*(M)⊆A
⟨*proof*⟩

lemma *Mult-iff-multiset*: $M \in$ *Mult*(A) <-> *multiset*(M) & *mset-of*(M)⊆A
⟨*proof*⟩

lemma *multiset-iff-Mult-mset-of*: *multiset*(M) <-> $M \in$ *Mult*(*mset-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) ==> *Finite*(*mset-of*(M))
⟨*proof*⟩

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

lemma *mset-is-0-iff*: *multiset*(M) ==> *mset-of*(M)=*0* <-> 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) *Un* *mset-of*(N)
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *munion-multiset* [*simp*]: [| *multiset*(*M*); *multiset*(*N*) |] ==> *multiset*(*M* +# *N*)
⟨*proof*⟩

lemma *mdiff-multiset* [*simp*]: *multiset*(*M* −# *N*)
⟨*proof*⟩

lemma *munion-0* [*simp*]: *multiset*(*M*) ==> *M* +# *0* = *M* & *0* +# *M* = *M*
⟨*proof*⟩

lemma *munion-commute*: *M* +# *N* = *N* +# *M*
⟨*proof*⟩

lemma *munion-assoc*: (*M* +# *N*) +# *K* = *M* +# (*N* +# *K*)
⟨*proof*⟩

lemma *munion-lcommute*: *M* +# (*N* +# *K*) = *N* +# (*M* +# *K*)
⟨*proof*⟩

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

lemma *mdiff-self-eq-0* [*simp*]: *M* −# *M* = *0*
⟨*proof*⟩

lemma *mdiff-0* [*simp*]: *0* −# *M* = *0*

⟨*proof*⟩

lemma *mdiff-0-right* [*simp*]: *multiset*(*M*) ==> *M* −# *0* = *M*
⟨*proof*⟩

lemma *mdiff-union-inverse2* [*simp*]: *multiset*(*M*) ==> *M* +# {#*a*#} −# {#*a*#}
= *M*
⟨*proof*⟩

lemma *mcount-type* [*simp*,*TC*]: *multiset*(*M*) ==> *mcount*(*M*, *a*) ∈ *nat*
⟨*proof*⟩

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*) |]
==> *mcount*(*M* +# *N*, *a*) = *mcount*(*M*, *a*) #+ *mcount* (*N*, *a*)
⟨*proof*⟩

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

lemma *mcount-elem*: [| *multiset*(*M*); *a* ∈ *mset-of*(*M*) |] ==> *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*) ∈ *int*
⟨*proof*⟩

lemma *msize-zpositive*: *multiset*(*M*)==> #*0* $≤ *msize*(*M*)
⟨*proof*⟩

lemma *msize-int-of-nat*: *multiset*(*M*) ==> ∃ *n* ∈ *nat*. *msize*(*M*)= $# *n*
⟨*proof*⟩

lemma *not-empty-multiset-imp-exist*:

[| $M \neq 0$; $multiset(M)$ |] ==> $\exists\, a \in mset\text{-}of(M).\ 0 < mcount(M,\ a)$
⟨*proof*⟩

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

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

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

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

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

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

lemma *munion-eq-0-iff* [*simp*]: [|$multiset(M)$; $multiset(N)$|]==>$(M +\# N = 0)$ <-> $(M = 0\ \&\ N = 0)$
⟨*proof*⟩

lemma *empty-eq-munion-iff* [*simp*]: [|$multiset(M)$; $multiset(N)$|]==>$(0 = M +\# N)$ <-> $(M = 0\ \&\ N = 0)$
⟨*proof*⟩

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

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

lemma *nat-add-eq-1-cases*: [| $m \in nat$; $n \in nat$ |] ==> $(m \#+ n = 1) <-> (m=1 \ \& \ n=0) \mid (m=0 \ \& \ n=1)$
⟨*proof*⟩

lemma *munion-is-single*:
[|*multiset*(M); *multiset*(N)|]
==> $(M +\# N = \{\#a\#\}) <-> (M=\{\#a\#\} \ \& \ N=0) \mid (M = 0 \ \& \ N = \{\#a\#\})$
⟨*proof*⟩

lemma *msingle-is-union*: [| *multiset*(M); *multiset*(N) |]
==> $(\{\#a\#\} = M +\# N) <-> (\{\#a\#\} = M \ \& \ N=0 \mid M = 0 \ \& \ \{\#a\#\} = N)$
⟨*proof*⟩

