

Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

June 8, 2008

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	17
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	42
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	43
10.4.1	Weakening, left and right	43
10.4.2	The deduction theorem	43
10.4.3	The cut rule	44
10.4.4	Soundness of the rules wrt truth-table semantics . . .	44
10.5	Completeness	44
10.5.1	Towards the completeness proof	44
10.5.2	Completeness – lemmas for reducing the set of as- sumptions	45
10.5.3	Completeness theorem	45
11	Lists of n elements	46
12	Combinatory Logic example: the Church-Rosser Theorem	47
12.1	Definitions	47
12.2	Transitive closure preserves the Church-Rosser property . . .	48
12.3	Results about Contraction	48
12.4	Non-contraction results	49
12.5	Results about Parallel Contraction	50
12.6	Basic properties of parallel contraction	50
13	Primitive Recursive Functions: the inductive definition	51
13.1	Basic definitions	51
13.2	Inductive definition of the PR functions	52
13.3	Ackermann's function cases	53
13.4	Main result	55

1 Sample datatype definitions

theory *Datatypes* **imports** *Main* **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 $data-mono$: $[| A \subseteq C; B \subseteq D |] \Rightarrow data(A, B) \subseteq data(C, D)$
 $\langle proof \rangle$

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

lemma $data-subset-univ$:
 $[| A \subseteq univ(C); B \subseteq univ(C) |] \Rightarrow data(A, B) \subseteq univ(C)$
 $\langle proof \rangle$

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** *Main* **begin**

2.1 Datatype definition

consts

$bt :: i \Rightarrow i$

datatype $bt(A) =$

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

declare $bt.intros$ $[simp]$

lemma $Br-neq-left$: $l \in bt(A) \Rightarrow Br(x, l, r) \neq l$

$\langle proof \rangle$

lemma $Br-iff$: $Br(a, l, r) = Br(a', l', r') \Leftrightarrow a = a' \ \& \ l = l' \ \& \ r = r'$

— 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 $bt-mono$: $A \subseteq B \Rightarrow bt(A) \subseteq bt(B)$

$\langle proof \rangle$

lemma $bt-univ$: $bt(univ(A)) \subseteq univ(A)$

$\langle proof \rangle$

lemma $bt-subset-univ$: $A \subseteq univ(B) \Rightarrow bt(A) \subseteq univ(B)$

$\langle proof \rangle$

lemma $bt-rec-type$:

$[\mid t \in bt(A);$

$c \in C(Lf);$

$!!x \ y \ z \ r \ s. [\mid x \in A; \ y \in bt(A); \ z \in bt(A); \ r \in C(y); \ s \in C(z) \mid] \Rightarrow$

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

$\mid] \Rightarrow bt-rec(c, h, t) \in C(t)$

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

$\langle proof \rangle$

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

consts $n-nodes :: i \Rightarrow i$

primrec

$n-nodes(Lf) = 0$

$n-nodes(Br(a, l, r)) = succ(n-nodes(l) \#+ n-nodes(r))$

lemma $n-nodes-type$ $[simp]$: $t \in bt(A) \Rightarrow n-nodes(t) \in nat$

$\langle proof \rangle$

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

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

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

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

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

2.4 Reflecting trees

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

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

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

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

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

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

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

end

3 Terms over an alphabet

theory *Term* **imports** *Main* **begin**

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

consts

term :: $i \Rightarrow i$

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) ==

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

definition

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

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

definition

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

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

definition

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

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

definition

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

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

definition

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

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

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

<proof>

lemma *term-induct2*:

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

$!!x. [| x \in A |] \Rightarrow P(\text{Apply}(x, \text{Nil}))];$

$$\begin{aligned} & !!x \ z \ zs. \ [\ x \in A; \ z \in \text{term}(A); \ zs: \text{list}(\text{term}(A)); \ P(\text{Apply}(x, zs)) \\ & \quad \] \implies P(\text{Apply}(x, \text{Cons}(z, zs))) \\ & \] \implies P(t) \\ & \text{— Induction on } \text{term}(A) \text{ followed by induction on } \text{list}. \\ & \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} & [\ t \in \text{term}(A); \\ & \quad !!x \ zs. \ [\ x \in A; \ zs: \text{list}(\text{term}(A)); \ \text{map}(f, zs) = \text{map}(g, zs) \] \implies \\ & \quad \quad f(\text{Apply}(x, zs)) = g(\text{Apply}(x, zs)) \\ & \] \implies f(t) = g(t) \\ & \text{— Induction on } \text{term}(A) \text{ to prove an equation.} \\ & \langle \text{proof} \rangle \end{aligned}$$

