

Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

March 13, 2025

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 <i>evnodd</i>	18
7.2	Dominoes	18
7.3	Tilings	18
7.4	The Operator <i>setsum</i>	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 multirell	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 as- sumptions	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 constructors, 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 $data-unfold$: $data(A, B) = (\{0\} + A) + (A \times B + A \times B \times data(A, B))$
 $\langle proof \rangle$

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

lemma $data-mono$: $\llbracket A \subseteq C; B \subseteq D \rrbracket \Longrightarrow data(A, B) \subseteq data(C, D)$
 $\langle proof \rangle$

lemma $data-univ$: $data(univ(A), univ(A)) \subseteq univ(A)$
 $\langle proof \rangle$

lemma $data-subset-univ$:
 $\llbracket A \subseteq univ(C); B \subseteq univ(C) \rrbracket \Longrightarrow data(A, B) \subseteq univ(C)$
 $\langle proof \rangle$

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 $Br\text{-}neq\text{-}left: l \in bt(A) \Longrightarrow Br(x, l, r) \neq l$

$\langle proof \rangle$

lemma $Br\text{-}iff: Br(a, l, r) = Br(a', l', r') \longleftrightarrow a = a' \wedge l = l' \wedge r = r'$

— Proving a freeness theorem.

$\langle proof \rangle$

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 $bt\text{-}mono: A \subseteq B \Longrightarrow bt(A) \subseteq bt(B)$

$\langle proof \rangle$

lemma $bt\text{-}univ: bt(univ(A)) \subseteq univ(A)$

$\langle proof \rangle$

lemma $bt\text{-}subset\text{-}univ: A \subseteq univ(B) \Longrightarrow bt(A) \subseteq univ(B)$

$\langle proof \rangle$

lemma $bt\text{-}rec\text{-}type:$

$\llbracket t \in bt(A);$

$c \in C(Lf);$

$\bigwedge x\ y\ z\ r\ s. \llbracket x \in A; y \in bt(A); z \in bt(A); r \in C(y); s \in C(z) \rrbracket \Longrightarrow$

$h(x, y, z, r, s) \in C(Br(x, y, z))$

$\rrbracket \Longrightarrow bt\text{-}rec(c, h, t) \in C(t)$

— Type checking for recursor – example only; not really needed.

$\langle proof \rangle$

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 $n\text{-}nodes\text{-}type$ $[simp]: t \in bt(A) \Longrightarrow n\text{-}nodes(t) \in nat$

$\langle proof \rangle$

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

lemma $n\text{-nodes}\text{-aux}\text{-eq}$:
 $t \in \text{bt}(A) \implies k \in \text{nat} \implies n\text{-nodes}\text{-aux}(t) + k = n\text{-nodes}(t) \# + k$
 $\langle \text{proof} \rangle$

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

lemma $t \in \text{bt}(A) \implies n\text{-nodes}\text{-tail}(t) = n\text{-nodes}(t)$
 $\langle \text{proof} \rangle$

2.3 Number of leaves

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

lemma $n\text{-leaves}\text{-type}$ $[simp]$: $t \in \text{bt}(A) \implies n\text{-leaves}(t) \in \text{nat}$
 $\langle \text{proof} \rangle$

2.4 Reflecting trees

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

lemma $\text{bt}\text{-reflect}\text{-type}$ $[simp]$: $t \in \text{bt}(A) \implies \text{bt}\text{-reflect}(t) \in \text{bt}(A)$
 $\langle \text{proof} \rangle$

Theorems about $n\text{-leaves}$.

lemma $n\text{-leaves}\text{-reflect}$: $t \in \text{bt}(A) \implies n\text{-leaves}(\text{bt}\text{-reflect}(t)) = n\text{-leaves}(t)$
 $\langle \text{proof} \rangle$

lemma $n\text{-leaves}\text{-nodes}$: $t \in \text{bt}(A) \implies n\text{-leaves}(t) = \text{succ}(n\text{-nodes}(t))$
 $\langle \text{proof} \rangle$

Theorems about $\text{bt}\text{-reflect}$.

lemma $\text{bt}\text{-reflect}\text{-bt}\text{-reflect}\text{-ident}$: $t \in \text{bt}(A) \implies \text{bt}\text{-reflect}(\text{bt}\text{-reflect}(t)) = t$
 $\langle \text{proof} \rangle$

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* ($a \in A, l \in \text{list}(\text{term}(A))$)

monos *list-mono*

type-elim *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

$Vrec(t, \lambda t \ g. \text{term-case}(\lambda x \ zs. \ d(x, zs, \text{map}(\lambda z. \ g'z, zs)), t))$

definition

term-map :: $[i \Rightarrow i, i] \Rightarrow i$ **where**

term-map(f, t) $\equiv \text{term-rec}(t, \lambda x \ zs \ rs. \ \text{Apply}(f(x), rs))$

definition

term-size :: $i \Rightarrow i$ **where**

term-size(t) $\equiv \text{term-rec}(t, \lambda x \ zs \ rs. \ \text{succ}(\text{list-add}(rs)))$

definition

reflect :: $i \Rightarrow i$ **where**

reflect(t) $\equiv \text{term-rec}(t, \lambda x \ zs \ rs. \ \text{Apply}(x, \text{rev}(rs)))$

definition

preorder :: $i \Rightarrow i$ **where**

preorder(t) $\equiv \text{term-rec}(t, \lambda x \ zs \ rs. \ \text{Cons}(x, \text{flat}(rs)))$

definition

postorder :: $i \Rightarrow i$ **where**

postorder(t) $\equiv \text{term-rec}(t, \lambda x \ zs \ rs. \ \text{flat}(rs) \ @ \ [x])$

lemma *term-unfold*: $\text{term}(A) = A * \text{list}(\text{term}(A))$

<proof>

lemma *term-induct2*:

$\llbracket t \in \text{term}(A);$

$\bigwedge x. \llbracket x \in A \rrbracket \implies P(\text{Apply}(x, \text{Nil});$

$\bigwedge x \ z \ zs. \llbracket x \in A; \ z \in \text{term}(A); \ zs: \text{list}(\text{term}(A)); \ P(\text{Apply}(x, zs))$

$\llbracket \implies P(\text{Apply}(x, \text{Cons}(z, zs)))$
 $\llbracket \implies P(t)$
 — Induction on $\text{term}(A)$ followed by induction on list .
 $\langle \text{proof} \rangle$

lemma *term-induct-eqn* [consumes 1, case-names *Apply*]:

$\llbracket t \in \text{term}(A);$
 $\quad \bigwedge x \text{ } zs. \llbracket x \in A; \text{ } zs: \text{list}(\text{term}(A)); \text{ } \text{map}(f, zs) = \text{map}(g, zs) \rrbracket \implies$
 $\quad \quad f(\text{Apply}(x, zs)) = g(\text{Apply}(x, zs))$
 $\llbracket \implies f(t) = g(t)$
 — Induction on $\text{term}(A)$ to prove an equation.
 $\langle \text{proof} \rangle$

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

lemma *term-mono*: $A \subseteq B \implies \text{term}(A) \subseteq \text{term}(B)$
 $\langle \text{proof} \rangle$

lemma *term-univ*: $\text{term}(\text{univ}(A)) \subseteq \text{univ}(A)$
 — Easily provable by induction also
 $\langle \text{proof} \rangle$

lemma *term-subset-univ*: $A \subseteq \text{univ}(B) \implies \text{term}(A) \subseteq \text{univ}(B)$
 $\langle \text{proof} \rangle$

lemma *term-into-univ*: $\llbracket t \in \text{term}(A); \text{ } A \subseteq \text{univ}(B) \rrbracket \implies t \in \text{univ}(B)$
 $\langle \text{proof} \rangle$

term-rec – by *Vset* recursion.

lemma *map-lemma*: $\llbracket l \in \text{list}(A); \text{ } \text{Ord}(i); \text{ } \text{rank}(l) < i \rrbracket$
 $\implies \text{map}(\lambda z. (\lambda x \in \text{Vset}(i). h(x)) \text{ } 'z, l) = \text{map}(h, l)$
 — *map* works correctly on the underlying list of terms.
 $\langle \text{proof} \rangle$

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

lemma *term-rec-type*:

assumes $t: t \in \text{term}(A)$
and $a: \bigwedge x \text{ } zs \text{ } r. \llbracket x \in A; \text{ } zs: \text{list}(\text{term}(A));$
 $\quad \quad r \in \text{list}(\bigcup t \in \text{term}(A). C(t)) \rrbracket$
 $\implies d(x, zs, r): C(\text{Apply}(x, zs))$
shows $\text{term-rec}(t, d) \in C(t)$
 — Slightly odd typing condition on r in the second premise!
 $\langle \text{proof} \rangle$

lemma *def-term-rec*:

$\llbracket \wedge t. j(t) \equiv \text{term-rec}(t, d); \text{ ts: list}(A) \rrbracket \implies$
 $j(\text{Apply}(a, \text{ts})) = d(a, \text{ts}, \text{map}(\lambda Z. j(Z), \text{ts}))$
 $\langle \text{proof} \rangle$

lemma *term-rec-simple-type* [TC]:

$\llbracket t \in \text{term}(A);$
 $\wedge x \text{ zs } r. \llbracket x \in A; \text{ zs: list}(\text{term}(A)); \text{ r} \in \text{list}(C) \rrbracket$
 $\implies d(x, \text{zs}, r): C$
 $\rrbracket \implies \text{term-rec}(t, d) \in C$
 $\langle \text{proof} \rangle$

term-map.

lemma *term-map* [simp]:

$\text{ts} \in \text{list}(A) \implies$
 $\text{term-map}(f, \text{Apply}(a, \text{ts})) = \text{Apply}(f(a), \text{map}(\text{term-map}(f), \text{ts}))$
 $\langle \text{proof} \rangle$

lemma *term-map-type* [TC]:

$\llbracket t \in \text{term}(A); \wedge x. x \in A \implies f(x): B \rrbracket \implies \text{term-map}(f, t) \in \text{term}(B)$
 $\langle \text{proof} \rangle$

lemma *term-map-type2* [TC]:

$t \in \text{term}(A) \implies \text{term-map}(f, t) \in \text{term}(\{f(u). u \in A\})$
 $\langle \text{proof} \rangle$

term-size.

lemma *term-size* [simp]:

$\text{ts} \in \text{list}(A) \implies \text{term-size}(\text{Apply}(a, \text{ts})) = \text{succ}(\text{list-add}(\text{map}(\text{term-size}, \text{ts})))$
 $\langle \text{proof} \rangle$

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

reflect.

lemma *reflect* [simp]:

$\text{ts} \in \text{list}(A) \implies \text{reflect}(\text{Apply}(a, \text{ts})) = \text{Apply}(a, \text{rev}(\text{map}(\text{reflect}, \text{ts})))$
 $\langle \text{proof} \rangle$

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

preorder.

lemma *preorder* [simp]:

$\text{ts} \in \text{list}(A) \implies \text{preorder}(\text{Apply}(a, \text{ts})) = \text{Cons}(a, \text{flat}(\text{map}(\text{preorder}, \text{ts})))$
 $\langle \text{proof} \rangle$

lemma *preorder-type* [TC]: $t \in \text{term}(A) \implies \text{preorder}(t) \in \text{list}(A)$
 $\langle \text{proof} \rangle$

postorder.

lemma *postorder* [simp]:
 $ts \in \text{list}(A) \implies \text{postorder}(\text{Apply}(a, ts)) = \text{flat}(\text{map}(\text{postorder}, ts)) @ [a]$
 $\langle \text{proof} \rangle$

lemma *postorder-type* [TC]: $t \in \text{term}(A) \implies \text{postorder}(t) \in \text{list}(A)$
 $\langle \text{proof} \rangle$

Theorems about *term-map*.

declare *map-compose* [simp]

lemma *term-map-ident*: $t \in \text{term}(A) \implies \text{term-map}(\lambda u. u, t) = t$
 $\langle \text{proof} \rangle$

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

lemma *term-map-reflect*:
 $t \in \text{term}(A) \implies \text{term-map}(f, \text{reflect}(t)) = \text{reflect}(\text{term-map}(f, t))$
 $\langle \text{proof} \rangle$