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

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

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

lemma *nat-le-1-cases*: $n \in nat$ ==> n *le* 1 <-> $(n=0 \mid n=1)$
⟨*proof*⟩

lemma *succ-pred-eq-self*: [| $0<n$; $n \in nat$ |] ==> *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*: !!*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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *munion-single-case1*:
[| *multiset*(*M*); *a* ∉*mset-of*(*M*) |] ==> *M* +# {#*a*#} = *cons*(<*a*, *1*>, *M*)
⟨*proof*⟩

lemma *munion-single-case2*:
[| *multiset*(*M*); *a* ∈ *mset-of*(*M*) |] ==> *M* +# {#*a*#} = *M*(*a*:=*M*‘*a* #+ *1*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *MCollect-multiset* [*simp*]:
multiset(*M*) ==> *multiset*({# *x* ∈ *M*. *P*(*x*)#})
⟨*proof*⟩

lemma *mset-of-MCollect* [*simp*]:
multiset(*M*) ==> *mset-of*({# *x* ∈ *M*. *P*(*x*) #}) ⊆ *mset-of*(*M*)

⟨*proof*⟩

lemma *MCollect-mem-iff* [*iff*]:
$x \in mset\text{-}of(\{\#x \in M.\ P(x)\#\}) <-> x \in mset\text{-}of(M)\ \&\ P(x)$
⟨*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*: $multiset(M) ==> M = \{\#\ x \in M.\ P(x)\ \#\} +\#\ \{\#\ x \in M.\ \sim P(x)\ \#\}$
⟨*proof*⟩

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

lemma *munion-eq-conv-diff*: $[|\ multiset(M);\ multiset(N)\ |]$
$==>\ (M +\#\ \{\#a\#\} = N +\#\ \{\#b\#\}) <->\ (M = N\ \&\ a = b\ |$
$M = N -\#\ \{\#a\#\} +\#\ \{\#b\#\}\ \&\ N = M -\#\ \{\#b\#\} +\#\ \{\#a\#\})$
⟨*proof*⟩

lemma *melem-diff-single*:
$multiset(M) ==>$
$k \in mset\text{-}of(M -\#\ \{\#a\#\}) <-> (k=a\ \&\ 1 < mcount(M,a))\ |\ (k \neq a\ \&\ k \in mset\text{-}of(M))$
⟨*proof*⟩

lemma *munion-eq-conv-exist*:
$[|\ M \in Mult(A);\ N \in Mult(A)\ |]$
$==> (M +\#\ \{\#a\#\} = N +\#\ \{\#b\#\}) <->$
$(M=N\ \&\ a=b\ |\ (\exists K \in Mult(A).\ M= K +\#\ \{\#b\#\}\ \&\ N=K +\#\ \{\#a\#\}))$
⟨*proof*⟩

8.2 Multiset Orderings

lemma *multirel1-type*: $multirel1(A,\ r) \subseteq Mult(A)*Mult(A)$
⟨*proof*⟩

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

lemma *multirel1-iff*:
$<N,\ M> \in multirel1(A,\ r) <->$
$(\exists a.\ a \in A\ \&$
$(\exists M0.\ M0 \in Mult(A)\ \&\ (\exists K.\ K \in Mult(A)\ \&$

$M{=}M0 +\# \{\#a\#\}$ & $N{=}M0 +\# K$ & $(\forall b \in mset\text{-}of(K).\ <b,a> \in r))))$
⟨*proof*⟩

