Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

October 24, 2026

Contents

1 Sample datatype definitions **2**
1.1 A type with four constructors . . . 3
1.2 Example of a big enumeration type . . . 3

2 Binary trees **3**
2.1 Datatype definition . . . 4
2.2 Number of nodes, with an example of tail-recursion . . . 4
2.3 Number of leaves . . . 5
2.4 Reflecting trees . . . 5

3 Terms over an alphabet **6**

4 Datatype definition n-ary branching trees **10**

5 Trees and forests, a mutually recursive type definition **12**
5.1 Datatype definition . . . 12
5.2 Operations . . . 14

6 Infinite branching datatype definitions **16**
6.1 The Brouwer ordinals . . . 16
6.2 The Martin-Löf wellordering type . . . 16

7 The Mutilated Chess Board Problem, formalized inductively **17**
7.1 Basic properties of evnodd . . . 18
7.2 Dominoes . . . 18
7.3 Tilings . . . 18
7.4 The Operator *setsum* . . . 22

8 The accessible part of a relation **24**
8.1 Properties of the original "restrict" from ZF.thy 27
8.2 Multiset Orderings 34
8.3 Toward the proof of well-foundedness of multirel1 35
8.4 Ordinal Multisets 38

9 An operator to "map" a relation over a list **40**

10 Meta-theory of propositional logic **41**
10.1 The datatype of propositions 41
10.2 The proof system 41
10.3 The semantics 42
10.3.1 Semantics of propositional logic. 42
10.3.2 Logical consequence 42
10.4 Proof theory of propositional logic 42
10.4.1 Weakening, left and right 43
10.4.2 The deduction theorem 43
10.4.3 The cut rule 43
10.4.4 Soundness of the rules wrt truth-table semantics . . . 43
10.5 Completeness 44
10.5.1 Towards the completeness proof 44
10.5.2 Completeness – lemmas for reducing the set of assumptions 44
10.5.3 Completeness theorem 45

11 Lists of n elements **45**

12 Combinatory Logic example: the Church-Rosser Theorem **46**
12.1 Definitions 46
12.2 Transitive closure preserves the Church-Rosser property . . . 48
12.3 Results about Contraction 48
12.4 Non-contraction results 48
12.5 Results about Parallel Contraction 49
12.6 Basic properties of parallel contraction 50

13 Primitive Recursive Functions: the inductive definition **50**
13.1 Basic definitions 50
13.2 Inductive definition of the PR functions 52
13.3 Ackermann's function cases 52
13.4 Main result 54

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 $enum =$
    $C00 \mid C01 \mid C02 \mid C03 \mid C04 \mid C05 \mid C06 \mid C07 \mid C08 \mid C09$
  | $C10 \mid C11 \mid C12 \mid C13 \mid C14 \mid C15 \mid C16 \mid C17 \mid C18 \mid C19$
  | $C20 \mid C21 \mid C22 \mid C23 \mid C24 \mid C25 \mid C26 \mid C27 \mid C28 \mid C29$
  | $C30 \mid C31 \mid C32 \mid C33 \mid C34 \mid C35 \mid C36 \mid C37 \mid C38 \mid C39$
  | $C40 \mid C41 \mid C42 \mid C43 \mid C44 \mid C45 \mid C46 \mid C47 \mid C48 \mid C49$
  | $C50 \mid C51 \mid C52 \mid C53 \mid C54 \mid C55 \mid C56 \mid C57 \mid C58 \mid C59$

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-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) ∈ *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) ∈ $bt(A)$
⟨*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** *ZF* **begin**

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

consts
  term :: $i \Rightarrow i$

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

declare *Apply* [*TC*]

definition
  term-rec :: [i, [i, i, i] $\Rightarrow$ i] $\Rightarrow$ i **where**
  term-rec(t,d) $\equiv$
    Vrec(t, $\lambda t\ g$. *term-case*($\lambda x\ zs$. $d(x, zs, map(\lambda z.\ g`z, zs))$, t))

definition
  term-map :: [$i \Rightarrow i$, i] $\Rightarrow$ i **where**
  term-map(f,t) $\equiv$ *term-rec*(t, $\lambda x\ zs\ rs$. *Apply*($f(x)$, rs))

definition
  term-size :: $i \Rightarrow i$ **where**
  term-size(t) $\equiv$ *term-rec*(t, $\lambda x\ zs\ rs$. *succ*(*list-add*(rs)))

definition
  reflect :: $i \Rightarrow i$ **where**
  reflect(t) $\equiv$ *term-rec*(t, $\lambda x\ zs\ rs$. *Apply*(x, *rev*(rs)))

definition
  preorder :: $i \Rightarrow i$ **where**
  preorder(t) $\equiv$ *term-rec*(t, $\lambda x\ zs\ rs$. *Cons*(x, *flat*(rs)))

definition
  postorder :: $i \Rightarrow i$ **where**
  postorder(t) $\equiv$ *term-rec*(t, $\lambda x\ zs\ rs$. *flat*(rs) @ [x])