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*: $[\ t \in \text{term}(A); \ A \subseteq \text{univ}(B) \] \implies t \in \text{univ}(B)$
 $\langle \text{proof} \rangle$

term-rec – by *Vset* recursion.

lemma *map-lemma*: $[\ l \in \text{list}(A); \ \text{Ord}(i); \ \text{rank}(l) < i \]$

$$\implies \text{map}(\lambda z. (\lambda x \in \text{Vset}(i). h(x)) \ 'z, l) = \text{map}(h, l)$$

 — *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 $a: !!x \ zs \ r. \ [\ x \in A; \ zs: \text{list}(\text{term}(A));$

$$r \in \text{list}(\bigcup t \in \text{term}(A). C(t)) \]$$

$$\implies d(x, zs, r): C(\text{Apply}(x, zs))$$

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*:

$[!t. j(t) == \text{term-rec}(t, d); \quad ts: \text{list}(A) \] ==>$
 $j(\text{Apply}(a, ts)) = d(a, ts, \text{map}(\lambda Z. j(Z), ts))$
 $\langle \text{proof} \rangle$

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

$[!t \in \text{term}(A);$
 $\quad !!x \text{ } zs \text{ } r. \ [!x \in A; \quad zs: \text{list}(\text{term}(A)); \quad r \in \text{list}(C) \]$
 $\quad ==> \ d(x, zs, r): C$
 $\] ==> \ \text{term-rec}(t, d) \in C$
 $\langle \text{proof} \rangle$

term-map.

lemma *term-map* [simp]:

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

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

$[!t \in \text{term}(A); \quad !!x. \ x \in A ==> f(x): B \] ==> \ \text{term-map}(f, t) \in \text{term}(B)$
 $\langle \text{proof} \rangle$

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

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

term-size.

lemma *term-size* [simp]:

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

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

$\langle \text{proof} \rangle$

reflect.

lemma *reflect* [simp]:

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

lemma *reflect-type* [TC]: $t \in \text{term}(A) ==> \text{reflect}(t) \in \text{term}(A)$

$\langle \text{proof} \rangle$

preorder.

lemma *preorder* [simp]:

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

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

postorder.

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

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

Theorems about *term-map*.

declare *map-compose* [simp]

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

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

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

Theorems about *term-size*.

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

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

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

Theorems about *reflect*.

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

Theorems about *preorder*.

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

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

```

     $t \in \text{term}(A) \implies \text{preorder}(\text{reflect}(t)) = \text{rev}(\text{postorder}(t))$ 
    <proof>

end

```

4 Datatype definition n-ary branching trees

theory *Ntree* **imports** *Main* **begin**

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

consts

```

    ntree ::  $i \Rightarrow i$ 
    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* ($a \in A, h \in \text{maptree}(A) \multimap \text{maptree}(A)$)
monos *FiniteFun-mono1* — Use monotonicity in BOTH args
type-intros *FiniteFun-univ1* [*THEN subsetD*]

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

```

definition

```

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

```

ntree

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

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

```

    assumes t:  $t \in \text{ntree}(A)$ 
    and step:  $!!x \ n \ h. [| \ x \in A; \ n \in \text{nat}; \ h \in n \rightarrow \text{ntree}(A); \ \forall i \in n. \ P(h'i) |] \implies P(\text{Branch}(x, h))$ 
    shows  $P(t)$ 

```

— A nicer induction rule than the standard one.
 $\langle proof \rangle$

lemma *ntree-induct-eqn* [*consumes 1*]:
assumes $t: t \in ntree(A)$
and $f: f \in ntree(A) \rightarrow B$
and $g: g \in ntree(A) \rightarrow B$
and *step*: $!!x \ n \ h. [| x \in A; \ n \in nat; \ h \in n \rightarrow ntree(A); \ f \ O \ h = g \ O \ h |]$
 $==>$
 $f \ ` \ Branch(x,h) = g \ ` \ Branch(x,h)$
shows $f \ ` \ t = g \ ` \ t$
— Induction on *ntree*(*A*) to prove an equation
 $\langle proof \rangle$