Theorems about *term-size*.

lemma *term-size-term-map*: $t \in \text{term}(A) \implies \text{term-size}(\text{term-map}(f, t)) = \text{term-size}(t)$
 $\langle \text{proof} \rangle$

lemma *term-size-reflect*: $t \in \text{term}(A) \implies \text{term-size}(\text{reflect}(t)) = \text{term-size}(t)$
 $\langle \text{proof} \rangle$

lemma *term-size-length*: $t \in \text{term}(A) \implies \text{term-size}(t) = \text{length}(\text{preorder}(t))$
 $\langle \text{proof} \rangle$

Theorems about *reflect*.

lemma *reflect-reflect-ident*: $t \in \text{term}(A) \implies \text{reflect}(\text{reflect}(t)) = t$
 $\langle \text{proof} \rangle$

Theorems about *preorder*.

lemma *preorder-term-map*:
 $t \in \text{term}(A) \implies \text{preorder}(\text{term-map}(f, t)) = \text{map}(f, \text{preorder}(t))$
 $\langle \text{proof} \rangle$

lemma *preorder-reflect-eq-rev-postorder*:
 $t \in \text{term}(A) \implies \text{preorder}(\text{reflect}(t)) = \text{rev}(\text{postorder}(t))$

$\langle proof \rangle$

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* ($a \in A, h \in (\bigcup n \in \text{nat}. n \rightarrow \text{ntree}(A))$)
monos *UN-mono* [*OF subset-refl Pi-mono*] — MUST have this form
type-intros *nat-fun-univ* [*THEN subsetD*]
type-elim *UN-E*

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

datatype *maptree2*(*A*, *B*) = *Sons2* ($a \in A, h \in B \rightarrow \text{maptree2}(A, B)$)
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*) \equiv
 $\text{Vrecursor}(\lambda \text{pr}. \text{ntree-case}(\lambda x \text{ h}. \text{b}(x, \text{h}), \lambda i \in \text{domain}(\text{h}). \text{pr}'(\text{h}'i)))$

definition

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

ntree

lemma *ntree-unfold*: $\text{ntree}(A) = A \times (\bigcup n \in \text{nat}. n \rightarrow \text{ntree}(A))$
 $\langle proof \rangle$

lemma *ntree-induct* [*consumes 1, case-names Branch, induct set: ntree*]:

assumes *t*: $t \in \text{ntree}(A)$

and step: $\bigwedge x \text{ n } h. \llbracket x \in A; n \in \text{nat}; h \in n \rightarrow \text{ntree}(A); \forall i \in n. P(\text{h}'i) \rrbracket$

$\implies P(\text{Branch}(x, h))$

shows $P(t)$

— A nicer induction rule than the standard one.

$\langle proof \rangle$

lemma *ntree-induct-eqn* [*consumes 1*]:
assumes $t: t \in \text{ntree}(A)$
and $f: f \in \text{ntree}(A) \rightarrow B$
and $g: g \in \text{ntree}(A) \rightarrow B$
and *step*: $\bigwedge x n h. \llbracket x \in A; n \in \text{nat}; h \in n \rightarrow \text{ntree}(A); f \circ h = g \circ h \rrbracket \implies$
 $f \text{ ` Branch}(x,h) = g \text{ ` Branch}(x,h)$
shows $f^t = g^t$
— Induction on $\text{ntree}(A)$ to prove an equation
 $\langle \text{proof} \rangle$

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

lemma *ntree-mono*: $A \subseteq B \implies \text{ntree}(A) \subseteq \text{ntree}(B)$
 $\langle \text{proof} \rangle$

lemma *ntree-univ*: $\text{ntree}(\text{univ}(A)) \subseteq \text{univ}(A)$
— Easily provable by induction also
 $\langle \text{proof} \rangle$

lemma *ntree-subset-univ*: $A \subseteq \text{univ}(B) \implies \text{ntree}(A) \subseteq \text{univ}(B)$
 $\langle \text{proof} \rangle$

ntree recursion.

lemma *ntree-rec-Branch*:
 $\text{function}(h) \implies$
 $\text{ntree-rec}(b, \text{Branch}(x,h)) = b(x, h, \lambda i \in \text{domain}(h). \text{ntree-rec}(b, h^i))$
 $\langle \text{proof} \rangle$

lemma *ntree-copy-Branch* [*simp*]:
 $\text{function}(h) \implies$
 $\text{ntree-copy}(\text{Branch}(x, h)) = \text{Branch}(x, \lambda i \in \text{domain}(h). \text{ntree-copy}(h^i))$
 $\langle \text{proof} \rangle$

lemma *ntree-copy-is-ident*: $z \in \text{ntree}(A) \implies \text{ntree-copy}(z) = z$
 $\langle \text{proof} \rangle$

maptree

lemma *maptree-unfold*: $\text{maptree}(A) = A \times (\text{maptree}(A) \multimap \text{maptree}(A))$
 $\langle \text{proof} \rangle$

lemma *maptree-induct* [*consumes 1, induct set: maptree*]:
assumes $t: t \in \text{maptree}(A)$
and *step*: $\bigwedge x n h. \llbracket x \in A; h \in \text{maptree}(A) \multimap \text{maptree}(A);$
 $\forall y \in \text{field}(h). P(y)$
 $\rrbracket \implies P(\text{Sons}(x,h))$
shows $P(t)$
— A nicer induction rule than the standard one.

<proof>

maptree2

lemma *maptree2-unfold*: $\text{maptree2}(A, B) = A \times (B \multimap \text{maptree2}(A, B))$
<proof>

lemma *maptree2-induct* [*consumes 1, induct set: maptree2*]:

assumes $t \in \text{maptree2}(A, B)$

and step: $\bigwedge x \ n \ h. \llbracket x \in A; \ h \in B \multimap \text{maptree2}(A, B) \rrbracket \Rightarrow \forall y \in \text{range}(h). P(y)$

|| $\Rightarrow P(\text{Sons2}(x, h))$

shows $P(t)$

<proof>

end

5 Trees and forests, a mutually recursive definition

theory *Tree-Forest* imports *ZF* begin

5.1 Datatype definition

consts

tree :: $i \Rightarrow i$

forest :: $i \Rightarrow i$

tree-forest :: $i \Rightarrow i$

datatype *tree*(A) = *Tcons* ($a \in A, f \in \text{forest}(A)$)

and *forest*(A) = *Fnil* | *Fcons* ($t \in \text{tree}(A), f \in \text{forest}(A)$)

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

<proof>

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

<proof>

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

lemma *tree-subset-TF*: $tree(A) \subseteq tree-forest(A)$
 $\langle proof \rangle$

lemma *treeI* [TC]: $x \in tree(A) \implies x \in tree-forest(A)$
 $\langle proof \rangle$

lemma *forest-subset-TF*: $forest(A) \subseteq tree-forest(A)$
 $\langle proof \rangle$

lemma *treeI'* [TC]: $x \in forest(A) \implies x \in tree-forest(A)$
 $\langle proof \rangle$

lemma *TF-equals-Un*: $tree(A) \cup forest(A) = tree-forest(A)$
 $\langle proof \rangle$

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

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

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

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

Type checking for recursor: Not needed; possibly interesting?

lemma *TF-rec-type*:
 $\llbracket z \in tree-forest(A);$
 $\bigwedge x f r. \llbracket x \in A; f \in forest(A); r \in C(f)$
 $\rrbracket \implies b(x,f,r) \in C(Tcons(x,f));$
 $c \in C(Fnil);$
 $\bigwedge t f r1 r2. \llbracket t \in tree(A); f \in forest(A); r1 \in C(t); r2 \in C(f)$
 $\rrbracket \implies d(t,f,r1,r2) \in C(Fcons(t,f))$
 $\rrbracket \implies tree-forest-rec(b,c,d,z) \in C(z)$
 $\langle proof \rangle$

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

$\mathbb{I} \implies (\forall t \in \text{tree}(A). \text{tree-forest-rec}(b,c,d,t) \in C(t)) \wedge$
 $(\forall f \in \text{forest}(A). \text{tree-forest-rec}(b,c,d,f) \in D(f))$
 — Mutually recursive version.
 $\langle \text{proof} \rangle$

5.2 Operations

consts

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

primrec

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

primrec

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

primrec

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

primrec

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

primrec

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

primrec

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

list-of-TF and of-list .

lemma $\text{list-of-TF-type } [TC]:$

$z \in \text{tree-forest}(A) \implies \text{list-of-TF}(z) \in \text{list}(\text{tree}(A))$

$\langle \text{proof} \rangle$

lemma *of-list-type* [TC]: $l \in \text{list}(\text{tree}(A)) \implies \text{of-list}(l) \in \text{forest}(A)$
 $\langle \text{proof} \rangle$

map.

lemma
assumes $\bigwedge x. x \in A \implies h(x): B$
shows *map-tree-type*: $t \in \text{tree}(A) \implies \text{map}(h, t) \in \text{tree}(B)$
and *map-forest-type*: $f \in \text{forest}(A) \implies \text{map}(h, f) \in \text{forest}(B)$
 $\langle \text{proof} \rangle$

size.

lemma *size-type* [TC]: $z \in \text{tree-forest}(A) \implies \text{size}(z) \in \text{nat}$
 $\langle \text{proof} \rangle$

preorder.

lemma *preorder-type* [TC]: $z \in \text{tree-forest}(A) \implies \text{preorder}(z) \in \text{list}(A)$
 $\langle \text{proof} \rangle$

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

lemma *forest-induct* [consumes 1, case-names Fnil Fcons]:
 $\llbracket f \in \text{forest}(A);$
 $R(\text{Fnil});$
 $\bigwedge t f. \llbracket t \in \text{tree}(A); f \in \text{forest}(A); R(f) \rrbracket \implies R(\text{Fcons}(t, f))$
 $\rrbracket \implies R(f)$
— Essentially the same as list induction.
 $\langle \text{proof} \rangle$

lemma *forest-iso*: $f \in \text{forest}(A) \implies \text{of-list}(\text{list-of-TF}(f)) = f$
 $\langle \text{proof} \rangle$

lemma *tree-list-iso*: $ts: \text{list}(\text{tree}(A)) \implies \text{list-of-TF}(\text{of-list}(ts)) = ts$
 $\langle \text{proof} \rangle$

Theorems about *map*.

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

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

Theorems about *size*.

lemma *size-map*: $z \in \text{tree-forest}(A) \implies \text{size}(\text{map}(h, z)) = \text{size}(z)$
 $\langle \text{proof} \rangle$

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

Theorems about *preorder*.

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

end

6 Infinite branching datatype definitions

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

6.1 The Brouwer ordinals

consts

brouwer :: *i*

datatype $\subseteq V_{\text{from}}(0, \text{csucc}(\text{nat}))$
 $\text{brouwer} = \text{Zero} \mid \text{Suc } (b \in \text{brouwer}) \mid \text{Lim } (h \in \text{nat} \rightarrow \text{brouwer})$
monos *Pi-mono*
type-intros *inf-datatype-intros*

lemma *brouwer-unfold*: $\text{brouwer} = \{0\} + \text{brouwer} + (\text{nat} \rightarrow \text{brouwer})$
 ⟨proof⟩

lemma *brouwer-induct2* [*consumes 1, case-names Zero Suc Lim*]:

assumes $b: b \in \text{brouwer}$

and cases:

$P(\text{Zero})$

$\bigwedge b. \llbracket b \in \text{brouwer}; P(b) \rrbracket \implies P(\text{Suc}(b))$

$\bigwedge h. \llbracket h \in \text{nat} \rightarrow \text{brouwer}; \forall i \in \text{nat}. P(h'i) \rrbracket \implies P(\text{Lim}(h))$

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 $\subseteq V_{\text{from}}(A \cup (\bigcup x \in A. B(x)), \text{csucc}(\text{nat} \cup |\bigcup x \in A. B(x)|))$

— The union with *nat* ensures that the cardinal is infinite.