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

lemma *term-induct2*:
  ⟦$t \in$ *term*(A);
    $\bigwedge x$. ⟦$x \in A$⟧ $\Longrightarrow$ P(*Apply*(x,*Nil*));
    $\bigwedge x\ z\ zs$. ⟦$x \in A$; $z \in$ *term*(A); zs: *list*(*term*(A)); P(*Apply*(x,zs))

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

lemma *term-induct-eqn* [*consumes 1*, *case-names Apply*]:
$[\![t \in term(A);$
$\bigwedge x\ zs.\ [\![x \in A;\ \ zs\colon list(term(A));\ \ map(f,zs) = map(g,zs)]\!] \Longrightarrow$
$f(Apply(x,zs)) = g(Apply(x,zs))$
$\rrbracket \Longrightarrow f(t) = g(t)$
— 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: $\bigwedge x\ zs\ r.\ [\![x \in A;\ \ zs\colon list(term(A));$
$r \in list(\bigcup t \in term(A).\ C(t))]\!]$
$\Longrightarrow d(x,\ zs,\ r)\colon C(Apply(x,zs))$
shows $term\text{-}rec(t,d) \in C(t)$
— Slightly odd typing condition on r in the second premise!
⟨*proof*⟩

lemma *def-term-rec*:

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

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

term-map.

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

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

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

term-size.

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

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

reflect.

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

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

preorder.

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

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

datatype *ntree*(A) = *Branch* ($a \in A$, $h \in (\bigcup n \in nat.\ n \mathrel{->} ntree(A))$)
  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* ($a \in A$, $h \in maptree(A) \mathrel{-||>} maptree(A)$)
  monos *FiniteFun-mono1* — Use monotonicity in BOTH args
  type-intros *FiniteFun-univ1* [*THEN subsetD*]

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

definition
  ntree-rec :: $[[i, i, i] \Rightarrow i, i] \Rightarrow i$ **where**
  ntree-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 :: $i \Rightarrow i$ **where**
  ntree-copy(z) ≡ *ntree-rec*($\lambda x\ h\ r.\ Branch(x,r)$, z)

ntree

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

lemma *ntree-induct* [*consumes 1*, *case-names Branch*, *induct set*: *ntree*]:
  assumes *t*: $t \in ntree(A)$
    and *step*: $\bigwedge x\ n\ h.\ [\![x \in A;\ \ n \in nat;\ \ h \in n \mathrel{->} ntree(A);\ \ \forall i \in n.\ P(h`i)\ ]\!] \Longrightarrow P(Branch(x,h))$
  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* × (*B* −||> *maptree2*(*A*, *B*))
⟨*proof*⟩

lemma *maptree2-induct* [*consumes 1*, *induct set*: *maptree2*]:
assumes *t*: *t* ∈ *maptree2*(*A*, *B*)
and *step*: ⋀*x n h*. ⟦*x* ∈ *A*; *h* ∈ *B* −||> *maptree2*(*A*,*B*); ∀ *y* ∈ *range*(*h*). *P*(*y*) ⟧ ⟹ *P*(*Sons2*(*x*,*h*))
shows *P*(*t*)
⟨*proof*⟩

end

5 Trees and forests, a mutually recursive type definition

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

5.1 Datatype definition

consts
tree :: *i* ⇒ *i*
forest :: *i* ⇒ *i*
tree-forest :: *i* ⇒ *i*

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

lemmas *tree'induct* =
tree-forest.mutual-induct [*THEN conjunct1*, *THEN spec*, *THEN* [*2*] *rev-mp*, *of concl*: - *t*, *consumes 1*]
and *forest'induct* =
tree-forest.mutual-induct [*THEN conjunct2*, *THEN spec*, *THEN* [*2*] *rev-mp*, *of concl*: - *f*, *consumes 1*]
for *t f*

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

lemma *tree-def*: *tree*(*A*) ≡ *Part*(*tree-forest*(*A*), *Inl*)
⟨*proof*⟩

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

$]\!] \Longrightarrow (\forall\, t \in \mathit{tree}(A).\quad \mathit{tree\text{-}forest\text{-}rec}(b,c,d,t) \in C(t)) \wedge$
$(\forall\, f \in \mathit{forest}(A).\ \mathit{tree\text{-}forest\text{-}rec}(b,c,d,f) \in D(f))$
— Mutually recursive version.
⟨*proof*⟩