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

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

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

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

ntree recursion.

lemma *ntree-rec-Branch*:
 $function(h) ==>$
 $ntree-rec(b, Branch(x,h)) = b(x, h, \lambda i \in domain(h). ntree-rec(b, h \ ` \ i))$
 $\langle proof \rangle$

lemma *ntree-copy-Branch* [*simp*]:
 $function(h) ==>$
 $ntree-copy \ (Branch(x, h)) = Branch(x, \lambda i \in domain(h). ntree-copy \ (h \ ` \ i))$
 $\langle proof \rangle$

lemma *ntree-copy-is-ident*: $z \in ntree(A) ==> ntree-copy(z) = z$
 $\langle proof \rangle$

maptree

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

lemma *maptree-induct* [*consumes 1*, *induct set*: *maptree*]:
assumes $t: t \in maptree(A)$
and *step*: $!!x \ n \ h. [| x \in A; \ h \in maptree(A) \multimap maptree(A);$
 $\forall y \in field(h). P(y)$

```

      [] ==> P(Sons(x,h))
shows P(t)
  — A nicer induction rule than the standard one.
  ⟨proof⟩

maptree2

lemma maptree2-unfold: maptree2(A, B) = A × (B -||> maptree2(A, B))
  ⟨proof⟩

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

end

```

5 Trees and forests, a mutually recursive type definition

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

5.1 Datatype definition

```

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

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

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

declare tree-forest.intros [simp, TC]

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

lemma forest-def: forest(A) == Part(tree-forest(A), Inr)

```

$\langle proof \rangle$

$tree\text{-}forest(A)$ as the union of $tree(A)$ and $forest(A)$.

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

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

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

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

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

lemma
notes $rews = tree\text{-}forest.con\text{-}defs\ tree\text{-}def\ forest\text{-}def$
shows
tree-forest-unfold: $tree\text{-}forest(A) =$
 $(A \times forest(A)) + (\{0\} + tree(A) \times forest(A))$
— NOT useful, but interesting ...
 $\langle proof \rangle$

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

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*:
 $\llbracket z \in tree\text{-}forest(A);$
 $\quad !!x\ f\ r. \llbracket x \in A; f \in forest(A); r \in C(f)$
 $\quad \llbracket \implies b(x,f,r) \in C(Tcons(x,f));$
 $\quad c \in C(Fnil);$
 $\quad !!t\ f\ r1\ r2. \llbracket t \in tree(A); f \in forest(A); r1 \in C(t); r2 \in C(f)$
 $\quad \llbracket \implies d(t,f,r1,r2) \in C(Fcons(t,f))$
 $\quad \llbracket \implies tree\text{-}forest\text{-}rec(b,c,d,z) \in C(z)$
 $\langle proof \rangle$

lemma *tree-forest-rec-type*:

$\llbracket \rrbracket !!x f r. \llbracket \rrbracket x \in A; f \in \text{forest}(A); r \in D(f)$
 $\llbracket \rrbracket ==> b(x,f,r) \in C(Tcons(x,f));$
 $c \in D(Fnil);$
 $!!t f r1 r2. \llbracket \rrbracket t \in \text{tree}(A); f \in \text{forest}(A); r1 \in C(t); r2 \in D(f)$
 $\llbracket \rrbracket ==> d(t,f,r1,r2) \in D(Fcons(t,f))$
 $\llbracket \rrbracket ==> (\forall t \in \text{tree}(A). \text{tree-forest-rec}(b,c,d,t) \in C(t)) \wedge$
 $(\forall f \in \text{forest}(A). \text{tree-forest-rec}(b,c,d,f) \in D(f))$
 — 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

$\text{reflect } (Tcons(x,f)) = Tcons(x, \text{reflect}(f))$
 $\text{reflect } (Fnil) = Fnil$

$reflect\ (Fcons(t,tf)) =$
 $of-list\ (list-of-TF\ (reflect(tf))\ @\ Cons(reflect(t), Nil))$

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

lemma *list-of-TF-type* [TC]:

$z \in tree-forest(A) ==> list-of-TF(z) \in list(tree(A))$
 $\langle proof \rangle$

lemma *of-list-type* [TC]: $l \in list(tree(A)) ==> of-list(l) \in forest(A)$

$\langle proof \rangle$

map.

lemma

assumes $!!x. x \in A ==> h(x): B$

shows *map-tree-type*: $t \in tree(A) ==> map(h,t) \in tree(B)$

and *map-forest-type*: $f \in forest(A) ==> map(h,f) \in forest(B)$

$\langle proof \rangle$

size.

lemma *size-type* [TC]: $z \in tree-forest(A) ==> size(z) \in nat$

$\langle proof \rangle$

preorder.