$\text{Well}(A, B) = \text{Sup } (a \in A, f \in B(a) \rightarrow \text{Well}(A, B))$

monos *Pi-mono*

type-intros *le-trans* [*OF UN-upper-cardinal le-nat-Un-cardinal*] *inf-datatype-intros*

lemma *Well-unfold*: $Well(A, B) = (\sum x \in A. B(x) \rightarrow Well(A, B))$
 ⟨proof⟩

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

lemma *Well-bool-unfold*: $Well(bool, \lambda x. x) = 1 + (1 \rightarrow Well(bool, \lambda x. x))$
 — 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: $\llbracket i \in nat; j \in nat \rrbracket \implies \{\langle i, j \rangle, \langle i, succ(j) \rangle\} \in domino$
vertl: $\llbracket i \in nat; j \in nat \rrbracket \implies \{\langle i, j \rangle, \langle succ(i), j \rangle\} \in domino$
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: $\llbracket a \in A; t \in tiling(A); a \cap t = 0 \rrbracket \implies a \cup t \in tiling(A)$
type-intros *empty-subsetI Union-upper Un-least PowI*
type-elim *PowD [elim-format]*

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

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

7.1 Basic properties of evnodd

lemma *evnodd-iff*: $\langle i,j \rangle: \text{evnodd}(A,b) \longleftrightarrow \langle i,j \rangle: A \wedge (i \# + j) \bmod 2 = b$
 $\langle \text{proof} \rangle$

lemma *evnodd-subset*: $\text{evnodd}(A, b) \subseteq A$
 $\langle \text{proof} \rangle$

lemma *Finite-evnodd*: $\text{Finite}(X) \implies \text{Finite}(\text{evnodd}(X,b))$
 $\langle \text{proof} \rangle$

lemma *evnodd-Un*: $\text{evnodd}(A \cup B, b) = \text{evnodd}(A,b) \cup \text{evnodd}(B,b)$
 $\langle \text{proof} \rangle$

lemma *evnodd-Diff*: $\text{evnodd}(A - B, b) = \text{evnodd}(A,b) - \text{evnodd}(B,b)$
 $\langle \text{proof} \rangle$

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

lemma *evnodd-0* [*simp*]: $\text{evnodd}(0, b) = 0$
 $\langle \text{proof} \rangle$

7.2 Dominoes

lemma *domino-Finite*: $d \in \text{domino} \implies \text{Finite}(d)$
 $\langle \text{proof} \rangle$

lemma *domino-singleton*:
 $\llbracket d \in \text{domino}; b < 2 \rrbracket \implies \exists i' j'. \text{evnodd}(d,b) = \{\langle i',j' \rangle\}$
 $\langle \text{proof} \rangle$

7.3 Tilings

The union of two disjoint tilings is a tiling

lemma *tiling-UnI*:
 $t \in \text{tiling}(A) \implies u \in \text{tiling}(A) \implies t \cap u = 0 \implies t \cup u \in \text{tiling}(A)$
 $\langle \text{proof} \rangle$

lemma *tiling-domino-Finite*: $t \in \text{tiling}(\text{domino}) \implies \text{Finite}(t)$
 $\langle \text{proof} \rangle$

lemma *tiling-domino-0-1*: $t \in \text{tiling}(\text{domino}) \implies |\text{evnodd}(t,0)| = |\text{evnodd}(t,1)|$
 $\langle \text{proof} \rangle$

lemma *dominoes-tile-row*:

$\llbracket i \in \text{nat}; n \in \text{nat} \rrbracket \implies \{i\} * (n \# + n) \in \text{tiling}(\text{domino})$
 $\langle \text{proof} \rangle$

lemma *dominoes-tile-matrix*:

$\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies m * (n \# + n) \in \text{tiling}(\text{domino})$
 $\langle \text{proof} \rangle$

lemma *eq-lt-E*: $\llbracket x=y; x<y \rrbracket \implies P$

$\langle \text{proof} \rangle$

theorem *mutl-not-tiling*: $\llbracket m \in \text{nat}; n \in \text{nat};$

$t = (\text{succ}(m) \# + \text{succ}(m)) * (\text{succ}(n) \# + \text{succ}(n));$

$t' = t - \{\langle 0, 0 \rangle\} - \{\langle \text{succ}(m \# + m), \text{succ}(n \# + n) \rangle\}$

$\implies t' \notin \text{tiling}(\text{domino})$

$\langle \text{proof} \rangle$

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*) $\subseteq \text{Fin}(A) * B$

intros

emptyI: $e \in B \implies \langle 0, e \rangle \in \text{fold-set}(A, B, f, e)$

consI: $\llbracket x \in A; x \notin C; \langle C, y \rangle \in \text{fold-set}(A, B, f, e); f(x, y) : B \rrbracket$

$\implies \langle \text{cons}(x, C), f(x, y) \rangle \in \text{fold-set}(A, B, f, e)$

type-intros *Fin.intros*

definition

fold :: $[i, [i, i] \Rightarrow i, i, i] \Rightarrow i$ ($\langle \text{fold}[_] \rangle'(-, -, -) \rangle$) **where**

$\text{fold}[B](f, e, A) \equiv \text{THE } x. \langle A, x \rangle \in \text{fold-set}(A, B, f, e)$

definition

setsum :: $[i \Rightarrow i, i] \Rightarrow i$ **where**

setsum(*g*, *C*) \equiv *if* *Finite*(*C*) *then*

$\text{fold}[\text{int}](\lambda x y. g(x) \$ + y, \#0, C)$ *else* $\#0$

inductive-cases *empty-fold-setE*: $\langle 0, x \rangle \in \text{fold-set}(A, B, f, e)$

inductive-cases *cons-fold-setE*: $\langle \text{cons}(x, C), y \rangle \in \text{fold-set}(A, B, f, e)$

lemma *cons-lemma1*: $\llbracket x \notin C; x \notin B \rrbracket \implies \text{cons}(x, B) = \text{cons}(x, C) \longleftrightarrow B = C$

$\langle \text{proof} \rangle$

lemma *cons-lemma2*: $\llbracket \text{cons}(x, B) = \text{cons}(y, C); x \neq y; x \notin B; y \notin C \rrbracket$
 $\implies B - \{y\} = C - \{x\} \wedge x \in C \wedge y \in B$
 $\langle \text{proof} \rangle$

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

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

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

lemma *Diff1-fold-set*:
 $\llbracket \langle C - \{x\}, y \rangle \in \text{fold-set}(A, B, f, e); x \in C; x \in A; f(x, y) \in B \rrbracket$
 $\implies \langle C, f(x, y) \rangle \in \text{fold-set}(A, B, f, e)$
 $\langle \text{proof} \rangle$

locale *fold-typing* =
fixes *A and B and e and f*
assumes *f*type [intro,simp]: $\llbracket x \in A; y \in B \rrbracket \implies f(x, y) \in B$
and *e*type [intro,simp]: $e \in B$
and *f*comm: $\llbracket x \in A; y \in A; z \in B \rrbracket \implies f(x, f(y, z)) = f(y, f(x, z))$

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

lemma *Diff-sing-imp*:
 $\llbracket C - \{b\} = D - \{a\}; a \neq b; b \in C \rrbracket \implies C = \text{cons}(b, D) - \{a\}$
 $\langle \text{proof} \rangle$

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

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

$$\begin{aligned} & \llbracket \langle C, x \rangle \in \text{fold-set}(A, B, f, e); \\ & \quad \langle C, y \rangle \in \text{fold-set}(A, B, f, e) \rrbracket \implies y=x \\ \langle \text{proof} \rangle \end{aligned}$$

lemma (in *fold-typing*) *fold-equality*:

$$\langle C, y \rangle \in \text{fold-set}(A, B, f, e) \implies \text{fold}[B](f, e, C) = y$$
 $\langle \text{proof} \rangle$

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

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

lemma (in *fold-typing*) *fold-set-imp-cons*:

$$\begin{aligned} & \llbracket \langle C, y \rangle \in \text{fold-set}(C, B, f, e); C \in \text{Fin}(A); c \in A; c \notin C \rrbracket \\ & \implies \langle \text{cons}(c, C), f(c, y) \rangle \in \text{fold-set}(\text{cons}(c, C), B, f, e) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma (in *fold-typing*) *fold-cons-lemma [rule-format]*:

$$\begin{aligned} & \llbracket C \in \text{Fin}(A); c \in A; c \notin C \rrbracket \\ & \implies \langle \text{cons}(c, C), v \rangle \in \text{fold-set}(\text{cons}(c, C), B, f, e) \longleftrightarrow \\ & \quad (\exists y. \langle C, y \rangle \in \text{fold-set}(C, B, f, e) \wedge v = f(c, y)) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma (in *fold-typing*) *fold-cons*:

$$\begin{aligned} & \llbracket C \in \text{Fin}(A); c \in A; c \notin C \rrbracket \\ & \implies \text{fold}[B](f, e, \text{cons}(c, C)) = f(c, \text{fold}[B](f, e, C)) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma (in *fold-typing*) *fold-type [simp, TC]*:

$$C \in \text{Fin}(A) \implies \text{fold}[B](f, e, C) : B$$
 $\langle \text{proof} \rangle$

lemma (in *fold-typing*) *fold-commute [rule-format]*:

$$\begin{aligned} & \llbracket C \in \text{Fin}(A); c \in A \rrbracket \\ & \implies (\forall y \in B. f(c, \text{fold}[B](f, y, C)) = \text{fold}[B](f, f(c, y), C)) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma (in *fold-typing*) *fold-nest-Un-Int*:

$$\begin{aligned} & \llbracket C \in \text{Fin}(A); D \in \text{Fin}(A) \rrbracket \\ & \implies \text{fold}[B](f, \text{fold}[B](f, e, D), C) = \\ & \quad \text{fold}[B](f, \text{fold}[B](f, e, (C \cap D)), C \cup D) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma (in *fold-typing*) *fold-nest-Un-disjoint*:

$$\begin{aligned} & \llbracket C \in \text{Fin}(A); D \in \text{Fin}(A); C \cap D = \emptyset \rrbracket \\ & \implies \text{fold}[B](f, e, C \cup D) = \text{fold}[B](f, \text{fold}[B](f, e, D), C) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *Finite-cons-lemma*: $Finite(C) \implies C \in Fin(cons(c, C))$
 $\langle proof \rangle$

7.4 The Operator *setsum*

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

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

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

lemma *setsum-Un-Int*:
 $\llbracket Finite(C); Finite(D) \rrbracket$
 $\implies setsum(g, C \cup D) \$+ setsum(g, C \cap D)$
 $= setsum(g, C) \$+ setsum(g, D)$
 $\langle proof \rangle$

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

lemma *setsum-Un-disjoint*:
 $\llbracket Finite(C); Finite(D); C \cap D = 0 \rrbracket$
 $\implies setsum(g, C \cup D) = setsum(g, C) \$+ setsum(g, D)$
 $\langle proof \rangle$