5.2 Operations

consts
$\mathit{map} :: [i \Rightarrow i,\ i] \Rightarrow i$
$\mathit{size} :: i \Rightarrow i$
$\mathit{preorder} :: i \Rightarrow i$
$\mathit{list\text{-}of\text{-}TF} :: i \Rightarrow i$
$\mathit{of\text{-}list} :: i \Rightarrow i$
$\mathit{reflect} :: i \Rightarrow i$

primrec
$\mathit{list\text{-}of\text{-}TF}\ (\mathit{Tcons}(x,f)) = [\mathit{Tcons}(x,f)]$
$\mathit{list\text{-}of\text{-}TF}\ (\mathit{Fnil}) = []$
$\mathit{list\text{-}of\text{-}TF}\ (\mathit{Fcons}(t,tf)) = \mathit{Cons}\ (t,\ \mathit{list\text{-}of\text{-}TF}(tf))$

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

primrec
$\mathit{map}\ (h,\ \mathit{Tcons}(x,f)) = \mathit{Tcons}(h(x),\ \mathit{map}(h,f))$
$\mathit{map}\ (h,\ \mathit{Fnil}) = \mathit{Fnil}$
$\mathit{map}\ (h,\ \mathit{Fcons}(t,tf)) = \mathit{Fcons}\ (\mathit{map}(h,\ t),\ \mathit{map}(h,\ tf))$

primrec
$\mathit{size}\ (\mathit{Tcons}(x,f)) = \mathit{succ}(\mathit{size}(f))$
$\mathit{size}\ (\mathit{Fnil}) = 0$
$\mathit{size}\ (\mathit{Fcons}(t,tf)) = \mathit{size}(t)\ \#{+}\ \mathit{size}(tf)$

primrec
$\mathit{preorder}\ (\mathit{Tcons}(x,f)) = \mathit{Cons}(x,\ \mathit{preorder}(f))$
$\mathit{preorder}\ (\mathit{Fnil}) = \mathit{Nil}$
$\mathit{preorder}\ (\mathit{Fcons}(t,tf)) = \mathit{preorder}(t)\ @\ \mathit{preorder}(tf)$

primrec
$\mathit{reflect}\ (\mathit{Tcons}(x,f)) = \mathit{Tcons}(x,\ \mathit{reflect}(f))$
$\mathit{reflect}\ (\mathit{Fnil}) = \mathit{Fnil}$
$\mathit{reflect}\ (\mathit{Fcons}(t,tf)) =$
$\mathit{of\text{-}list}\ (\mathit{list\text{-}of\text{-}TF}\ (\mathit{reflect}(tf))\ @\ \mathit{Cons}(\mathit{reflect}(t),\ \mathit{Nil}))$

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 $\bigwedge x.\ x \in A \Longrightarrow h(x)$: 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*: $list(tree(A)) \Longrightarrow list\text{-}of\text{-}TF(of\text{-}list(ts)) = ts$
⟨*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* $(h \in nat \to$ *brouwer*$)$
monos *Pi-mono*
type-intros *inf-datatype-intros*

lemma *brouwer-unfold*: *brouwer* $= \{0\} +$ *brouwer* $+ (nat \to$ *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*$(b))$
$\bigwedge h.\ [\![h \in nat \to$ *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* $(a \in A,\ f \in B(a) \to$ *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*:
$[\![i \in nat;\ \ n \in nat]\!] \Longrightarrow \{i\} * (n\ \#+\ n) \in tiling(domino)$
⟨*proof*⟩

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

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

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

end

theory *FoldSet* **imports** *ZF* **begin**

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

inductive
domains *fold-set*$(A,\ B,\ f,e) \subseteq Fin(A)*B$
intros
emptyI: $e \in B \Longrightarrow \langle 0,\ e\rangle \in$ *fold-set*$(A,\ B,\ f,e)$
consI: $[\![x \in A;\ x \notin C;\ \ \langle C,y\rangle \in$ *fold-set*$(A,\ B,f,e);\ f(x,y):B]\!]$
$\Longrightarrow\ \ <cons(x,C),\ f(x,y)> \in$ *fold-set*$(A,\ B,\ f,\ e)$
type-intros *Fin.intros*

definition
fold :: $[i,\ [i,i]\Rightarrow i,\ i,\ i] \Rightarrow i$ (‹*fold*[-]'(-,-,-')›) **where**
fold$[B](f,e,\ A) \equiv$ *THE x.* $\langle A,\ x\rangle \in$ *fold-set*$(A,\ B,\ f,e)$

definition
setsum :: $[i \Rightarrow i,\ i] \Rightarrow i$ **where**
setsum$(g,\ C) \equiv$ *if Finite*(C) *then*
fold$[int](\lambda x\ y.\ g(x)\ \$+\ y,\ \#0,\ C)$ *else* $\#0$