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

$\langle proof \rangle$

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

lemma *forest-induct* [consumes 1, case-names Fnil Fcons]:

$[[f \in forest(A);$

$R(Fnil);$

$!!t f. [[t \in tree(A); f \in forest(A); R(f)]] ==> R(Fcons(t,f))$

$]] ==> R(f)$

— Essentially the same as list induction.

$\langle proof \rangle$

lemma *forest-iso*: $f \in forest(A) ==> of-list(list-of-TF(f)) = f$

$\langle proof \rangle$

lemma *tree-list-iso*: $ts: list(tree(A)) ==> list-of-TF(of-list(ts)) = ts$

$\langle proof \rangle$

Theorems about *map*.

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

$\langle proof \rangle$

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

Theorems about *preorder*.

lemma *preorder-map*:

$z \in \text{tree-forest}(A) \implies \text{preorder}(\text{map}(h, z)) = \text{List-ZF.map}(h, \text{preorder}(z))$
 $\langle \text{proof} \rangle$

end

6 Infinite branching datatype definitions

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

6.1 The Brouwer ordinals

consts

brouwer :: *i*

datatype $\subseteq V_{\text{from}}(0, \text{csucc}(\text{nat}))$

brouwer = *Zero* | *Suc* ($b \in \text{brouwer}$) | *Lim* ($h \in \text{nat} \rightarrow \text{brouwer}$)

monos *Pi-mono*

type-intros *inf-datatype-intros*

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

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

assumes $b \in \text{brouwer}$

and cases:

$P(\text{Zero})$

$!!b. [\![\ b \in \text{brouwer};\ P(b)\]\!] \implies P(\text{Suc}(b))$

$!!h. [\![\ h \in \text{nat} \rightarrow \text{brouwer};\ \forall i \in \text{nat}. P(h^i)\]\!] \implies P(\text{Lim}(h))$

shows $P(b)$

— A nicer induction rule than the standard one.

$\langle \text{proof} \rangle$

6.2 The Martin-Löf wellordering type

consts

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

datatype $\subseteq Vfrom(A \cup (\bigcup x \in A. B(x)), csucc(nat \cup |\bigcup x \in A. B(x)|))$

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

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

monos *Pi-mono*

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

lemma *Well-unfold*: $Well(A, B) = (\Sigma\ x \in A. B(x) \rightarrow Well(A, B))$

<proof>

lemma *Well-induct2* [*consumes 1, case-names step*]:

assumes *w*: $w \in Well(A, B)$

and step: $!!a\ f. [a \in A; f \in B(a) \rightarrow Well(A, B); \forall y \in B(a). P(f\ y)]$