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

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

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

lemma *fold-set-cong*:

$$\begin{aligned} & \llbracket A=A'; B=B'; e=e'; (\forall x \in A'. \forall y \in B'. f(x,y) = f'(x,y)) \rrbracket \\ & \implies \text{fold-set}(A,B,f,e) = \text{fold-set}(A',B',f',e') \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *fold-cong*:

$$\begin{aligned} & \llbracket B=B'; A=A'; e=e'; \\ & \quad \bigwedge x y. \llbracket x \in A'; y \in B' \rrbracket \implies f(x,y) = f'(x,y) \rrbracket \implies \\ & \quad \text{fold}[B](f,e,A) = \text{fold}[B'](f', e', A') \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setsum-cong*:

$$\begin{aligned} & \llbracket A=B; \bigwedge x. x \in B \implies f(x) = g(x) \rrbracket \implies \\ & \quad \text{setsum}(f, A) = \text{setsum}(g, B) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setsum-Un*:

$$\begin{aligned} & \llbracket \text{Finite}(A); \text{Finite}(B) \rrbracket \\ & \implies \text{setsum}(f, A \cup B) = \\ & \quad \text{setsum}(f, A) \$+ \text{setsum}(f, B) \$- \text{setsum}(f, A \cap B) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setsum-zneg-or-0* [rule-format (no-asm)]:

$$\begin{aligned} & \text{Finite}(A) \implies (\forall x \in A. g(x) \$\leq \#0) \longrightarrow \text{setsum}(g, A) \$\leq \#0 \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setsum-succD-lemma* [rule-format]:

$$\begin{aligned} & \text{Finite}(A) \\ & \implies \forall n \in \text{nat}. \text{setsum}(f, A) = \$\# \text{succ}(n) \longrightarrow (\exists a \in A. \#0 \$< f(a)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setsum-succD*:

$$\begin{aligned} & \llbracket \text{setsum}(f, A) = \$\# \text{succ}(n); n \in \text{nat} \rrbracket \implies \exists a \in A. \#0 \$< f(a) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *g-zpos-imp-setsum-zpos* [rule-format]:

$$\begin{aligned} & \text{Finite}(A) \implies (\forall x \in A. \#0 \$\leq g(x)) \longrightarrow \#0 \$\leq \text{setsum}(g, A) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *g-zpos-imp-setsum-zpos2* [rule-format]:

$$\begin{aligned} & \llbracket \text{Finite}(A); \forall x. \#0 \$\leq g(x) \rrbracket \implies \#0 \$\leq \text{setsum}(g, A) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *g-zspos-imp-setsum-zspos* [rule-format]:

$$\begin{aligned} & \text{Finite}(A) \\ & \implies (\forall x \in A. \#0 \$< g(x)) \longrightarrow A \neq 0 \longrightarrow (\#0 \$< \text{setsum}(g, A)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setsum-Diff* [rule-format]:
 $Finite(A) \implies \forall a. M(a) = \#0 \longrightarrow setsum(M, A) = setsum(M, A - \{a\})$
 <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
image: $\llbracket r - \{\langle a \rangle\} : Pow(acc(r)); a \in field(r) \rrbracket \implies a \in acc(r)$
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*: $\llbracket \bigwedge b. \langle b, a \rangle : r \implies b \in acc(r); a \in field(r) \rrbracket \implies a \in acc(r)$
 <proof>

lemma *acc-downward*: $\llbracket b \in acc(r); \langle a, b \rangle : r \rrbracket \implies a \in acc(r)$
 <proof>

lemma *acc-induct* [consumes 1, case-names *image*, induct set: *acc*]:
 $\llbracket a \in acc(r); \bigwedge x. \llbracket x \in acc(r); \forall y. \langle y, x \rangle : r \longrightarrow P(y) \rrbracket \implies P(x) \rrbracket \implies P(a)$
 <proof>

lemma *wf-on-acc*: $wf[acc(r)](r)$
 <proof>

lemma *acc-wfI*: $field(r) \subseteq acc(r) \implies wf(r)$
 <proof>

lemma *acc-wfD*: $wf(r) \implies field(r) \subseteq acc(r)$
 <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
 $Mult :: i \Rightarrow i$ **where**
 $Mult(A) \equiv A -||> nat-\{0\}$

definition

$funrestrict :: [i, i] \Rightarrow i$ **where**
 $funrestrict(f, A) \equiv \lambda x \in A. f'x$

definition

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

definition

$mset-of :: i \Rightarrow i$ **where**
 $mset-of(M) \equiv domain(M)$

definition

$munion :: [i, i] \Rightarrow i$ (**infixl** $\langle + \# \rangle$ 65) **where**
 $M + \# N \equiv \lambda x \in mset-of(M) \cup mset-of(N).$
 if $x \in mset-of(M) \cap mset-of(N)$ then $(M'x) \# + (N'x)$
 else (if $x \in mset-of(M)$ then $M'x$ else $N'x$)

definition

$normalize :: i \Rightarrow i$ **where**
 $normalize(f) \equiv$
 if $(\exists A. f \in A -> nat \wedge Finite(A))$ then
 $funrestrict(f, \{x \in mset-of(f). 0 < f'x\})$
 else 0

definition

$mdiff :: [i, i] \Rightarrow i$ (**infixl** $\langle - \# \rangle$ 65) **where**
 $M - \# N \equiv normalize(\lambda x \in mset-of(M).$
 if $x \in mset-of(N)$ then $M'x \# - N'x$ else $M'x$)

definition

$msingle :: i \Rightarrow i$ ($\langle (\langle open-block\ notation = \langle mixfix\ multiset \rangle \{ \# - \# \}) \rangle \rangle$) **where**

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

definition

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

definition

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

definition

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

abbreviation

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

syntax

$-MColl :: [pttrn, i, o] \Rightarrow i$ $(\langle \langle indent = 1\ notation = \langle mixfix multiset comprehension \rangle \{ \# - \in - / - \# \} \rangle \rangle)$

syntax-consts

$-MColl \Rightarrow MCollect$

translations

$\{\#x \in M. P\# \} == CONST\ MCollect(M, \lambda x. P)$

definition

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

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

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

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

definition

$mle :: [i, i] \Rightarrow o$ (**infixl** $< \# =$ 50) **where**
 $M < \# = N \equiv (\text{omultiset}(M) \wedge M = N) \mid M < \# N$

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

lemma *funrestrict-subset*: $\llbracket f \in \text{Pi}(C, B); A \subseteq C \rrbracket \Longrightarrow \text{funrestrict}(f, A) \subseteq f$
 $\langle \text{proof} \rangle$

lemma *funrestrict-type*:

$\llbracket \bigwedge x. x \in A \Longrightarrow f'x \in B(x) \rrbracket \Longrightarrow \text{funrestrict}(f, A) \in \text{Pi}(A, B)$
 $\langle \text{proof} \rangle$

lemma *funrestrict-type2*: $\llbracket f \in \text{Pi}(C, B); A \subseteq C \rrbracket \Longrightarrow \text{funrestrict}(f, A) \in \text{Pi}(A, B)$
 $\langle \text{proof} \rangle$

lemma *funrestrict [simp]*: $a \in A \Longrightarrow \text{funrestrict}(f, A) ' a = f'a$
 $\langle \text{proof} \rangle$

lemma *funrestrict-empty [simp]*: $\text{funrestrict}(f, 0) = 0$
 $\langle \text{proof} \rangle$

lemma *domain-funrestrict [simp]*: $\text{domain}(\text{funrestrict}(f, C)) = C$
 $\langle \text{proof} \rangle$

lemma *fun-cons-funrestrict-eq*:

$f \in \text{cons}(a, b) \rightarrow B \Longrightarrow f = \text{cons}(\langle a, f ' a \rangle, \text{funrestrict}(f, b))$
 $\langle \text{proof} \rangle$

declare *domain-of-fun [simp]*

declare *domainE [rule del]*

A useful simplification rule

lemma *multiset-fun-iff*:

$(f \in A \rightarrow \text{nat} - \{0\}) \longleftrightarrow f \in A \rightarrow \text{nat} \wedge (\forall a \in A. f'a \in \text{nat} \wedge 0 < f'a)$
 $\langle \text{proof} \rangle$

lemma *multiset-into-Mult*: $\llbracket \text{multiset}(M); \text{mset-of}(M) \subseteq A \rrbracket \Longrightarrow M \in \text{Mult}(A)$
 $\langle \text{proof} \rangle$

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

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

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

The *multiset* operator

lemma *multiset-0* [simp]: $\text{multiset}(0)$
 $\langle \text{proof} \rangle$

The *mset-of* operator

lemma *multiset-set-of-Finite* [simp]: $\text{multiset}(M) \implies \text{Finite}(\text{mset-of}(M))$
 $\langle \text{proof} \rangle$

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

lemma *mset-is-0-iff*: $\text{multiset}(M) \implies \text{mset-of}(M)=0 \longleftrightarrow M=0$
 $\langle \text{proof} \rangle$

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

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

lemma *mset-of-diff* [simp]: $\text{mset-of}(M) \subseteq A \implies \text{mset-of}(M -\# N) \subseteq A$
 $\langle \text{proof} \rangle$

lemma *msingle-not-0* [iff]: $\{\#a\#\} \neq 0 \wedge 0 \neq \{\#a\#\}$
 $\langle \text{proof} \rangle$

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

lemma *msingle-multiset* [iff, TC]: $\text{multiset}(\{\#a\#\})$
 $\langle \text{proof} \rangle$

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

lemma *normalize-idem* [simp]: $\text{normalize}(\text{normalize}(f)) = \text{normalize}(f)$
 $\langle \text{proof} \rangle$

lemma *normalize-multiset* [simp]: $\text{multiset}(M) \implies \text{normalize}(M) = M$
 $\langle \text{proof} \rangle$

lemma *multiset-normalize* [simp]: $\text{multiset}(\text{normalize}(f))$
 $\langle \text{proof} \rangle$

lemma *munion-multiset* [simp]: $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies \text{multiset}(M +\# N)$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

lemma *mcount-0* [simp]: $\text{mcount}(0, a) = 0$
 $\langle \text{proof} \rangle$

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

lemma *mcount-union* [simp]: $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket$
 $\implies \text{mcount}(M +\# N, a) = \text{mcount}(M, a) \#+ \text{mcount}(N, a)$
 $\langle \text{proof} \rangle$

lemma *mcount-diff* [simp]:
 $\text{multiset}(M) \implies \text{mcount}(M -\# N, a) = \text{mcount}(M, a) \#- \text{mcount}(N, a)$
 $\langle \text{proof} \rangle$

lemma *mcount-elem*: $\llbracket \text{multiset}(M); a \in \text{mset-of}(M) \rrbracket \implies 0 < \text{mcount}(M, a)$
 $\langle \text{proof} \rangle$

lemma *msize-0* [simp]: $\text{msize}(0) = \#0$
 $\langle \text{proof} \rangle$

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

lemma *msize-type* [simp, TC]: $\text{msize}(M) \in \text{int}$
 $\langle \text{proof} \rangle$

lemma *msize-zpositive*: $\text{multiset}(M) \implies \#0 \leq \text{msize}(M)$
 $\langle \text{proof} \rangle$

lemma *msize-int-of-nat*: $\text{multiset}(M) \implies \exists n \in \text{nat}. \text{msize}(M) = \#n$
 $\langle \text{proof} \rangle$

lemma *not-empty-multiset-imp-exist*:
 $\llbracket M \neq 0; \text{multiset}(M) \rrbracket \implies \exists a \in \text{mset-of}(M). 0 < \text{mcount}(M, a)$
 $\langle \text{proof} \rangle$

lemma *msize-eq-0-iff*: $\text{multiset}(M) \implies \text{msize}(M) = \#0 \longleftrightarrow M = 0$
 $\langle \text{proof} \rangle$

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

lemma *msize-union* [simp]:

$$\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies \text{msize}(M +\# N) = \text{msize}(M) \$+ \text{msize}(N)$$

$\langle \text{proof} \rangle$

lemma *msize-eq-succ-imp-lem*: $\llbracket \text{msize}(M) = \$\# \text{succ}(n); n \in \text{nat} \rrbracket \implies \exists a. a \in \text{mset-of}(M)$

$\langle \text{proof} \rangle$

lemma *equality-lemma*:

$$\begin{aligned} & \llbracket \text{multiset}(M); \text{multiset}(N); \forall a. \text{mcount}(M, a) = \text{mcount}(N, a) \rrbracket \\ & \implies \text{mset-of}(M) = \text{mset-of}(N) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *multiset-equality*:

$$\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies M = N \longleftrightarrow (\forall a. \text{mcount}(M, a) = \text{mcount}(N, a))$$

$\langle \text{proof} \rangle$

lemma *munion-eq-0-iff* [simp]: $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies (M +\# N = 0) \longleftrightarrow (M = 0 \wedge N = 0)$

$\langle \text{proof} \rangle$

lemma *empty-eq-munion-iff* [simp]: $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies (0 = M +\# N) \longleftrightarrow (M = 0 \wedge N = 0)$

$\langle \text{proof} \rangle$

lemma *munion-right-cancel* [simp]:

$$\llbracket \text{multiset}(M); \text{multiset}(N); \text{multiset}(K) \rrbracket \implies (M +\# K = N +\# K) \longleftrightarrow (M = N)$$