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

lemma *cons-lemma1*: $[\![x \notin C;\ x \notin B]\!] \Longrightarrow cons(x,B)=cons(x,C) \longleftrightarrow B = C$

⟨*proof*⟩

lemma *cons-lemma2*: ⟦*cons*(*x*, *B*)=*cons*(*y*, *C*); *x*≠*y*; *x*∉*B*; *y*∉*C*⟧
⟹ *B* − {*y*} = *C*−{*x*} ∧ *x*∈*C* ∧ *y*∈*B*
⟨*proof*⟩

lemma *fold-set-mono-lemma*:
⟨*C*, *x*⟩ ∈ *fold-set*(*A*, *B*, *f*, *e*)
⟹ ∀ *D*. *A*<=*D* ⟶ ⟨*C*, *x*⟩ ∈ *fold-set*(*D*, *B*, *f*, *e*)
⟨*proof*⟩

lemma *fold-set-mono*: *C*<=*A* ⟹ *fold-set*(*C*, *B*, *f*, *e*) ⊆ *fold-set*(*A*, *B*, *f*, *e*)
⟨*proof*⟩

lemma *fold-set-lemma*:
⟨*C*, *x*⟩∈*fold-set*(*A*, *B*, *f*, *e*) ⟹ ⟨*C*, *x*⟩∈*fold-set*(*C*, *B*, *f*, *e*) ∧ *C*<=*A*
⟨*proof*⟩

lemma *Diff1-fold-set*:
⟦<*C*−{*x*},*y*> ∈ *fold-set*(*A*, *B*, *f*,*e*); *x*∈*C*; *x*∈*A*; *f*(*x*, *y*):*B*⟧
⟹ <*C*, *f*(*x*, *y*)> ∈ *fold-set*(*A*, *B*, *f*, *e*)
⟨*proof*⟩

locale *fold-typing* =
fixes *A* **and** *B* **and** *e* **and** *f*
assumes *ftype* [*intro*,*simp*]: ⟦*x* ∈ *A*; *y* ∈ *B*⟧ ⟹ *f*(*x*,*y*) ∈ *B*
and *etype* [*intro*,*simp*]: *e* ∈ *B*
and *fcomm*: ⟦*x* ∈ *A*; *y* ∈ *A*; *z* ∈ *B*⟧ ⟹ *f*(*x*, *f*(*y*, *z*))=*f*(*y*, *f*(*x*, *z*))

lemma (**in** *fold-typing*) *Fin-imp-fold-set*:
C∈*Fin*(*A*) ⟹ (∃ *x*. ⟨*C*, *x*⟩ ∈ *fold-set*(*A*, *B*, *f*,*e*))
⟨*proof*⟩

lemma *Diff-sing-imp*:
⟦*C* − {*b*} = *D* − {*a*}; *a* ≠ *b*; *b* ∈ *C*⟧ ⟹ *C* = *cons*(*b*,*D*) − {*a*}
⟨*proof*⟩

lemma (**in** *fold-typing*) *fold-set-determ-lemma* [*rule-format*]:
n∈*nat*
⟹ ∀ *C*. |*C*|<*n* ⟶
(∀ *x*. ⟨*C*, *x*⟩ ∈ *fold-set*(*A*, *B*, *f*,*e*)⟶
(∀ *y*. ⟨*C*, *y*⟩ ∈ *fold-set*(*A*, *B*, *f*,*e*) ⟶ *y*=*x*))
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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* (‹(‹*open-block notation*=‹*mixfix multiset*››{#-#})›) **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
$melem :: [i,i] \Rightarrow o \quad (\langle(\langle notation=\langle infix\ :\#\rangle\rangle\text{-}/\ :\#\ \text{-})\rangle\ [50,\ 51]\ 50)$ **where**
$a :\#\ M \equiv a \in mset\text{-}of(M)$

syntax
$\text{-}MColl :: [pttrn,\ i,\ o] \Rightarrow i\ (\langle(\langle indent=1\ notation=\langle mixfix\ multiset\ comprehension\rangle\rangle\{\#\ \text{-} \in \text{-}./\ \text{-}\#\})\rangle)$
syntax-consts
$\text{-}MColl \rightleftharpoons MCollect$
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**
$multirel(A,\ r) \equiv multirel1(A,\ r)^{+}$

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

definition
$mless :: [i,\ i] \Rightarrow o\ (\textbf{infixl}\ \langle <\#\rangle\ 50)$ **where**