Monotonicity of *multirel1*

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

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

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

8.3 Toward the proof of well-foundedness of multirel1

lemma *not-less-0* [*iff*]: $<M,0> \notin multirel1(A, r)$
⟨*proof*⟩

lemma *less-munion*: $[\!|\ <N, M0 +\# \{\#a\#\}> \in multirel1(A, r);\ M0 \in Mult(A)\ |\!] \Longrightarrow$
$(\exists M.\ <M, M0> \in multirel1(A, r)\ \&\ N = M +\# \{\#a\#\})\ |$
$(\exists K.\ K \in Mult(A)\ \&\ (\forall b \in mset\text{-}of(K).\ <b, a> \in r)\ \&\ N = M0 +\# K)$
⟨*proof*⟩

lemma *multirel1-base*: $[\!|\ M \in Mult(A);\ a \in A\ |\!] \Longrightarrow <M, M +\# \{\#a\#\}> \in multirel1(A, r)$
⟨*proof*⟩

lemma *acc-0*: $acc(0){=}0$
⟨*proof*⟩

lemma *lemma1*: $[\!|\ \forall b \in A.\ <b,a> \in r \longrightarrow$
$(\forall M \in acc(multirel1(A, r)).\ M +\# \{\#b\#\}{:}acc(multirel1(A, r)));$
$M0 \in acc(multirel1(A, r));\ a \in A;$
$\forall M.\ <M,M0> \in multirel1(A, r) \longrightarrow M +\# \{\#a\#\} \in acc(multirel1(A, r))\ |\!]$
$\Longrightarrow M0 +\# \{\#a\#\} \in acc(multirel1(A, r))$
⟨*proof*⟩

lemma *lemma2*: $[\!|\ \forall b \in A.\ <b,a> \in r$
$\longrightarrow (\forall M \in acc(multirel1(A, r)).\ M +\# \{\#b\#\} :acc(multirel1(A, r)));$
$M \in acc(multirel1(A, r));\ a \in A|\!] \Longrightarrow M +\# \{\#a\#\} \in acc(multirel1(A, r))$
⟨*proof*⟩

lemma *lemma3*: $[\!|\ wf[A](r);\ a \in A\ |\!]$
$\Longrightarrow \forall M \in acc(multirel1(A, r)).\ M +\# \{\#a\#\} \in acc(multirel1(A, r))$
⟨*proof*⟩

lemma *lemma4*: $multiset(M) ==> mset\text{-}of(M) \subseteq A --> wf[A](r) --> M \in field(multirel1(A, r)) --> M \in acc(multirel1(A, r))$
⟨*proof*⟩

lemma *all-accessible*: $[|\ wf[A](r);\ M \in Mult(A);\ A \neq 0|] ==> M \in acc(multirel1(A, r))$
⟨*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*: $multirel(A, r) \subseteq Mult(A) * Mult(A)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *one-step-implies-multirel*:
[| $J \neq 0$; $\forall k \in mset\text{-}of(K).\ \exists j \in mset\text{-}of(J).\ <k,j> \in r$;
$I \in Mult(A)$; $J \in Mult(A)$; $K \in Mult(A)$ |]
==> $<I+\#K, I+\#J> \in multirel(A, r)$
⟨*proof*⟩

lemma *multirel-irrefl-lemma*:
$Finite(A)$ ==> $part\text{-}ord(A, r)$ −−> $(\forall x \in A.\ \exists y \in A.\ <x,y> \in r)$ −−>$A=0$
⟨*proof*⟩

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

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

lemma *multirel-trans*:
[| $<M, N> \in multirel(A, r)$; $<N, K> \in multirel(A, r)$ |] ==> $<M, K> \in multirel(A,r)$
⟨*proof*⟩

lemma *trans-multirel*: $trans(multirel(A,r))$
⟨*proof*⟩

lemma *part-ord-multirel*: *part-ord*(*A*,*r*) ==> *part-ord*(*Mult*(*A*), *multirel*(*A*, *r*))
⟨*proof*⟩