$\langle \text{proof} \rangle$

lemma *munion-left-cancel* [simp]:

$$\llbracket \text{multiset}(K); \text{multiset}(M); \text{multiset}(N) \rrbracket \implies (K +\# M = K +\# N) \longleftrightarrow (M = N)$$

$\langle \text{proof} \rangle$

lemma *nat-add-eq-1-cases*: $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies (m \#+ n = 1) \longleftrightarrow (m = 1 \wedge n = 0) \mid (m = 0 \wedge n = 1)$

$\langle \text{proof} \rangle$

lemma *munion-is-single*:

$$\begin{aligned} & \llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \\ & \implies (M +\# N = \{\#a\# \}) \longleftrightarrow (M = \{\#a\# \} \wedge N = 0) \mid (M = 0 \wedge N = \{\#a\# \}) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *msingle-is-union*: $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket$

$\implies (\{\#a\# \} = M + \# N) \longleftrightarrow (\{\#a\# \} = M \wedge N=0 \mid M = 0 \wedge \{\#a\# \} = N)$
 $\langle \text{proof} \rangle$

lemma *setsum-decr*:

$\text{Finite}(A)$
 $\implies (\forall M. \text{multiset}(M) \longrightarrow$
 $(\forall a \in \text{mset-of}(M). \text{setsum}(\lambda z. \$\# \text{mcount}(M(a:=M'a \#- 1), z), A) =$
 $(\text{if } a \in A \text{ then } \text{setsum}(\lambda z. \$\# \text{mcount}(M, z), A) \$- \#1$
 $\quad \text{else } \text{setsum}(\lambda z. \$\# \text{mcount}(M, z), A)))$
 $\langle \text{proof} \rangle$

lemma *setsum-decr2*:

$\text{Finite}(A)$
 $\implies \forall M. \text{multiset}(M) \longrightarrow (\forall a \in \text{mset-of}(M).$
 $\text{setsum}(\lambda x. \$\# \text{mcount}(\text{funrestrict}(M, \text{mset-of}(M)-\{a\}), x), A) =$
 $(\text{if } a \in A \text{ then } \text{setsum}(\lambda x. \$\# \text{mcount}(M, x), A) \$- \$\# M'a$
 $\quad \text{else } \text{setsum}(\lambda x. \$\# \text{mcount}(M, x), A)))$
 $\langle \text{proof} \rangle$

lemma *setsum-decr3*: $\llbracket \text{Finite}(A); \text{multiset}(M); a \in \text{mset-of}(M) \rrbracket$

$\implies \text{setsum}(\lambda x. \$\# \text{mcount}(\text{funrestrict}(M, \text{mset-of}(M)-\{a\}), x), A - \{a\})$
 $=$
 $(\text{if } a \in A \text{ then } \text{setsum}(\lambda x. \$\# \text{mcount}(M, x), A) \$- \$\# M'a$
 $\quad \text{else } \text{setsum}(\lambda x. \$\# \text{mcount}(M, x), A))$
 $\langle \text{proof} \rangle$

lemma *nat-le-1-cases*: $n \in \text{nat} \implies n \leq 1 \longleftrightarrow (n=0 \mid n=1)$

$\langle \text{proof} \rangle$

lemma *succ-pred-eq-self*: $\llbracket 0 < n; n \in \text{nat} \rrbracket \implies \text{succ}(n \#- 1) = n$

$\langle \text{proof} \rangle$

Specialized for use in the proof below.

lemma *multiset-funrestrict*:

$\llbracket \forall a \in A. M' a \in \text{nat} \wedge 0 < M' a; \text{Finite}(A) \rrbracket$
 $\implies \text{multiset}(\text{funrestrict}(M, A - \{a\}))$
 $\langle \text{proof} \rangle$

lemma *multiset-induct-aux*:

assumes *prem1*: $\bigwedge M a. \llbracket \text{multiset}(M); a \notin \text{mset-of}(M); P(M) \rrbracket \implies P(\text{cons}(\langle a, 1 \rangle, M))$
and *prem2*: $\bigwedge M b. \llbracket \text{multiset}(M); b \in \text{mset-of}(M); P(M) \rrbracket \implies P(M(b:= M'b \#+ 1))$
shows
 $\llbracket n \in \text{nat}; P(0) \rrbracket$
 $\implies (\forall M. \text{multiset}(M) \longrightarrow$
 $(\text{setsum}(\lambda x. \$\# \text{mcount}(M, x), \{x \in \text{mset-of}(M). 0 < M'x\}) = \$\# n) \longrightarrow P(M))$

$\langle proof \rangle$

lemma *multiset-induct2*:

$\llbracket multiset(M); P(0);$
 $(\bigwedge M a. \llbracket multiset(M); a \notin mset-of(M); P(M) \rrbracket \implies P(cons(\langle a, 1 \rangle, M)))$;
 $(\bigwedge M b. \llbracket multiset(M); b \in mset-of(M); P(M) \rrbracket \implies P(M(b := M \cdot b \# + 1))) \rrbracket$
 $\implies P(M)$
 $\langle proof \rangle$

lemma *munion-single-case1*:

$\llbracket multiset(M); a \notin mset-of(M) \rrbracket \implies M + \# \{ \# a \# \} = cons(\langle a, 1 \rangle, M)$
 $\langle proof \rangle$

lemma *munion-single-case2*:

$\llbracket multiset(M); a \in mset-of(M) \rrbracket \implies M + \# \{ \# a \# \} = M(a := M \cdot a \# + 1)$
 $\langle proof \rangle$

lemma *multiset-induct*:

assumes $M: multiset(M)$
and $P0: P(0)$
and step: $\bigwedge M a. \llbracket multiset(M); P(M) \rrbracket \implies P(M + \# \{ \# a \# \})$
shows $P(M)$
 $\langle proof \rangle$

lemma *MCollect-multiset [simp]*:

$multiset(M) \implies multiset(\{ \# x \in M. P(x) \# \})$
 $\langle proof \rangle$

lemma *mset-of-MCollect [simp]*:

$multiset(M) \implies mset-of(\{ \# x \in M. P(x) \# \}) \subseteq mset-of(M)$
 $\langle proof \rangle$

lemma *MCollect-mem-iff [iff]*:

$x \in mset-of(\{ \# x \in M. P(x) \# \}) \longleftrightarrow x \in mset-of(M) \wedge P(x)$
 $\langle proof \rangle$

lemma *mcount-MCollect [simp]*:

$mcount(\{ \# x \in M. P(x) \# \}, a) = (if P(a) then mcount(M, a) else 0)$
 $\langle proof \rangle$

lemma *multiset-partition*: $multiset(M) \implies M = \{ \# x \in M. P(x) \# \} + \# \{ \# x \in M. \neg P(x) \# \}$
 $\langle proof \rangle$

lemma *natify-elem-is-self [simp]*:

$\llbracket \text{multiset}(M); a \in \text{mset-of}(M) \rrbracket \implies \text{nativify}(M'a) = M'a$
 $\langle \text{proof} \rangle$

lemma *munion-eq-conv-diff*: $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket$
 $\implies (M + \# \{ \#a\# \} = N + \# \{ \#b\# \}) \longleftrightarrow (M = N \wedge a = b \mid$
 $M = N - \# \{ \#a\# \} + \# \{ \#b\# \} \wedge N = M - \# \{ \#b\# \} + \# \{ \#a\# \})$
 $\langle \text{proof} \rangle$

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

lemma *munion-eq-conv-exist*:
 $\llbracket M \in \text{Mult}(A); N \in \text{Mult}(A) \rrbracket$
 $\implies (M + \# \{ \#a\# \} = N + \# \{ \#b\# \}) \longleftrightarrow$
 $(M=N \wedge a=b \mid (\exists K \in \text{Mult}(A). M = K + \# \{ \#b\# \} \wedge N = K + \# \{ \#a\# \}))$
 $\langle \text{proof} \rangle$

8.2 Multiset Orderings

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

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

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

Monotonicity of *multirel1*

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

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

lemma *multirel1-mono*:
 $\llbracket A \subseteq B; r \subseteq s \rrbracket \implies \text{multirel1}(A, r) \subseteq \text{multirel1}(B, s)$
 $\langle \text{proof} \rangle$

8.3 Toward the proof of well-foundedness of multirel1

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

<proof>

lemma *less-munion*: $\llbracket \langle N, M0 +\# \{\#a\# \} \rangle \in \text{multirel1}(A, r); M0 \in \text{Mult}(A) \rrbracket$

\implies

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

<proof>

lemma *multirel1-base*: $\llbracket M \in \text{Mult}(A); a \in A \rrbracket \implies \langle M, M +\# \{\#a\# \} \rangle \in \text{multirel1}(A, r)$

<proof>

lemma *acc-0*: $\text{acc}(0) = 0$

<proof>

lemma *lemma1*: $\llbracket \forall b \in A. \langle b, a \rangle \in r \longrightarrow$

$(\forall M \in \text{acc}(\text{multirel1}(A, r)). M +\# \{\#b\# \} : \text{acc}(\text{multirel1}(A, r)))$;

$M0 \in \text{acc}(\text{multirel1}(A, r)); a \in A$;

$\forall M. \langle M, M0 \rangle \in \text{multirel1}(A, r) \longrightarrow M +\# \{\#a\# \} \in \text{acc}(\text{multirel1}(A, r)) \rrbracket$

$\implies M0 +\# \{\#a\# \} \in \text{acc}(\text{multirel1}(A, r))$

<proof>

lemma *lemma2*: $\llbracket \forall b \in A. \langle b, a \rangle \in r$

$\longrightarrow (\forall M \in \text{acc}(\text{multirel1}(A, r)). M +\# \{\#b\# \} : \text{acc}(\text{multirel1}(A, r)))$;

$M \in \text{acc}(\text{multirel1}(A, r)); a \in A \rrbracket \implies M +\# \{\#a\# \} \in \text{acc}(\text{multirel1}(A,$

$r))$

<proof>

lemma *lemma3*: $\llbracket \text{wf}[A](r); a \in A \rrbracket$

$\implies \forall M \in \text{acc}(\text{multirel1}(A, r)). M +\# \{\#a\# \} \in \text{acc}(\text{multirel1}(A, r))$

<proof>

lemma *lemma4*: $\text{multiset}(M) \implies \text{mset-of}(M) \subseteq A \longrightarrow$

$\text{wf}[A](r) \longrightarrow M \in \text{field}(\text{multirel1}(A, r)) \longrightarrow M \in \text{acc}(\text{multirel1}(A, r))$

<proof>

lemma *all-accessible*: $\llbracket \text{wf}[A](r); M \in \text{Mult}(A); A \neq 0 \rrbracket \implies M \in \text{acc}(\text{multirel1}(A,$

$r))$

<proof>

lemma *wf-on-multirel1*: $\text{wf}[A](r) \implies \text{wf}[A - \{\{0\}\}](\text{multirel1}(A, r))$

<proof>

lemma *wf-multirel1*: $\text{wf}(r) \implies \text{wf}(\text{multirel1}(\text{field}(r), r))$

<proof>

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

lemma *multirel-mono*:
 $\llbracket A \subseteq B; r \subseteq s \rrbracket \implies \text{multirel}(A, r) \subseteq \text{multirel}(B, s)$
 $\langle \text{proof} \rangle$

lemma *add-diff-eq*: $k \in \text{nat} \implies 0 < k \longrightarrow n \# + k \# - 1 = n \# + (k \# - 1)$
 $\langle \text{proof} \rangle$

lemma *mdiff-union-single-conv*: $\llbracket a \in \text{mset-of}(J); \text{multiset}(I); \text{multiset}(J) \rrbracket$
 $\implies I \# + J \# - \{ \# a \# \} = I \# + (J \# - \{ \# a \# \})$
 $\langle \text{proof} \rangle$

lemma *diff-add-commute*: $\llbracket n \leq m; m \in \text{nat}; n \in \text{nat}; k \in \text{nat} \rrbracket \implies m \# - n \# + k = m \# + k \# - n$
 $\langle \text{proof} \rangle$