$M <\# N \equiv \exists i.\ Ord(i) \wedge \langle M,\ N\rangle \in multirel(field(Memrel(i)),\ Memrel(i))$

definition
mle :: $[i,\ i] \Rightarrow o$ (**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*:
$f \in cons(a,\ b) \to B \Longrightarrow f = cons(<a,\ f\ `\ a>,\ funrestrict(f,\ b))$
⟨*proof*⟩

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

A useful simplification rule

lemma *multiset-fun-iff*:
$(f \in A \to nat-\{0\}) \longleftrightarrow f \in A \to nat \wedge (\forall a \in A.\ f`a \in nat \wedge 0 < f`a)$
⟨*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*]: $multiset(M) \Longrightarrow Finite(mset\text{-}of(M))$
⟨*proof*⟩

lemma *mset-of-0* [*iff*]: $mset\text{-}of(0) = 0$
⟨*proof*⟩

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

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

lemma *mset-of-union* [*iff*]: $mset\text{-}of(M +\# N) = mset\text{-}of(M) \cup mset\text{-}of(N)$
⟨*proof*⟩

lemma *mset-of-diff* [*simp*]: $mset\text{-}of(M)\subseteq A \Longrightarrow mset\text{-}of(M -\# N) \subseteq A$
⟨*proof*⟩

lemma *msingle-not-0* [*iff*]: $\{\#a\#\} \neq 0 \wedge 0 \neq \{\#a\#\}$
⟨*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*]: $[\![multiset(M);\ multiset(N)]\!] \Longrightarrow multiset(M +\# N)$
⟨*proof*⟩

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

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

lemma *mdiff-union-inverse2* [*simp*]: $multiset(M) \Longrightarrow M +\# \{\#a\#\} -\# \{\#a\#\} = M$
⟨*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*≠*0*; *multiset*(*M*)⟧ ⟹ ∃ *a* ∈ *mset-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*(λ*a*. $# *mcount*(*N*, *a*), *A* ∩ *mset-of*(*N*))
= *setsum*(λ*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*)
$\Longrightarrow$ $(\forall M.\ multiset(M) \longrightarrow$
$(\forall a \in mset\text{-}of(M).\ setsum(\lambda z.\ \$\#\ mcount(M(a{:=}M`a\ \#{-}\ 1),\ z),\ A) =$
$(if\ a \in A\ then\ setsum(\lambda z.\ \$\#\ mcount(M,\ z),\ A)\ \${-}\ \#1$
$else\ setsum(\lambda z.\ \$\#\ mcount(M,\ z),\ A))))$
⟨*proof*⟩

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

lemma *setsum-decr3*: $[\![Finite(A);\ multiset(M);\ a \in mset\text{-}of(M)]\!]$
$\Longrightarrow setsum(\lambda x.\ \$\#\ mcount(funrestrict(M,\ mset\text{-}of(M){-}\{a\}),\ x),\ A - \{a\})$
=
$(if\ a \in A\ then\ setsum(\lambda x.\ \$\#\ mcount(M,\ x),\ A)\ \${-}\ \$\#\ M`a$
$else\ setsum(\lambda x.\ \$\#\ mcount(M,\ x),\ A))$
⟨*proof*⟩

lemma *nat-le-1-cases*: $n \in nat \Longrightarrow n \leq 1 \longleftrightarrow (n=0 \mid n=1)$
⟨*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*: $\bigwedge M\ a.\ [\![multiset(M);\ a{\notin}mset\text{-}of(M);\ P(M)]\!] \Longrightarrow P(cons(\langle a,\ 1\rangle,\ M))$
and *prem2*: $\bigwedge M\ b.\ [\![multiset(M);\ b \in mset\text{-}of(M);\ P(M)]\!] \Longrightarrow P(M(b{:=}\ M`b\ \#{+}\ 1))$
shows
$[\![n \in nat;\ P(0)]\!]$
$\Longrightarrow (\forall M.\ multiset(M) \longrightarrow$
$(setsum(\lambda x.\ \$\#\ mcount(M,\ x),\ \{x \in mset\text{-}of(M).\ 0 < M`x\}) = \$\#\ n) \longrightarrow P(M))$

⟨*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 ∈ *mset-of*({#*x* ∈ *M*. *P*(*x*)#}) ⟷ *x* ∈ *mset-of*(*M*) ∧ *P*(*x*)
⟨*proof*⟩

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

lemma *multiset-partition*: *multiset*(*M*) ⟹ *M* = {# *x* ∈ *M*. *P*(*x*) #} +# {# *x* ∈ *M*. ¬ *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*: $[\![multiset(M);\ multiset(N)]\!]$
$\Longrightarrow\ (M +\# \{\#a\#\} = N +\# \{\#b\#\}) \longleftrightarrow\ (M = N \wedge a = b \mid$
$M = N -\# \{\#a\#\} +\# \{\#b\#\} \wedge N = M -\# \{\#b\#\} +\# \{\#a\#\})$
⟨*proof*⟩