lemma *munion-multirel1-mono*:
[|<*M*,*N*> ∈ *multirel1*(*A*, *r*); *K* ∈ *Mult*(*A*) |] ==> <*K* +# *M*, *K* +# *N*> ∈ *multirel1*(*A*, *r*)
⟨*proof*⟩

lemma *munion-multirel-mono2*:
[| <*M*, *N*> ∈ *multirel*(*A*, *r*); *K* ∈ *Mult*(*A*) |]==><*K* +# *M*, *K* +# *N*> ∈ *multirel*(*A*, *r*)
⟨*proof*⟩

lemma *munion-multirel-mono1*:
[|<*M*, *N*> ∈ *multirel*(*A*, *r*); *K* ∈ *Mult*(*A*)|] ==> <*M* +# *K*, *N* +# *K*> ∈ *multirel*(*A*, *r*)
⟨*proof*⟩

lemma *munion-multirel-mono*:
[|<*M*,*K*> ∈ *multirel*(*A*, *r*); <*N*,*L*> ∈ *multirel*(*A*, *r*)|]
==> <*M* +# *N*, *K* +# *L*> ∈ *multirel*(*A*, *r*)
⟨*proof*⟩

8.4 Ordinal Multisets

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

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

lemma *omultiset-is-multiset* [*simp*]: *omultiset*(*M*) ==> *multiset*(*M*)
⟨*proof*⟩

lemma *munion-omultiset* [*simp*]: [| *omultiset*(*M*); *omultiset*(*N*) |] ==> *omultiset*(*M* +# *N*)
⟨*proof*⟩

lemma *mdiff-omultiset* [*simp*]: *omultiset*(*M*) ==> *omultiset*(*M* −# *N*)
⟨*proof*⟩

lemma *irrefl-Memrel*: *Ord*(*i*) ==> *irrefl*(*field*(*Memrel*(*i*)), *Memrel*(*i*))
⟨*proof*⟩

lemma *trans-iff-trans-on*: *trans*(*r*) <-> *trans*[*field*(*r*)](*r*)

⟨*proof*⟩

lemma *part-ord-Memrel*: *Ord*(*i*) ==>*part-ord*(*field*(*Memrel*(*i*)), *Memrel*(*i*))
⟨*proof*⟩

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

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

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

lemma *mless-trans*: [| *K* <# *M*; *M* <# *N* |] ==> *K* <# *N*
⟨*proof*⟩

lemma *mless-not-sym*: *M* <# *N* ==> ~ *N* <# *M*
⟨*proof*⟩

lemma *mless-asym*: [| *M* <# *N*; ~*P* ==> *N* <# *M* |] ==> *P*
⟨*proof*⟩

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

lemma *mle-antisym*:
[| *M* <#= *N*; *N* <#= *M* |] ==> *M* = *N*
⟨*proof*⟩

lemma *mle-trans*: [| *K* <#= *M*; *M* <#= *N* |] ==> *K* <#= *N*
⟨*proof*⟩

lemma *mless-le-iff*: *M* <# *N* <-> (*M* <#= *N* & *M* ≠ *N*)
⟨*proof*⟩

lemma *munion-less-mono2*: [| *M* <# *N*; *omultiset*(*K*) |] ==> *K* +# *M* <# *K* +# *N*
⟨*proof*⟩

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

lemma *mless-imp-omultiset*: $M <\# N$ ==> *omultiset*(M) & *omultiset*(N)
⟨*proof*⟩

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

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

lemma *mle-mono*: [| $M <\#= K$; $N <\#= L$ |] ==> $M +\# N <\#= K +\# L$
⟨*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** *Main* **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* ∈ *A*->*B*; *x* ∈ *A*; *xs*: *list*(*A*) |]
==> *rmap*(*f*) ‘ *Cons*(*x*,*xs*) = *Cons*(*f*‘*x*, *rmap*(*f*)‘*xs*)
⟨*proof*⟩

end

10 Meta-theory of propositional logic

theory *PropLog* **imports** *Main* **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* ∈ *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*
⟨*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**
  $H \models p == \forall t.\ (\forall q \in H.\ \textit{is-true}(q,t)) \dashrightarrow \textit{is-true}(p,t)$