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

lemma *melem-imp-eq-diff-union* [simp]: $\llbracket a \in \text{mset-of}(M); \text{multiset}(M) \rrbracket \implies M \# - \{ \# a \# \} \# + \{ \# a \# \} = M$
 $\langle \text{proof} \rangle$

lemma *msize-eq-succ-imp-eq-union*:
 $\llbracket \text{msize}(M) = \$ \# \text{ succ}(n); M \in \text{Mult}(A); n \in \text{nat} \rrbracket$
 $\implies \exists a N. M = N \# + \{ \# a \# \} \wedge N \in \text{Mult}(A) \wedge a \in A$
 $\langle \text{proof} \rangle$

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

$\longrightarrow \langle I +\# K, I +\# J \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

lemma *one-step-implies-multirel*:

$\llbracket J \neq 0; \forall k \in \text{mset-of}(K). \exists j \in \text{mset-of}(J). \langle k, j \rangle \in r; \\ I \in \text{Mult}(A); J \in \text{Mult}(A); K \in \text{Mult}(A) \rrbracket \\ \implies \langle I +\# K, I +\# J \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

lemma *multirel-irrefl-lemma*:

$\text{Finite}(A) \implies \text{part-ord}(A, r) \longrightarrow (\forall x \in A. \exists y \in A. \langle x, y \rangle \in r) \longrightarrow A=0$
 $\langle \text{proof} \rangle$

lemma *irrefl-on-multirel*:

$\text{part-ord}(A, r) \implies \text{irrefl}(\text{Mult}(A), \text{multirel}(A, r))$
 $\langle \text{proof} \rangle$

lemma *trans-on-multirel*: $\text{trans}[\text{Mult}(A)](\text{multirel}(A, r))$
 $\langle \text{proof} \rangle$

lemma *multirel-trans*:

$\llbracket \langle M, N \rangle \in \text{multirel}(A, r); \langle N, K \rangle \in \text{multirel}(A, r) \rrbracket \implies \langle M, K \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

lemma *trans-multirel*: $\text{trans}(\text{multirel}(A, r))$
 $\langle \text{proof} \rangle$

lemma *part-ord-multirel*: $\text{part-ord}(A, r) \implies \text{part-ord}(\text{Mult}(A), \text{multirel}(A, r))$
 $\langle \text{proof} \rangle$

lemma *munion-multirel1-mono*:

$\llbracket \langle M, N \rangle \in \text{multirel1}(A, r); K \in \text{Mult}(A) \rrbracket \implies \langle K +\# M, K +\# N \rangle \in \text{multirel1}(A, r)$
 $\langle \text{proof} \rangle$

lemma *munion-multirel-mono2*:

$\llbracket \langle M, N \rangle \in \text{multirel}(A, r); K \in \text{Mult}(A) \rrbracket \implies \langle K +\# M, K +\# N \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

lemma *munion-multirel-mono1*:

$\llbracket \langle M, N \rangle \in \text{multirel}(A, r); K \in \text{Mult}(A) \rrbracket \implies \langle M +\# K, N +\# K \rangle \in \text{multirel}(A, r)$

$\langle \text{proof} \rangle$

lemma *munion-multirel-mono*:

$$\llbracket \langle M, K \rangle \in \text{multirel}(A, r); \langle N, L \rangle \in \text{multirel}(A, r) \rrbracket \\ \implies \langle M +\# N, K +\# L \rangle \in \text{multirel}(A, r)$$

$\langle \text{proof} \rangle$

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*]: $\text{omultiset}(M) \implies \text{multiset}(M)$

$\langle \text{proof} \rangle$

lemma *munion-omultiset* [*simp*]: $\llbracket \text{omultiset}(M); \text{omultiset}(N) \rrbracket \implies \text{omultiset}(M +\# N)$

$\langle \text{proof} \rangle$

lemma *mdiff-omultiset* [*simp*]: $\text{omultiset}(M) \implies \text{omultiset}(M -\# N)$

$\langle \text{proof} \rangle$

lemma *irrefl-Memrel*: $\text{Ord}(i) \implies \text{irrefl}(\text{field}(\text{Memrel}(i)), \text{Memrel}(i))$

$\langle \text{proof} \rangle$

lemma *trans-iff-trans-on*: $\text{trans}(r) \longleftrightarrow \text{trans}[\text{field}(r)](r)$

$\langle \text{proof} \rangle$

lemma *part-ord-Memrel*: $\text{Ord}(i) \implies \text{part-ord}(\text{field}(\text{Memrel}(i)), \text{Memrel}(i))$

$\langle \text{proof} \rangle$

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

lemma *mless-not-refl*: $\neg(M <\# M)$

$\langle \text{proof} \rangle$

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

lemma *mless-trans*: $\llbracket K <\# M; M <\# N \rrbracket \implies K <\# N$
 $\langle proof \rangle$

lemma *mless-not-sym*: $M <\# N \implies \neg N <\# M$
 $\langle proof \rangle$

lemma *mless-asym*: $\llbracket M <\# N; \neg P \rrbracket \implies N <\# M \implies P$
 $\langle proof \rangle$

lemma *mle-refl [simp]*: $omultiset(M) \implies M <\# = M$
 $\langle proof \rangle$

lemma *mle-antisym*:
 $\llbracket M <\# = N; N <\# = M \rrbracket \implies M = N$
 $\langle proof \rangle$

lemma *mle-trans*: $\llbracket K <\# = M; M <\# = N \rrbracket \implies K <\# = N$
 $\langle proof \rangle$

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

lemma *munion-less-mono2*: $\llbracket M <\# N; omultiset(K) \rrbracket \implies K +\# M <\# K +\# N$
 $\langle proof \rangle$

lemma *munion-less-mono1*: $\llbracket M <\# N; omultiset(K) \rrbracket \implies M +\# K <\# N +\# K$
 $\langle proof \rangle$

lemma *mless-imp-omultiset*: $M <\# N \implies omultiset(M) \wedge omultiset(N)$
 $\langle proof \rangle$

lemma *munion-less-mono*: $\llbracket M <\# K; N <\# L \rrbracket \implies M +\# N <\# K +\# L$
 $\langle proof \rangle$

lemma *mle-imp-omultiset*: $M <\# = N \implies omultiset(M) \wedge omultiset(N)$
 $\langle proof \rangle$

lemma *mle-mono*: $\llbracket M <\# = K; N <\# = L \rrbracket \implies M +\# N <\# = K +\# L$
 $\langle proof \rangle$

lemma *omultiset-0* [iff]: *omultiset*(0)

<proof>

lemma *empty-leI* [simp]: *omultiset*(*M*) \implies 0 <# = *M*

<proof>

lemma *munion-upper1*: $\llbracket \text{omultiset}(M); \text{omultiset}(N) \rrbracket \implies M <\# = M +\# N$

<proof>

end

9 An operator to “map” a relation over a list

theory *Rmap* imports *ZF* begin

consts

rmap :: $i \Rightarrow i$

inductive

domains *rmap*(*r*) \subseteq *list*(*domain*(*r*)) \times *list*(*range*(*r*))

intros

NilI: $\langle \text{Nil}, \text{Nil} \rangle \in \text{rmap}(r)$

ConsI: $\llbracket \langle x, y \rangle: r; \langle xs, ys \rangle \in \text{rmap}(r) \rrbracket$
 $\implies \langle \text{Cons}(x, xs), \text{Cons}(y, ys) \rangle \in \text{rmap}(r)$

type-intros *domainI* *rangeI* *list.intros*

lemma *rmap-mono*: $r \subseteq s \implies \text{rmap}(r) \subseteq \text{rmap}(s)$

<proof>

inductive-cases

Nil-rmap-case [elim!]: $\langle \text{Nil}, zs \rangle \in \text{rmap}(r)$

and *Cons-rmap-case* [elim!]: $\langle \text{Cons}(x, xs), zs \rangle \in \text{rmap}(r)$

declare *rmap.intros* [intro]

lemma *rmap-rel-type*: $r \subseteq A \times B \implies \text{rmap}(r) \subseteq \text{list}(A) \times \text{list}(B)$

<proof>

lemma *rmap-total*: $A \subseteq \text{domain}(r) \implies \text{list}(A) \subseteq \text{domain}(\text{rmap}(r))$

<proof>

lemma *rmap-functional*: $\text{function}(r) \implies \text{function}(\text{rmap}(r))$

<proof>

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

lemma *rmap-fun-type*: $f \in A \multimap B \implies \text{rmap}(f): \text{list}(A) \multimap \text{list}(B)$
 $\langle \text{proof} \rangle$

lemma *rmap-Nil*: $\text{rmap}(f) \text{ `Nil} = \text{Nil}$
 $\langle \text{proof} \rangle$

lemma *rmap-Cons*: $\llbracket f \in A \multimap B; x \in A; xs: \text{list}(A) \rrbracket$
 $\implies \text{rmap}(f) \text{ `Cons}(x, xs) = \text{Cons}(f \text{ `} x, \text{rmap}(f) \text{ `} xs)$
 $\langle \text{proof} \rangle$

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 $H \models p$ then $G \models p$ where $G \in \text{Fin}(H)$

10.1 The datatype of propositions

consts

propn :: *i*

datatype *propn* =

Fls
 $| \text{Var } (n \in \text{nat}) \quad (\langle \# \rightarrow [100] \ 100)$
 $| \text{Imp } (p \in \text{propn}, q \in \text{propn}) \quad (\text{infixr } \langle \Rightarrow \rangle \ 90)$

10.2 The proof system

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

abbreviation

thms-syntax :: $[i, i] \Rightarrow o \quad (\text{infixl } \langle |- \rangle \ 50)$
where $H \vdash p \equiv p \in \text{thms}(H)$

inductive

domains $\text{thms}(H) \subseteq \text{propn}$

intros

$H: \llbracket p \in H; p \in \text{propn} \rrbracket \implies H \vdash p$
 $K: \llbracket p \in \text{propn}; q \in \text{propn} \rrbracket \implies H \vdash p \Rightarrow q \Rightarrow p$
 $S: \llbracket p \in \text{propn}; q \in \text{propn}; r \in \text{propn} \rrbracket$
 $\implies H \vdash (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$
 $DN: p \in \text{propn} \implies H \vdash ((p \Rightarrow \text{Fls}) \Rightarrow \text{Fls}) \Rightarrow p$

MP: $\llbracket H \mid - p \Rightarrow q; H \mid - p; p \in \text{propn}; q \in \text{propn} \rrbracket \implies H \mid - q$
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** $\langle | \Rightarrow \rangle$ 50) **where**

$H \mid = p \equiv \forall t. (\forall q \in H. \text{is-true}(q, t)) \longrightarrow \text{is-true}(p, t)$

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

consts

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

primrec

hyps(*Fls*, *t*) = 0

hyps(*Var*(*v*), *t*) = (if $v \in t$ then $\{\#v\}$ else $\{\#v \Rightarrow \text{Fls}\}$)

hyps($p \Rightarrow q$, *t*) = *hyps*(*p*, *t*) \cup *hyps*(*q*, *t*)

10.4 Proof theory of propositional logic

lemma *thms-mono*: $G \subseteq H \implies \text{thms}(G) \subseteq \text{thms}(H)$
 <proof>

lemmas $thms\text{-}in\text{-}pl = thms.\text{dom}\text{-}subset \ [THEN\ subsetD]$

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

lemma $thms\text{-}MP: \llbracket H \mid - p \Rightarrow q; \ H \mid - p \rrbracket \Longrightarrow H \mid - q$
 — Stronger Modus Ponens rule: no typechecking!
 $\langle proof \rangle$

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

10.4.1 Weakening, left and right

lemma $weaken\text{-}left: \llbracket G \subseteq H; \ G \mid - p \rrbracket \Longrightarrow H \mid - p$
 — Order of premises is convenient with $THEN$
 $\langle proof \rangle$

lemma $weaken\text{-}left\text{-}cons: H \mid - p \Longrightarrow cons(a, H) \mid - p$
 $\langle proof \rangle$

lemmas $weaken\text{-}left\text{-}Un1 = Un\text{-}upper1 \ [THEN\ weaken\text{-}left]$
lemmas $weaken\text{-}left\text{-}Un2 = Un\text{-}upper2 \ [THEN\ weaken\text{-}left]$

lemma $weaken\text{-}right: \llbracket H \mid - q; \ p \in propn \rrbracket \Longrightarrow H \mid - p \Rightarrow q$
 $\langle proof \rangle$