$\Rightarrow P(Sup(a, f))$

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** *Main* **begin**

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

consts

domino :: *i*

tiling :: *i* $\Rightarrow i$

inductive

domains *domino* $\subseteq Pow(nat \times nat)$

intros

horiz: $[i \in nat; j \in nat] \Rightarrow \{<i, j>, <i, succ(j)>\} \in domino$

vertl: $[i \in nat; j \in nat] \Rightarrow \{<i, j>, <succ(i), j>\} \in domino$

type-intros *empty-subsetI cons-subsetI PowI SigmaI nat-succI*

inductive**domains** $tiling(A) \subseteq Pow(Union(A))$ **intros***empty*: $0 \in tiling(A)$ *Un*: $[| a \in A; t \in tiling(A); a \text{ Int } t = 0 |] ==> a \text{ Un } t \in tiling(A)$ **type-intros** *empty-subsetI Union-upper Un-least PowI***type-elim** *PowD [elim-format]***definition***evnodd* :: $[i, i] ==> i$ **where***evnodd*(A, b) == $\{z \in A. \exists i j. z = \langle i, j \rangle \wedge (i \# + j) \bmod 2 = b\}$ **7.1 Basic properties of evnodd****lemma** *evnodd-iff*: $\langle i, j \rangle: evnodd(A, b) \leftrightarrow \langle i, j \rangle: A \ \& \ (i \# + j) \bmod 2 = b$
*<proof>***lemma** *evnodd-subset*: $evnodd(A, b) \subseteq A$
*<proof>***lemma** *Finite-evnodd*: $Finite(X) ==> Finite(evnodd(X, b))$
*<proof>***lemma** *evnodd-Un*: $evnodd(A \text{ Un } B, b) = evnodd(A, b) \text{ Un } evnodd(B, b)$
*<proof>***lemma** *evnodd-Diff*: $evnodd(A - B, b) = evnodd(A, b) - evnodd(B, b)$
*<proof>***lemma** *evnodd-cons* [*simp*]:
 $evnodd(cons(\langle i, j \rangle, C), b) =$
 $(\text{if } (i \# + j) \bmod 2 = b \text{ then } cons(\langle i, j \rangle, evnodd(C, b)) \text{ else } evnodd(C, b))$
*<proof>***lemma** *evnodd-0* [*simp*]: $evnodd(0, b) = 0$
*<proof>***7.2 Dominoes****lemma** *domino-Finite*: $d \in domino ==> Finite(d)$
*<proof>***lemma** *domino-singleton*:
 $[| d \in domino; b < 2 |] ==> \exists i' j'. evnodd(d, b) = \{\langle i', j' \rangle\}$
*<proof>***7.3 Tilings**

The union of two disjoint tilings is a tiling

lemma *tiling-UnI*:

$t \in \text{tiling}(A) \implies u \in \text{tiling}(A) \implies t \text{ Int } u = 0 \implies t \text{ Un } u \in \text{tiling}(A)$
 $\langle \text{proof} \rangle$

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*:

$[i \in \text{nat}; n \in \text{nat}] \implies \{i\} * (n \# + n) \in \text{tiling}(\text{domino})$

$\langle \text{proof} \rangle$

lemma *dominoes-tile-matrix*:

$[m \in \text{nat}; n \in \text{nat}] \implies m * (n \# + n) \in \text{tiling}(\text{domino})$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

$t' = t - \{<0, 0>\} - \{<\text{succ}(m \# + m), \text{succ}(n \# + n)>\}]$

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

$\langle \text{proof} \rangle$

end

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

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

inductive

domains *fold-set*(*A*, *B*, *f*, *e*) <= *Fin*(*A*)**B*

intros

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

consI: $[x \in A; x \notin C; <C, y> : \text{fold-set}(A, B, f, e); f(x, y) : B]$

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

type-intros *Fin.intros*

definition

fold :: $[i, [i, i] \Rightarrow i, i, i] \Rightarrow i$ (*fold*[-]'(-, -, -)) **where**

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

definition

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

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

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

lemma *cons-lemma1*: [| *x* ∉ *C*; *x* ∉ *B* |] ==> *cons*(*x*, *B*) = *cons*(*x*, *C*) <-> *B* = *C*
 <proof>

lemma *cons-lemma2*: [| *cons*(*x*, *B*) = *cons*(*y*, *C*); *x* ≠ *y*; *x* ∉ *B*; *y* ∉ *C* |]
 ==> *B* - {*y*} = *C* - {*x*} & *x* ∈ *C* & *y* ∈ *B*
 <proof>

lemma *fold-set-mono-lemma*:
 <*C*, *x*> : *fold-set*(*A*, *B*, *f*, *e*)
 ==> ALL *D. A* <= *D* --> <*C*, *x*> : *fold-set*(*D*, *B*, *f*, *e*)
 <proof>

lemma *fold-set-mono*: *C* <= *A* ==> *fold-set*(*C*, *B*, *f*, *e*) <= *fold-set*(*A*, *B*, *f*, *e*)
 <proof>

lemma *fold-set-lemma*:
 <*C*, *x*> ∈ *fold-set*(*A*, *B*, *f*, *e*) ==> <*C*, *x*> ∈ *fold-set*(*C*, *B*, *f*, *e*) & *C* <= *A*
 <proof>

lemma *Diff1-fold-set*:
 [| <*C* - {*x*}, *y*> : *fold-set*(*A*, *B*, *f*, *e*); *x* ∈ *C*; *x* ∈ *A*; *f*(*x*, *y*): *B* |]
 ==> <*C*, *f*(*x*, *y*)> : *fold-set*(*A*, *B*, *f*, *e*)
 <proof>