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*: $G \subseteq H \Longrightarrow \textit{thms}(G) \subseteq \textit{thms}(H)$
  ⟨*proof*⟩

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!
  ⟨*proof*⟩

lemma *thms-I*: *p* ∈ *propn* ==> *H* |− *p*=>*p*
  — Rule is called *I* for Identity Combinator, not for Introduction.
  ⟨*proof*⟩

10.4.1 Weakening, left and right

lemma *weaken-left*: [| *G* ⊆ *H*; *G*|−*p* |] ==> *H*|−*p*
  — Order of premises is convenient with *THEN*
  ⟨*proof*⟩

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

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

lemma *weaken-right*: [| *H* |− *q*; *p* ∈ *propn* |] ==> *H* |− *p*=>*q*
  ⟨*proof*⟩

10.4.2 The deduction theorem

theorem *deduction*: [| *cons*(*p*,*H*) |− *q*; *p* ∈ *propn* |] ==> *H* |− *p*=>*q*

⟨*proof*⟩

10.4.3 The cut rule

lemma *cut*: [| *H*|−*p*; *cons*(*p*,*H*) |− *q* |] ==> *H* |− *q*
⟨*proof*⟩

lemma *thms-FlsE*: [| *H* |− *Fls*; *p* ∈ *propn* |] ==> *H* |− *p*
⟨*proof*⟩

lemma *thms-notE*: [| *H* |− *p*=>*Fls*; *H* |− *p*; *q* ∈ *propn* |] ==> *H* |− *q*
⟨*proof*⟩

10.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: *H* |− *p* ==> *H* |= *p*
⟨*proof*⟩

10.5 Completeness

10.5.1 Towards the completeness proof

lemma *Fls-Imp*: [| *H* |− *p*=>*Fls*; *q* ∈ *propn* |] ==> *H* |− *p*=>*q*
⟨*proof*⟩

lemma *Imp-Fls*: [| *H* |− *p*; *H* |− *q*=>*Fls* |] ==> *H* |− (*p*=>*q*)=>*Fls*
⟨*proof*⟩

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.
⟨*proof*⟩

lemma *logcon-thms-p*: [| *p* ∈ *propn*; *0* |= *p* |] ==> *hyps*(*p*,*t*) |− *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* ∈ *propn*; *q* ∈ *propn* |] ==> *H* |− (*p*=>*q*) => ((*p*=>*Fls*)=>*q*) => *q*
⟨*proof*⟩

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
⟨*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*:

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

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

end

11 Lists of n elements

theory *ListN* **imports** *Main* **begin**

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

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

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

lemma *listn-iff*: $<n,l> \in listn(A) <-> l \in list(A) \ \&\ length(l)=n$
⟨*proof*⟩

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

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

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

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

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

end

12 Combinatory Logic example: the Church-Rosser Theorem

theory *Comb* **imports** *Main* **begin**

Curiously, combinators do not include free variables.
Example taken from [**?**].

12.1 Definitions

Datatype definition of combinators S and K.

consts *comb* :: *i*
datatype *comb* =
  K
| *S*
| *app* (*p* ∈ *comb*, *q* ∈ *comb*) (**infixl** @@ *90*)

notation (*xsymbols*)
  app (**infixl** · *90*)

Inductive definition of contractions, *−1−>* and (multi-step) reductions, *−−−>*.

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−1−>q*; *r* ∈ *comb* |] ==> *p*·*r* *−1−>* *q*·*r*
    Ap2: [| *p−1−>q*; *r* ∈ *comb* |] ==> *r*·*p* *−1−>* *r*·*q*
  type-intros *comb.intros*

Inductive definition of parallel contractions, *=1=>* and (multi-step) parallel reductions, *===>*.

consts
  parcontract :: *i*

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

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

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

Misc definitions.

definition
$I :: i$ **where**
$I == S{\cdot}K{\cdot}K$

definition
diamond :: $i \Rightarrow o$ **where**
$diamond(r) ==$
$\forall x\ y.\ \langle x,y\rangle{\in}r \longrightarrow (\forall y'.\ \langle x,y'\rangle{\in}r \longrightarrow (\exists z.\ \langle y,z\rangle{\in}r\ \&\ \langle y',z\rangle \in r))$

12.2 Transitive closure preserves the Church-Rosser property

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

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

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

declare *comb.intros* [*intro*!]