10.4.2 The deduction theorem

theorem $deduction: \llbracket cons(p, H) \mid - q; \ p \in propn \rrbracket \Longrightarrow H \mid - p \Rightarrow q$
 $\langle proof \rangle$

10.4.3 The cut rule

lemma $cut: \llbracket H \mid - p; \ cons(p, H) \mid - q \rrbracket \Longrightarrow H \mid - q$
 $\langle proof \rangle$

lemma $thms\text{-}FlsE: \llbracket H \mid - Fls; \ p \in propn \rrbracket \Longrightarrow H \mid - p$
 $\langle proof \rangle$

lemma $thms\text{-}notE: \llbracket H \mid - p \Rightarrow Fls; \ H \mid - p; \ q \in propn \rrbracket \Longrightarrow H \mid - q$
 $\langle proof \rangle$

10.4.4 Soundness of the rules wrt truth-table semantics

theorem $soundness: H \mid - p \Longrightarrow H \models p$
 $\langle proof \rangle$

10.5 Completeness

10.5.1 Towards the completeness proof

lemma *Fls-Imp*: $\llbracket H \mid - p \Rightarrow Fls; q \in propn \rrbracket \Longrightarrow H \mid - p \Rightarrow q$
 $\langle proof \rangle$

lemma *Imp-Fls*: $\llbracket H \mid - p; H \mid - q \Rightarrow Fls \rrbracket \Longrightarrow H \mid - (p \Rightarrow q) \Rightarrow Fls$
 $\langle proof \rangle$

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

lemma *logcon-thms-p*: $\llbracket p \in propn; 0 \models p \rrbracket \Longrightarrow hyps(p, t) \mid - p$
 — Key lemma for completeness; yields a set of assumptions satisfying p
 $\langle proof \rangle$

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*:
 $\llbracket p \in propn; q \in propn \rrbracket \Longrightarrow H \mid - (p \Rightarrow q) \Rightarrow ((p \Rightarrow Fls) \Rightarrow q) \Rightarrow q$
 $\langle proof \rangle$

lemma *thms-excluded-middle-rule*:
 $\llbracket cons(p, H) \mid - q; cons(p \Rightarrow Fls, H) \mid - q; p \in propn \rrbracket \Longrightarrow H \mid - q$
 — Hard to prove directly because it requires cuts
 $\langle proof \rangle$

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

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

Two lemmas for use with *weaken-left*

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

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

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

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

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

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

lemma *completeness-0-lemma* [*rule-format*]:
 $\llbracket p \in \text{propn}; 0 \models p \rrbracket \implies \forall t. \text{hyps}(p, t) - \text{hyps}(p, t0) \vdash p$
 ⟨proof⟩

10.5.3 Completeness theorem

lemma *completeness-0*: $\llbracket p \in \text{propn}; 0 \models p \rrbracket \implies 0 \vdash p$
 — The base case for completeness
 ⟨proof⟩

lemma *logcon-Imp*: $\llbracket \text{cons}(p, H) \models q \rrbracket \implies H \models p \Rightarrow q$
 — A semantic analogue of the Deduction Theorem
 ⟨proof⟩

lemma *completeness*:
 $H \in \text{Fin}(\text{propn}) \implies p \in \text{propn} \implies H \models p \implies H \vdash p$
 ⟨proof⟩

theorem *thms-iff*: $H \in \text{Fin}(\text{propn}) \implies H \vdash p \longleftrightarrow H \models p \wedge p \in \text{propn}$
 ⟨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 $\text{listn}(A) \subseteq \text{nat} \times \text{list}(A)$

intros

$NilI: \langle 0, Nil \rangle \in listn(A)$
 $ConsI: \llbracket a \in A; \langle n, l \rangle \in listn(A) \rrbracket \implies \langle succ(n), Cons(a, l) \rangle \in listn(A)$
type-intros *nat-typechecks list.intros*

lemma *list-into-listn*: $l \in list(A) \implies \langle length(l), l \rangle \in listn(A)$
 $\langle proof \rangle$

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

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

lemma *listn-mono*: $A \subseteq B \implies listn(A) \subseteq listn(B)$
 $\langle proof \rangle$

lemma *listn-append*:
 $\llbracket \langle n, l \rangle \in listn(A); \langle n', l' \rangle \in listn(A) \rrbracket \implies \langle n \# + n', l @ l' \rangle \in listn(A)$
 $\langle proof \rangle$

inductive-cases

$Nil\text{-}listn\text{-}case: \langle i, Nil \rangle \in listn(A)$
and $Cons\text{-}listn\text{-}case: \langle i, Cons(x, l) \rangle \in listn(A)$

inductive-cases

$zero\text{-}listn\text{-}case: \langle 0, l \rangle \in listn(A)$
and $succ\text{-}listn\text{-}case: \langle succ(i), l \rangle \in listn(A)$

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
| $app (p \in comb, q \in comb) \text{ (infixl } \langle \cdot \rangle 90)$

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

consts *contract* :: i

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

abbreviation *contract-multi* :: $[i, i] \Rightarrow o \text{ (infixl } \langle \rightarrow \rangle 50)$

where $p \rightarrow q \equiv \langle p, q \rangle \in contract^*$

inductive

domains *contract* $\subseteq comb \times comb$

intros

$K: \llbracket p \in comb; q \in comb \rrbracket \Longrightarrow K \cdot p \cdot q \rightarrow^1 p$

$S: \llbracket p \in comb; q \in comb; r \in comb \rrbracket \Longrightarrow S \cdot p \cdot q \cdot r \rightarrow^1 (p \cdot r) \cdot (q \cdot r)$

$Ap1: \llbracket p \rightarrow^1 q; r \in comb \rrbracket \Longrightarrow p \cdot r \rightarrow^1 q \cdot r$

$Ap2: \llbracket p \rightarrow^1 q; r \in comb \rrbracket \Longrightarrow r \cdot p \rightarrow^1 r \cdot q$

type-intros *comb.intros*

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

consts *parcontract* :: i

abbreviation *parcontract-syntax* :: $[i, i] \Rightarrow o \text{ (infixl } \langle \Rightarrow^1 \rangle 50)$

where $p \Rightarrow^1 q \equiv \langle p, q \rangle \in parcontract$

abbreviation *parcontract-multi* :: $[i, i] \Rightarrow o \text{ (infixl } \langle \Rightarrow \rangle 50)$

where $p \Rightarrow q \equiv \langle p, q \rangle \in parcontract^+$

inductive

domains *parcontract* $\subseteq comb \times comb$

intros

$refl: \llbracket p \in comb \rrbracket \Longrightarrow p \Rightarrow^1 p$

$K: \llbracket p \in comb; q \in comb \rrbracket \Longrightarrow K \cdot p \cdot q \Rightarrow^1 p$

$S: \llbracket p \in comb; q \in comb; r \in comb \rrbracket \Longrightarrow S \cdot p \cdot q \cdot r \Rightarrow^1 (p \cdot r) \cdot (q \cdot r)$

$Ap: \llbracket p \Rightarrow^1 q; r \Rightarrow^1 s \rrbracket \Longrightarrow p \cdot r \Rightarrow^1 q \cdot s$

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

$\forall x y. \langle x, y \rangle \in r \longrightarrow (\forall y'. \langle x, y' \rangle \in r \longrightarrow (\exists z. \langle y, z \rangle \in r \wedge \langle y', z \rangle \in r))$

12.2 Transitive closure preserves the Church-Rosser property

lemma *diamond-strip-lemmaD* [*rule-format*]:

$\llbracket \text{diamond}(r); \langle x, y \rangle : r^+ \rrbracket \implies$
 $\forall y'. \langle x, y' \rangle : r \longrightarrow (\exists z. \langle y', z \rangle : r^+ \wedge \langle y, z \rangle : r)$
 $\langle \text{proof} \rangle$

lemma *diamond-trancl*: $\text{diamond}(r) \implies \text{diamond}(r^+)$
 $\langle \text{proof} \rangle$

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

12.3 Results about Contraction

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

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*: $\text{field}(\text{contract}) = \text{comb}$
 $\langle \text{proof} \rangle$

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 $p \in \text{comb} \implies I \cdot p \rightarrow p$
— Example only: not used
 $\langle \text{proof} \rangle$

lemma *comb-I*: $I \in \text{comb}$
 $\langle \text{proof} \rangle$

12.4 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases *K-contractE* [*elim!*]: $K \rightarrow^1 r$

and $S\text{-contractE}$ $[elim!]: S \rightarrow^1 r$
and $Ap\text{-contractE}$ $[elim!]: p \cdot q \rightarrow^1 r$

lemma $I\text{-contract-E}$: $I \rightarrow^1 r \implies P$
 $\langle proof \rangle$

lemma $KI\text{-contractD}$: $K \cdot p \rightarrow^1 r \implies (\exists q. r = K \cdot q \wedge p \rightarrow^1 q)$
 $\langle proof \rangle$

lemma $Ap\text{-reduce1}$: $\llbracket p \rightarrow q; r \in comb \rrbracket \implies p \cdot r \rightarrow q \cdot r$
 $\langle proof \rangle$

lemma $Ap\text{-reduce2}$: $\llbracket p \rightarrow q; r \in comb \rrbracket \implies r \cdot p \rightarrow r \cdot q$
 $\langle proof \rangle$

Counterexample to the diamond property for \rightarrow^1 .

lemma $KIII\text{-contract1}$: $K \cdot I \cdot (I \cdot I) \rightarrow^1 I$
 $\langle proof \rangle$

lemma $KIII\text{-contract2}$: $K \cdot I \cdot (I \cdot I) \rightarrow^1 K \cdot I \cdot ((K \cdot I) \cdot (K \cdot I))$
 $\langle proof \rangle$

lemma $KIII\text{-contract3}$: $K \cdot I \cdot ((K \cdot I) \cdot (K \cdot I)) \rightarrow^1 I$
 $\langle proof \rangle$

lemma $not\text{-diamond-contract}$: $\neg diamond(contract)$
 $\langle proof \rangle$

12.5 Results about Parallel Contraction

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

lemmas $parcontract\text{-combE2} = parcontract.dom\text{-subset}$ $[THEN subsetD, THEN SigmaE2]$

and $parcontract\text{-combD1} = parcontract.dom\text{-subset}$ $[THEN subsetD, THEN SigmaD1]$

and $parcontract\text{-combD2} = parcontract.dom\text{-subset}$ $[THEN subsetD, THEN SigmaD2]$

lemma $field\text{-parcontract-eq}$: $field(parcontract) = comb$
 $\langle proof \rangle$

Derive a case for each combinator constructor.

inductive-cases

$K\text{-parcontractE}$ $[elim!]: K \Rightarrow^1 r$
and $S\text{-parcontractE}$ $[elim!]: S \Rightarrow^1 r$
and $Ap\text{-parcontractE}$ $[elim!]: p \cdot q \Rightarrow^1 r$

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 $p \rightarrow q$ and $p \Rightarrow q$.

lemma *contract-imp-parcontract*: $p \rightarrow^1 q \Longrightarrow p \Rightarrow^1 q$

<proof>

lemma *reduce-imp-parreduce*: $p \rightarrow q \Longrightarrow p \Rightarrow q$

<proof>

lemma *parcontract-imp-reduce*: $p \Rightarrow^1 q \Longrightarrow p \rightarrow q$

<proof>

lemma *parreduce-imp-reduce*: $p \Rightarrow q \Longrightarrow p \rightarrow q$

<proof>

lemma *parreduce-iff-reduce*: $p \Rightarrow q \longleftrightarrow p \rightarrow q$

<proof>

end

13 Primitive Recursive Functions: the inductive definition

theory *Primrec* imports *ZF* begin

Proof adopted from [4].