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

lemma (in *fold-typing*) *Fin-imp-fold-set*:
C ∈ *Fin*(*A*) ==> (EX *x. <C*, *x*> : *fold-set*(*A*, *B*, *f*, *e*))
 <proof>

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]:
 $n \in \text{nat}$

$\implies \text{ALL } C. |C| < n \dashv\dashv$
 $(\text{ALL } x. \langle C, x \rangle : \text{fold-set}(A, B, f, e) \dashv\dashv$
 $(\text{ALL } y. \langle C, y \rangle : \text{fold-set}(A, B, f, e) \dashv\dashv y=x))$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

$\llbracket C : \text{Fin}(A); c : A; c \notin C \rrbracket$
 $\implies \langle \text{cons}(c, C), v \rangle : \text{fold-set}(\text{cons}(c, C), B, f, e) \dashv\dashv$
 $(\text{EX } y. \langle C, y \rangle : \text{fold-set}(C, B, f, e) \ \& \ v = f(c, y))$
 $\langle \text{proof} \rangle$

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

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

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]:

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

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

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

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

$$\begin{aligned} & [[C \in \text{Fin}(A); D \in \text{Fin}(A); C \text{ Int } D = 0]] \\ & \implies \text{fold}[B](f, e, C \text{ Un } D) = \text{fold}[B](f, \text{fold}[B](f, e, D), C) \end{aligned}$$
 $\langle \text{proof} \rangle$

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

7.4 The Operator *setsum*

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

lemma *setsum-cons* [*simp*]:

$$\begin{aligned} & \text{Finite}(C) \implies \\ & \quad \text{setsum}(g, \text{cons}(c, C)) = \\ & \quad (\text{if } c : C \text{ then } \text{setsum}(g, C) \text{ else } g(c) \$+ \text{setsum}(g, C)) \end{aligned}$$
 $\langle \text{proof} \rangle$

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

lemma *setsum-Un-Int*:

$$\begin{aligned} & [[\text{Finite}(C); \text{Finite}(D)]] \\ & \implies \text{setsum}(g, C \text{ Un } D) \$+ \text{setsum}(g, C \text{ Int } D) \\ & \quad = \text{setsum}(g, C) \$+ \text{setsum}(g, D) \end{aligned}$$
 $\langle \text{proof} \rangle$

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

lemma *setsum-Un-disjoint*:

$$\begin{aligned} & [[\text{Finite}(C); \text{Finite}(D); C \text{ Int } D = 0]] \\ & \implies \text{setsum}(g, C \text{ Un } D) = \text{setsum}(g, C) \$+ \text{setsum}(g, D) \end{aligned}$$
 $\langle \text{proof} \rangle$

lemma *Finite-RepFun* [*rule-format* (*no-asm*)]:

$$\text{Finite}(I) \implies (\forall i \in I. \text{Finite}(C(i))) \longrightarrow \text{Finite}(\text{RepFun}(I, C))$$
 $\langle \text{proof} \rangle$

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

$Finite(I)$
 $==> (\forall i \in I. Finite(C(i))) \dashv\dashv$
 $(\forall i \in I. \forall j \in I. i \neq j \dashv\dashv C(i) \text{ Int } C(j) = 0) \dashv\dashv$
 $setsum(f, \bigcup i \in I. C(i)) = setsum (\%i. setsum(f, C(i)), I)$
 $\langle proof \rangle$

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

lemma *fold-set-cong*:
 $[[A=A'; B=B'; e=e'; (\forall x \in A'. \forall y \in B'. f(x,y) = f'(x,y))]]$
 $==> fold-set(A,B,f,e) = fold-set(A',B',f',e')$
 $\langle proof \rangle$

lemma *fold-cong*:
 $[[B=B'; A=A'; e=e';$
 $!!x y. [[x \in A'; y \in B']] ==> f(x,y) = f'(x,y)]] ==>$
 $fold[B](f,e,A) = fold[B'](f', e', A')$
 $\langle proof \rangle$

lemma *setsum-cong*:
 $[[A=B; !!x. x \in B ==> f(x) = g(x)]] ==>$
 $setsum(f, A) = setsum(g, B)$
 $\langle proof \rangle$

lemma *setsum-Un*:
 $[[Finite(A); Finite(B)]]$
 $==> setsum(f, A \text{ Un } B) =$
 $setsum(f, A) \$+ setsum(f, B) \$- setsum(f, A \text{ Int } B)$
 $\langle proof \rangle$

lemma *setsum-zneg-or-0* [*rule-format* (*no-asm*)]:
 $Finite(A) ==> (\forall x \in A. g(x) \$<= \#0) \dashv\dashv setsum(g, A) \$<= \#0$
 $\langle proof \rangle$

lemma *setsum-succD-lemma* [*rule-format*]:
 $Finite(A)$
 $==> \forall n \in nat. setsum(f, A) = \$\# succ(n) \dashv\dashv (\exists a \in A. \#0 \$< f(a))$
 $\langle proof \rangle$