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

lemma *munion-eq-conv-exist*:
$[\![M \in Mult(A);\ N \in Mult(A)]\!]$
$\Longrightarrow (M +\# \{\#a\#\} = N +\# \{\#b\#\}) \longleftrightarrow$
$(M{=}N \wedge a{=}b \mid (\exists K \in Mult(A).\ M{=} K +\# \{\#b\#\} \wedge N{=}K +\# \{\#a\#\}))$
⟨*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*:
$\langle N,\ M\rangle \in multirel1(A,\ r) \longleftrightarrow$
$(\exists a.\ a \in A \wedge$
$(\exists M0.\ M0 \in Mult(A) \wedge (\exists K.\ K \in Mult(A) \wedge$
$M{=}M0 +\# \{\#a\#\} \wedge N{=}M0 +\# K \wedge (\forall b \in mset\text{-}of(K).\ \langle b,a\rangle \in r))))$
⟨*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*]: $\langle M,0\rangle \notin multirel1(A,\ r)$
⟨*proof*⟩

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

lemma *lemma2*: $[\![\forall b \in A.\ \langle b,a\rangle \in r$
$\longrightarrow (\forall M \in acc(multirel1(A,\ r)).\ M\ +\#\ \{\#b\#\}\ :acc(multirel1(A,\ r)));$
$M \in acc(multirel1(A,\ r));\ a \in A]\!] \Longrightarrow M\ +\#\ \{\#a\#\} \in acc(multirel1(A,\ r))$
⟨*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*: $multiset(M) \Longrightarrow mset\text{-}of(M)\subseteq A \longrightarrow$
$wf[A](r) \longrightarrow M \in field(multirel1(A,\ r)) \longrightarrow M \in acc(multirel1(A,\ r))$
⟨*proof*⟩

lemma *all-accessible*: $[\![wf[A](r);\ M \in Mult(A);\ A \neq 0]\!] \Longrightarrow M \in acc(multirel1(A,\ r))$
⟨*proof*⟩

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

lemma *wf-multirel1*: $wf(r) \Longrightarrow wf(multirel1(field(r),\ r))$
⟨*proof*⟩

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

lemma *multirel-mono*:
$[\![A \subseteq B;\ r \subseteq s]\!] \Longrightarrow multirel(A,\ r) \subseteq multirel(B,s)$
⟨*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*: $[\![n \leq m;\ \ m \in nat;\ n \in nat;\ k \in nat]\!] \Longrightarrow m\ \#-\ n\ \#+$
$k = m\ \#+\ k\ \#-\ n$
⟨*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*)]:
$n \in nat \Longrightarrow$
$(\forall\, I\ J\ K.$
$I \in Mult(A) \wedge J \in Mult(A) \wedge K \in Mult(A) \wedge$
$(msize(J) = \$\#\ n \wedge J \neq 0 \wedge\ (\forall\, k \in mset\text{-}of(K).\ \exists\, j \in mset\text{-}of(J).\ \langle k,\ j\rangle \in r))$

$\longrightarrow <I +\# K, I +\# J> \in multirel(A, r))$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

⟨*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*]

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*]

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*: $f \in A->B \Longrightarrow rmap(f)$: $list(A)->list(B)$
⟨*proof*⟩

lemma *rmap-Nil*: $rmap(f)\,`Nil = Nil$
⟨*proof*⟩

lemma *rmap-Cons*: $[\![f \in A->B;\ x \in A;\ xs$: $list(A)]\!]$
$\Longrightarrow rmap(f)\ `\ Cons(x,xs) = Cons(f`x,\ rmap(f)\,`xs)$
⟨*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: $[\![p \in propn;\ q \in propn]\!] \Longrightarrow H \mid- p⇒q⇒p$
S: $[\![p \in propn;\ q \in propn;\ r \in propn]\!]$
$\Longrightarrow H \mid- (p⇒q⇒r) \Rightarrow (p⇒q) \Rightarrow p⇒r$
DN: $p \in propn \Longrightarrow H \mid- ((p⇒Fls) \Rightarrow Fls) \Rightarrow p$

MP: $[\![H \mid\!- p \Rightarrow q;\ \ H \mid\!- p;\ \ p \in \mathit{propn};\ \ q \in \mathit{propn}]\!] \Longrightarrow H \mid\!- q$
type-intros *propn.intros*

declare *propn.intros* [*simp*]