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

13.1 Basic definitions

definition

$SC :: i$ where

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

definition

$CONSTANT :: i \Rightarrow i$ **where**
 $CONSTANT(k) \equiv \lambda l \in \text{list}(\text{nat}). k$

definition

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

definition

$COMP :: [i, i] \Rightarrow i$ **where**
 $COMP(g, fs) \equiv \lambda l \in \text{list}(\text{nat}). g \text{ ' } \text{map}(\lambda f. f^l, fs)$

definition

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

consts

$ACK :: i \Rightarrow i$

primrec

$ACK(0) = SC$
 $ACK(\text{succ}(i)) = PREC (CONSTANT (ACK(i) \text{ ' } [1]), COMP(ACK(i), [PROJ(0)]))$

abbreviation

$ack :: [i, i] \Rightarrow i$ **where**
 $ack(x, y) \equiv ACK(x) \text{ ' } [y]$

Useful special cases of evaluation.

lemma SC : $\llbracket x \in \text{nat}; l \in \text{list}(\text{nat}) \rrbracket \Longrightarrow SC \text{ ' } (\text{Cons}(x, l)) = \text{succ}(x)$
 $\langle \text{proof} \rangle$

lemma $CONSTANT$: $l \in \text{list}(\text{nat}) \Longrightarrow CONSTANT(k) \text{ ' } l = k$
 $\langle \text{proof} \rangle$

lemma $PROJ-0$: $\llbracket x \in \text{nat}; l \in \text{list}(\text{nat}) \rrbracket \Longrightarrow PROJ(0) \text{ ' } (\text{Cons}(x, l)) = x$
 $\langle \text{proof} \rangle$

lemma $COMP-1$: $l \in \text{list}(\text{nat}) \Longrightarrow COMP(g, [f]) \text{ ' } l = g \text{ ' } [f^l]$
 $\langle \text{proof} \rangle$

lemma $PREC-0$: $l \in \text{list}(\text{nat}) \Longrightarrow PREC(f, g) \text{ ' } (\text{Cons}(0, l)) = f^l$
 $\langle \text{proof} \rangle$

lemma $PREC\text{-succ}$:

$\llbracket x \in \text{nat}; l \in \text{list}(\text{nat}) \rrbracket$

$\implies \text{PREC}(f,g) \text{ ' } (\text{Cons}(\text{succ}(x),l)) =$
 $g \text{ ' } \text{Cons}(\text{PREC}(f,g) \text{ ' } (\text{Cons}(x,l)), \text{Cons}(x,l))$
 $\langle \text{proof} \rangle$

13.2 Inductive definition of the PR functions

consts

prim-rec :: *i*

inductive

domains *prim-rec* $\subseteq \text{list}(\text{nat}) \rightarrow \text{nat}$

intros

SC \in *prim-rec*

k \in *nat* $\implies \text{CONSTANT}(k) \in$ *prim-rec*

i \in *nat* $\implies \text{PROJ}(i) \in$ *prim-rec*

$\llbracket g \in \text{prim-rec}; fs \in \text{list}(\text{prim-rec}) \rrbracket \implies \text{COMP}(g,fs) \in$ *prim-rec*

$\llbracket f \in \text{prim-rec}; g \in \text{prim-rec} \rrbracket \implies \text{PREC}(f,g) \in$ *prim-rec*

monos *list-mono*

con-defs *SC-def* *CONSTANT-def* *PROJ-def* *COMP-def* *PREC-def*

type-intros *nat-typechecks* *list.intros*

lam-type *list-case-type* *drop-type* *map-type*

apply-type *rec-type*

lemma *prim-rec-into-fun* [TC]: *c* \in *prim-rec* $\implies c \in \text{list}(\text{nat}) \rightarrow \text{nat}$
 $\langle \text{proof} \rangle$

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* $\implies \text{ACK}(i) \in$ *prim-rec*
 $\langle \text{proof} \rangle$

lemma *ack-type* [TC]: $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(i,j) \in \text{nat}$
 $\langle \text{proof} \rangle$

13.3 Ackermann's function cases

lemma *ack-0*: *j* \in *nat* $\implies \text{ack}(0,j) = \text{succ}(j)$
 — PROPERTY A 1
 $\langle \text{proof} \rangle$

lemma *ack-succ-0*: $\text{ack}(\text{succ}(i), 0) = \text{ack}(i,1)$
 — PROPERTY A 2
 $\langle \text{proof} \rangle$

lemma *ack-succ-succ*:

$\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(\text{succ}(i), \text{succ}(j)) = \text{ack}(i, \text{ack}(\text{succ}(i), j))$
 — PROPERTY A 3
 $\langle \text{proof} \rangle$

lemmas $[\text{simp}] = \text{ack-0 ack-succ-0 ack-succ-succ ack-type}$
and $[\text{simp del}] = \text{ACK.simps}$

lemma $\text{lt-ack2}: i \in \text{nat} \implies j \in \text{nat} \implies j < \text{ack}(i, j)$
 — PROPERTY A 4
 $\langle \text{proof} \rangle$

lemma $\text{ack-lt-ack-succ2}: \llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(i, j) < \text{ack}(i, \text{succ}(j))$
 — PROPERTY A 5-, the single-step lemma
 $\langle \text{proof} \rangle$

lemma $\text{ack-lt-mono2}: \llbracket j < k; i \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, j) < \text{ack}(i, k)$
 — PROPERTY A 5, monotonicity for <
 $\langle \text{proof} \rangle$

lemma $\text{ack-le-mono2}: \llbracket j \leq k; i \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, j) \leq \text{ack}(i, k)$
 — PROPERTY A 5', monotonicity for \leq
 $\langle \text{proof} \rangle$

lemma $\text{ack2-le-ack1}: \llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(i, \text{succ}(j)) \leq \text{ack}(\text{succ}(i), j)$
 — PROPERTY A 6
 $\langle \text{proof} \rangle$

lemma $\text{ack-lt-ack-succ1}: \llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(i, j) < \text{ack}(\text{succ}(i), j)$
 — PROPERTY A 7-, the single-step lemma
 $\langle \text{proof} \rangle$

lemma $\text{ack-lt-mono1}: \llbracket i < j; j \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, k) < \text{ack}(j, k)$
 — PROPERTY A 7, monotonicity for <
 $\langle \text{proof} \rangle$

lemma $\text{ack-le-mono1}: \llbracket i \leq j; j \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, k) \leq \text{ack}(j, k)$
 — PROPERTY A 7', monotonicity for \leq
 $\langle \text{proof} \rangle$

lemma $\text{ack-1}: j \in \text{nat} \implies \text{ack}(1, j) = \text{succ}(\text{succ}(j))$
 — PROPERTY A 8
 $\langle \text{proof} \rangle$

lemma $\text{ack-2}: j \in \text{nat} \implies \text{ack}(\text{succ}(1), j) = \text{succ}(\text{succ}(\text{succ}(j\# + j)))$
 — PROPERTY A 9
 $\langle \text{proof} \rangle$

lemma *ack-nest-bound*:

$\llbracket i1 \in \text{nat}; i2 \in \text{nat}; j \in \text{nat} \rrbracket$
 $\implies \text{ack}(i1, \text{ack}(i2, j)) < \text{ack}(\text{succ}(\text{succ}(i1 \# + i2)), j)$
 — PROPERTY A 10
 $\langle \text{proof} \rangle$

lemma *ack-add-bound*:

$\llbracket i1 \in \text{nat}; i2 \in \text{nat}; j \in \text{nat} \rrbracket$
 $\implies \text{ack}(i1, j) \# + \text{ack}(i2, j) < \text{ack}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(i1 \# + i2))))), j)$
 — PROPERTY A 11
 $\langle \text{proof} \rangle$

lemma *ack-add-bound2*:

$\llbracket i < \text{ack}(k, j); j \in \text{nat}; k \in \text{nat} \rrbracket$
 $\implies i \# + j < \text{ack}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(k))))), j)$
 — PROPERTY A 12.
 — Article uses existential quantifier but the ALF proof used $k \# + \#4$.
 — Quantified version must be nested $\exists k'. \forall i, j \dots$
 $\langle \text{proof} \rangle$

13.4 Main result

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

lemma *SC-case*: $l \in \text{list}(\text{nat}) \implies \text{SC} \text{ ' } l < \text{ack}(1, \text{list-add}(l))$
 $\langle \text{proof} \rangle$

lemma *lt-ack1*: $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies i < \text{ack}(i, j)$
 — PROPERTY A 4'? Extra lemma needed for *CONSTANT* case, constant functions.
 $\langle \text{proof} \rangle$

lemma *CONSTANT-case*:

$\llbracket l \in \text{list}(\text{nat}); k \in \text{nat} \rrbracket \implies \text{CONSTANT}(k) \text{ ' } l < \text{ack}(k, \text{list-add}(l))$
 $\langle \text{proof} \rangle$

lemma *PROJ-case* [*rule-format*]:

$l \in \text{list}(\text{nat}) \implies \forall i \in \text{nat}. \text{PROJ}(i) \text{ ' } l < \text{ack}(0, \text{list-add}(l))$
 $\langle \text{proof} \rangle$

COMP case.

lemma *COMP-map-lemma*:

$fs \in \text{list}(\{f \in \text{prim-rec}. \exists kf \in \text{nat}. \forall l \in \text{list}(\text{nat}). f^l < \text{ack}(kf, \text{list-add}(l))\})$
 $\implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}).$
 $\text{list-add}(\text{map}(\lambda f. f \text{ ' } l, fs)) < \text{ack}(k, \text{list-add}(l))$
 $\langle \text{proof} \rangle$

lemma *COMP-case*:

$\llbracket kg \in \text{nat};$

$$\begin{aligned}
& \forall l \in \text{list}(\text{nat}). g'l < \text{ack}(kg, \text{list-add}(l)); \\
& fs \in \text{list}(\{f \in \text{prim-rec} . \\
& \quad \exists kf \in \text{nat}. \forall l \in \text{list}(\text{nat}). \\
& \quad \quad f'l < \text{ack}(kf, \text{list-add}(l))\}) \\
& \implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}). \text{COMP}(g, fs)'l < \text{ack}(k, \text{list-add}(l)) \\
& \langle \text{proof} \rangle
\end{aligned}$$

PREC case.

lemma *PREC-case-lemma*:

$$\begin{aligned}
& \llbracket \forall l \in \text{list}(\text{nat}). f'l \# + \text{list-add}(l) < \text{ack}(kf, \text{list-add}(l)); \\
& \quad \forall l \in \text{list}(\text{nat}). g'l \# + \text{list-add}(l) < \text{ack}(kg, \text{list-add}(l)); \\
& \quad f \in \text{prim-rec}; \quad kf \in \text{nat}; \\
& \quad g \in \text{prim-rec}; \quad kg \in \text{nat}; \\
& \quad l \in \text{list}(\text{nat}) \rrbracket \\
& \implies \text{PREC}(f, g)'l \# + \text{list-add}(l) < \text{ack}(\text{succ}(kf \# + kg), \text{list-add}(l)) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *PREC-case*:

$$\begin{aligned}
& \llbracket f \in \text{prim-rec}; \quad kf \in \text{nat}; \\
& \quad g \in \text{prim-rec}; \quad kg \in \text{nat}; \\
& \quad \forall l \in \text{list}(\text{nat}). f'l < \text{ack}(kf, \text{list-add}(l)); \\
& \quad \forall l \in \text{list}(\text{nat}). g'l < \text{ack}(kg, \text{list-add}(l)) \rrbracket \\
& \implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}). \text{PREC}(f, g)'l < \text{ack}(k, \text{list-add}(l)) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *ack-bounds-prim-rec*:

$$\begin{aligned}
& f \in \text{prim-rec} \implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}). f'l < \text{ack}(k, \text{list-add}(l)) \\
& \langle \text{proof} \rangle
\end{aligned}$$

theorem *ack-not-prim-rec*:

$$\begin{aligned}
& (\lambda l \in \text{list}(\text{nat}). \text{list-case}(0, \lambda x xs. \text{ack}(x, x), l)) \notin \text{prim-rec} \\
& \langle \text{proof} \rangle
\end{aligned}$$

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.