12.3 Results about Contraction

For type checking: replaces $a\ -1\rightarrow\ b$ 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 $p \in comb \Longrightarrow I \cdot p \longrightarrow p$
— 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*!]: $K\ -1\rightarrow\ r$
and *S-contractE* [*elim*!]: $S\ -1\rightarrow\ r$
and *Ap-contractE* [*elim*!]: $p \cdot q\ -1\rightarrow\ r$

lemma *I-contract-E*: $I\ -1\rightarrow\ r \Longrightarrow P$
⟨*proof*⟩

lemma *K1-contractD*: $K \cdot p\ -1\rightarrow\ r \Longrightarrow (\exists q.\ r = K \cdot q\ \&\ p\ -1\rightarrow\ q)$
⟨*proof*⟩

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

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

Counterexample to the diamond property for $-1\rightarrow$.

lemma *KIII-contract1*: $K \cdot I \cdot (I \cdot I)\ -1\rightarrow\ I$
⟨*proof*⟩

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

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

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

12.5 Results about Parallel Contraction

For type checking: replaces $a\ =1=>\ b$ 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*!]: $K\ =1=>\ r$
and *S-parcontractE* [*elim*!]: $S\ =1=>\ r$
and *Ap-parcontractE* [*elim*!]: $p{\cdot}q\ =1=>\ r$

declare *parcontract.intros* [*intro*]

12.6 Basic properties of parallel contraction

lemma *K1-parcontractD* [*dest*!]:
$K{\cdot}p\ =1=>\ r\ ==>\ (\exists p'.\ r = K{\cdot}p'\ \&\ p\ =1=>\ p')$
⟨*proof*⟩

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

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

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

Equivalence of $p \dashrightarrow q$ and $p \Longrightarrow q$.

lemma *contract-imp-parcontract*: $p-1->q \Longrightarrow p=1=>q$
⟨*proof*⟩

lemma *reduce-imp-parreduce*: $p--->q \Longrightarrow p===>q$
⟨*proof*⟩

lemma *parcontract-imp-reduce*: $p=1=>q \Longrightarrow p--->q$
⟨*proof*⟩

lemma *parreduce-imp-reduce*: $p===>q \Longrightarrow p--->q$
⟨*proof*⟩

lemma *parreduce-iff-reduce*: $p===>q <-> p--->q$
⟨*proof*⟩

end

13 Primitive Recursive Functions: the inductive definition

theory *Primrec* **imports** *Main* **begin**

Proof adopted from [**?**].
See also [**?**, 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*: $[|\ x \in nat;\ \ l \in list(nat)\ |] ==> SC\ `\ (Cons(x,l)) = succ(x)$
⟨*proof*⟩

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

lemma *PROJ-0*: $[|\ x \in nat;\ \ l \in list(nat)\ |] ==> PROJ(0)\ `\ (Cons(x,l)) = x$
⟨*proof*⟩

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

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

lemma *PREC-succ*:
$[|\ x \in nat;\ \ l \in list(nat)\ |]$
$==> PREC(f,g)\ `\ (Cons(succ(x),l)) =$
$g\ `\ Cons(PREC(f,g)`(Cons(x,l)),\ Cons(x,l))$
⟨*proof*⟩

13.2 Inductive definition of the PR functions

consts
$prim\text{-}rec :: i$

inductive
domains $prim\text{-}rec \subseteq list(nat)->nat$
intros
$SC \in prim\text{-}rec$
$k \in nat ==> CONSTANT(k) \in prim\text{-}rec$
$i \in nat ==> PROJ(i) \in prim\text{-}rec$
$[|\ g \in prim\text{-}rec;\ fs \in list(prim\text{-}rec)\ |] ==> COMP(g,fs) \in prim\text{-}rec$