10.3 The semantics

10.3.1 Semantics of propositional logic.

consts
is-true-fun :: $[i,i] \Rightarrow i$
primrec
is-true-fun(*Fls*, *t*) = *0*
is-true-fun(*Var*(*v*), *t*) = (*if* $v \in t$ *then 1 else 0*)
is-true-fun($p \Rightarrow q$, *t*) = (*if is-true-fun*(*p*,*t*) = *1 then is-true-fun*(*q*,*t*) *else 1*)

definition
is-true :: $[i,i] \Rightarrow o$ **where**
is-true(*p*,*t*) $\equiv$ *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*) $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *is-true-Var* [*simp*]: *is-true*($\#v$,*t*) $\longleftrightarrow$ $v \in t$
⟨*proof*⟩

lemma *is-true-Imp* [*simp*]: *is-true*($p \Rightarrow q$,*t*) $\longleftrightarrow$ (*is-true*(*p*,*t*)$\longrightarrow$*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 :: $[i,i] \Rightarrow o$ (**infixl** ‹|=› *50*) **where**
$H \mid\!= p \equiv \forall t.\ (\forall q \in H.\ \mathit{is\text{-}true}(q,t)) \longrightarrow \mathit{is\text{-}true}(p,t)$

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

consts
hyps :: $[i,i] \Rightarrow i$
primrec
hyps(*Fls*, *t*) = *0*
hyps(*Var*(*v*), *t*) = (*if* $v \in t$ *then* $\{\#v\}$ *else* $\{\#v \Rightarrow \mathit{Fls}\}$)
hyps($p \Rightarrow q$, *t*) = *hyps*(*p*,*t*) $\cup$ *hyps*(*q*,*t*)

10.4 Proof theory of propositional logic

lemma *thms-mono*: $G \subseteq H \Longrightarrow \mathit{thms}(G) \subseteq \mathit{thms}(H)$
⟨*proof*⟩

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

inductive-cases *ImpE*: $p \Rightarrow q \in propn$

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*: $[\![G \subseteq H;\ G \mid\!- p]\!] \Longrightarrow H \mid\!- p$
— 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*: $[\![H \mid\!- p;\ cons(p,H) \mid\!- q]\!] \Longrightarrow H \mid\!- q$
⟨*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*:
$p \in propn \Longrightarrow hyps(p,t) \mid\!- (\mathit{if}\ \mathit{is\text{-}true}(p,t)\ \mathit{then}\ p\ \mathit{else}\ p \Rightarrow Fls)$
— 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*]:
$[\![p \in propn;\ \ 0 \models p]\!] \Longrightarrow \forall t.\ hyps(p,t) - hyps(p,t0) \vdash p$
⟨*proof*⟩

10.5.3 Completeness theorem

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

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

lemma *completeness*:
$H \in Fin(propn) \Longrightarrow p \in propn \Longrightarrow H \models p \Longrightarrow H \vdash p$
⟨*proof*⟩

theorem *thms-iff*: $H \in Fin(propn) \Longrightarrow H \vdash p \longleftrightarrow H \models p \wedge p \in propn$
⟨*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*: $listn(A)\text{``}\{n\} = \{l \in list(A).\ length(l)=n\}$
⟨*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, $\Rightarrow^1$ and (multi-step) parallel reductions, $\Rightarrow$.

consts *parcontract* :: i

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

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

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

Misc definitions.

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

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

12.2 Transitive closure preserves the Church-Rosser property

lemma *diamond-strip-lemmaD* [*rule-format*]:
⟦*diamond*(*r*); ⟨*x*,*y*⟩:*r*^+⟧ ⟹
∀ *y′*. <*x*,*y′*>:*r* ⟶ (∃ *z*. <*y′*,*z*>: *r*^+ ∧ ⟨*y*,*z*⟩: *r*)
⟨*proof*⟩

lemma *diamond-trancl*: *diamond*(*r*) ⟹ *diamond*(*r*^+)
⟨*proof*⟩

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

12.3 Results about Contraction

For type checking: replaces *a* →^1 *b* by *a*, *b* ∈ *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* ∈ *comb* ⟹ *I*·*p* → *p*
— Example only: not used
⟨*proof*⟩

lemma *comb-I*: *I* ∈ *comb*
⟨*proof*⟩

12.4 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases *K-contractE* [*elim*!]: *K* →^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*!]:
$K \cdot p \Rightarrow^1 r \Longrightarrow (\exists p'.\ r = K \cdot p' \wedge p \Rightarrow^1 p')$
⟨*proof*⟩

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

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

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

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

lemma *contract-imp-parcontract*: $p \to^1 q \Longrightarrow p \Rightarrow^1 q$
⟨*proof*⟩

lemma *reduce-imp-parreduce*: $p \to q \Longrightarrow p \Rightarrow q$
⟨*proof*⟩

lemma *parcontract-imp-reduce*: $p \Rightarrow^1 q \Longrightarrow p \to q$
⟨*proof*⟩

lemma *parreduce-imp-reduce*: $p \Rightarrow q \Longrightarrow p \to q$
⟨*proof*⟩

lemma *parreduce-iff-reduce*: $p \Rightarrow q \longleftrightarrow p \to q$
⟨*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-0*: $[\![x \in nat;\ \ l \in list(nat)]\!] \Longrightarrow PROJ(0)\ `\ (Cons(x,l)) = x$
⟨*proof*⟩

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

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

lemma *PREC-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* $\subseteq$ *list*(*nat*)−>*nat*
intros
$SC \in prim\text{-}rec$
$k \in nat \Longrightarrow CONSTANT(k) \in prim\text{-}rec$
$i \in nat \Longrightarrow PROJ(i) \in prim\text{-}rec$
$[\![g \in prim\text{-}rec;\ fs \in list(prim\text{-}rec)]\!] \Longrightarrow COMP(g,fs) \in prim\text{-}rec$
$[\![f \in prim\text{-}rec;\ g \in prim\text{-}rec]\!] \Longrightarrow PREC(f,g) \in prim\text{-}rec$
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*]: $c \in prim\text{-}rec \Longrightarrow c \in list(nat) -> nat$
⟨*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*]: $i \in nat \Longrightarrow ACK(i) \in prim\text{-}rec$
⟨*proof*⟩

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

13.3 Ackermann's function cases

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

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

lemma *ack-succ-succ*:

$[\![i{\in}nat;\ \ j{\in}nat]\!] \Longrightarrow ack(succ(i),\ succ(j)) = ack(i,\ ack(succ(i),\ j))$
— 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 $\le$
⟨*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 $\le$
⟨*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* ∈ *nat*; *i2* ∈ *nat*; *j* ∈ *nat*⟧
⟹ *ack*(*i1*, *ack*(*i2*,*j*)) < *ack*(*succ*(*succ*(*i1*#+*i2*)), *j*)
— PROPERTY A 10
⟨*proof*⟩

lemma *ack-add-bound*:
⟦*i1* ∈ *nat*; *i2* ∈ *nat*; *j* ∈ *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* ∈ *nat*; *k* ∈ *nat*⟧
⟹ *i*#+*j* < *ack*(*succ*(*succ*(*succ*(*succ*(*k*)))), *j*)
— PROPERTY A 12.
— Article uses existential quantifier but the ALF proof used *k* #+ #4.
— Quantified version must be nested ∃ *k*′. ∀ *i*,*j* ....
⟨*proof*⟩

13.4 Main result

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

lemma *SC-case*: *l* ∈ *list*(*nat*) ⟹ *SC* ‘ *l* < *ack*(*1*, *list-add*(*l*))
⟨*proof*⟩

lemma *lt-ack1*: ⟦*i* ∈ *nat*; *j* ∈ *nat*⟧ ⟹ *i* < *ack*(*i*,*j*)
— PROPERTY A 4’? Extra lemma needed for *CONSTANT* case, constant functions.
⟨*proof*⟩

lemma *CONSTANT-case*:
⟦*l* ∈ *list*(*nat*); *k* ∈ *nat*⟧ ⟹ *CONSTANT*(*k*) ‘ *l* < *ack*(*k*, *list-add*(*l*))
⟨*proof*⟩

lemma *PROJ-case* [*rule-format*]:
l ∈ *list*(*nat*) ⟹ ∀ *i* ∈ *nat*. *PROJ*(*i*) ‘ *l* < *ack*(*0*, *list-add*(*l*))
⟨*proof*⟩

COMP case.

lemma *COMP-map-lemma*:
fs ∈ *list*({*f* ∈ *prim-rec*. ∃ *kf* ∈ *nat*. ∀ *l* ∈ *list*(*nat*). *f*‘*l* < *ack*(*kf*, *list-add*(*l*))})
⟹ ∃ *k* ∈ *nat*. ∀ *l* ∈ *list*(*nat*).
list-add(*map*(λ*f*. *f* ‘ *l*, *fs*)) < *ack*(*k*, *list-add*(*l*))
⟨*proof*⟩

lemma *COMP-case*:
⟦*kg*∈*nat*;

$\forall\, l \in list(nat).\ g`l < ack(kg,\ list\text{-}add(l))$;
$fs \in list(\{f \in prim\text{-}rec\ .$
$\exists\, kf \in nat.\ \forall\, l \in list(nat).$
$f`l < ack(kf,\ list\text{-}add(l))\})[\![$
$\Longrightarrow \exists\, k \in nat.\ \forall\, l \in list(nat).\ COMP(g,fs)`l < ack(k,\ list\text{-}add(l))$
$\langle proof \rangle$

PREC case.

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

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

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

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

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.