[| $f \in prim\text{-}rec$; $g \in prim\text{-}rec$ |] ==> $PREC(f,g) \in prim\text{-}rec$
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*]: $c \in prim\text{-}rec$ ==> $c \in list(nat)$ -> *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*]: $i \in nat$ ==> $ACK(i) \in prim\text{-}rec$
⟨*proof*⟩

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

13.3 Ackermann's function cases

lemma *ack-0*: $j \in nat$ ==> $ack(0,j) = succ(j)$
— PROPERTY A 1
⟨*proof*⟩

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

lemma *ack-succ-succ*:
[| $i \in nat$; $j \in nat$ |] ==> $ack(succ(i),\ succ(j)) = ack(i,\ ack(succ(i),\ j))$
— PROPERTY A 3
⟨*proof*⟩

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

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

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

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

lemma *ack-le-mono2*: [|$j \leq k$; $i \in nat$; $k \in nat$|] ==> $ack(i,j) \leq ack(i,k)$
— PROPERTY A 5', monotonicity for $\leq$
⟨*proof*⟩

lemma *ack2-le-ack1*:
[| $i \in nat$; $j \in nat$ |] ==> $ack(i, succ(j)) \leq ack(succ(i), j)$
— PROPERTY A 6
⟨*proof*⟩

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
⟨*proof*⟩

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

lemma *ack-le-mono1*: [| $i \leq j$; $j \in nat$; $k \in nat$ |] ==> $ack(i,k) \leq ack(j,k)$
— PROPERTY A 7', monotonicity for $\leq$
⟨*proof*⟩

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

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

lemma *ack-nest-bound*:
[| $i1 \in nat$; $i2 \in nat$; $j \in nat$ |]
==> $ack(i1, ack(i2,j)) < ack(succ(succ(i1\#+i2)), j)$
— PROPERTY A 10
⟨*proof*⟩

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

lemma *ack-add-bound2*:
[| $i < ack(k,j)$; $j \in nat$; $k \in nat$ |]
==> $i\#+j < ack(succ(succ(succ(succ(k)))), j)$

— PROPERTY A 12.
— Article uses existential quantifier but the ALF proof used k #+ *integ-of*(*Pls BIT 1 BIT 0 BIT 0*).
— 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*: [| $i \in nat$; $j \in nat$ |] ==> $i < ack(i,j)$
— PROPERTY A 4'? Extra lemma needed for *CONSTANT* case, constant functions.
⟨*proof*⟩

lemma *CONSTANT-case*:
[| $l \in list(nat)$; $k \in nat$ |] ==> $CONSTANT(k)\ `\ l < ack(k,\ list\text{-}add(l))$
⟨*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*:
[| $kg \in nat$;
$\forall l \in list(nat).\ g`l < ack(kg,\ list\text{-}add(l))$;
$fs \in list(\{f \in prim\text{-}rec\ .$
$\exists kf \in nat.\ \forall l \in list(nat).$
$f`l < ack(kf,\ list\text{-}add(l))\})$ |]
$\Longrightarrow \exists k \in nat.\ \forall l \in list(nat).\ COMP(g,fs)`l < ack(k,\ list\text{-}add(l))$
⟨*proof*⟩

PREC case.

lemma *PREC-case-lemma*:
[| $\forall l \in list(nat).\ f`l$ #+ $list\text{-}add(l) < ack(kf,\ list\text{-}add(l))$;
$\forall l \in list(nat).\ g`l$ #+ $list\text{-}add(l) < ack(kg,\ list\text{-}add(l))$;
$f \in prim\text{-}rec$; $kf \in nat$;
$g \in prim\text{-}rec$; $kg \in nat$;

$l \in list(nat)$ |]
==> $PREC(f,g)`l$ #+ $list\text{-}add(l) < ack(succ(kf\#+kg), list\text{-}add(l))$
⟨*proof*⟩

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

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