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

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

$Finite(A) ==> (\forall x \in A. \#0 \ \$<= g(x)) \dashrightarrow \#0 \ \$<= setsum(g, A)$
 $\langle proof \rangle$

lemma *g-zpos-imp-setsum-zpos2* [rule-format]:
 $[[Finite(A); \forall x. \#0 \ \$<= g(x)]] ==> \#0 \ \$<= setsum(g, A)$
 $\langle proof \rangle$

lemma *g-zspos-imp-setsum-zspos* [rule-format]:
 $Finite(A)$
 $==> (\forall x \in A. \#0 \ \$< g(x)) \dashrightarrow A \neq 0 \dashrightarrow (\#0 \ \$< setsum(g, A))$
 $\langle proof \rangle$

lemma *setsum-Diff* [rule-format]:
 $Finite(A) ==> \forall a. M(a) = \#0 \dashrightarrow setsum(M, A) = setsum(M, A - \{a\})$
 $\langle proof \rangle$

end

8 The accessible part of a relation

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

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

consts
 $acc :: i ==> i$

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

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

The intended introduction rule:

lemma *accI*: $[[!!b. <b,a>:r ==> b \in acc(r); a \in field(r)]] ==> a \in acc(r)$
 $\langle proof \rangle$

lemma *acc-downward*: $[[b \in acc(r); <a,b>:r]] ==> a \in acc(r)$
 $\langle proof \rangle$

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

lemma *wf-on-acc*: $wf[acc(r)](r)$
 $\langle proof \rangle$

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

lemma *acc-wfD*: $wf(r) \implies field(r) \subseteq acc(r)$
 $\langle proof \rangle$

lemma *wf-acc-iff*: $wf(r) <-> field(r) \subseteq acc(r)$
 $\langle proof \rangle$

end

theory *Multiset*
imports *FoldSet Acc*
begin

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

definition

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

definition

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

definition

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

definition

$munion :: [i, i] \Rightarrow i$ (**infixl** $+\#$ 65) **where**
 $M +\# N == \lambda x \in mset-of(M) \cup mset-of(N).$
 $\quad if \ x \in mset-of(M) \ Int \ mset-of(N) \ then \ (M^i x) +\# (N^i x)$
 $\quad else \ (if \ x \in mset-of(M) \ then \ M^i x \ else \ N^i x)$

definition

$normalize :: i \Rightarrow i$ **where**
 $normalize(f) ==$

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

definition

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

definition

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

definition

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

definition

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

definition

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

abbreviation

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

syntax

$\text{@MColl} :: [\text{pttrn}, i, o] \Rightarrow i \ ((1\{\# \ - : \ -/\ -\#\})$

syntax (*xsymbols*)

$\text{@MColl} :: [\text{pttrn}, i, o] \Rightarrow i \ ((1\{\# \ - \in \ -/\ -\#\})$

translations

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

definition

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

definition

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

definition

omultiset :: *i* => *o* **where**
omultiset(*M*) == $\exists i. \text{Ord}(i) \ \& \ M \in \text{Mult}(\text{field}(\text{Memrel}(i)))$

definition

mless :: [*i*, *i*] => *o* (**infixl** <# 50) **where**
 $M <\# N == \exists i. \text{Ord}(i) \ \& \ \langle M, N \rangle \in \text{multirel}(\text{field}(\text{Memrel}(i)), \text{Memrel}(i))$

definition

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

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

lemma *funrestrict-subset*: [$f \in \text{Pi}(C, B); \ A \subseteq C$] ==> *funrestrict*(*f*, *A*) $\subseteq f$
 <proof>

lemma *funrestrict-type*:

[$!!x. x \in A ==> f'x \in B(x)$] ==> *funrestrict*(*f*, *A*) $\in \text{Pi}(A, B)$
 <proof>

lemma *funrestrict-type2*: [$f \in \text{Pi}(C, B); \ A \subseteq C$] ==> *funrestrict*(*f*, *A*) $\in \text{Pi}(A, B)$
 <proof>

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

lemma *funrestrict-empty [simp]*: *funrestrict*(*f*, 0) = 0
 <proof>

lemma *domain-funrestrict [simp]*: *domain*(*funrestrict*(*f*, *C*)) = *C*
 <proof>

lemma *fun-cons-funrestrict-eq*:

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

declare *domain-of-fun [simp]*

declare *domainE [rule del]*

A useful simplification rule

lemma *multiset-fun-iff*:

$(f \in A \rightarrow \text{nat} - \{0\}) \leftrightarrow f \in A \rightarrow \text{nat} \ \& \ (\forall a \in A. f'a \in \text{nat} \ \& \ 0 < f'a)$
 <proof>

lemma *multiset-into-Mult*: $[| \text{multiset}(M); \text{mset-of}(M) \subseteq A |] \implies M \in \text{Mult}(A)$
 $\langle \text{proof} \rangle$

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

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

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

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

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]: $\{ \#a \# \} \neq 0 \ \& \ 0 \neq \{ \#a \# \}$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *normalize-multiset [simp]: multiset(M) ==> normalize(M) = M*
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

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

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

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

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

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

lemma *mcount-union* [simp]: $[| multiset(M); multiset(N) |]$
 $==> mcount(M +\# N, a) = mcount(M, a) \#+ mcount(N, a)$
 $\langle proof \rangle$

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

lemma *mcount-elem*: $[| multiset(M); a \in mset-of(M) |] ==> 0 < mcount(M, a)$
 $\langle proof \rangle$

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

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

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

lemma *msize-zpositive*: $multiset(M) ==> \#0 \leq msize(M)$
 $\langle proof \rangle$

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

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

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

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-elem*: $\llbracket \text{msize}(M) = \$\# \text{succ}(n); n \in \text{nat} \rrbracket \implies \exists a. a \in \text{mset-of}(M)$
 $\langle \text{proof} \rangle$

lemma *equality-lemma*:
 $\llbracket \text{multiset}(M); \text{multiset}(N); \forall a. \text{mcount}(M, a) = \text{mcount}(N, a) \rrbracket$
 $\implies \text{mset-of}(M) = \text{mset-of}(N)$
 $\langle \text{proof} \rangle$

lemma *multiset-equality*:
 $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies M = N \iff (\forall a. \text{mcount}(M, a) = \text{mcount}(N, a))$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *nat-add-eq-1-cases*: $[[m \in \text{nat}; n \in \text{nat}]] \implies (m \# + n = 1) \iff (m=1 \ \& \ n=0) \mid (m=0 \ \& \ n=1)$
 $\langle \text{proof} \rangle$

lemma *munion-is-single*:
 $[[\text{multiset}(M); \text{multiset}(N)]] \implies (M \# + N = \{\#a\# \}) \iff (M=\{\#a\# \} \ \& \ N=0) \mid (M = 0 \ \& \ N = \{\#a\# \})$
 $\langle \text{proof} \rangle$

lemma *msingle-is-union*: $[[\text{multiset}(M); \text{multiset}(N)]] \implies (\{\#a\# \} = M \# + N) \iff (\{\#a\# \} = M \ \& \ N=0 \mid M = 0 \ \& \ \{\#a\# \} = N)$
 $\langle \text{proof} \rangle$

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

lemma *setsum-decr2*:
 $\text{Finite}(A) \implies \forall M. \text{multiset}(M) \implies (\forall a \in \text{mset-of}(M). \text{setsum}(\%x. \#\ \text{mcount}(\text{funrestrict}(M, \text{mset-of}(M) - \{a\}), x), A) = (\text{if } a \in A \text{ then } \text{setsum}(\%x. \#\ \text{mcount}(M, x), A) \# - \#\ M'a \text{ else } \text{setsum}(\%x. \#\ \text{mcount}(M, x), A))))$
 $\langle \text{proof} \rangle$

lemma *setsum-decr3*: $[[\text{Finite}(A); \text{multiset}(M); a \in \text{mset-of}(M)]] \implies \text{setsum}(\%x. \#\ \text{mcount}(\text{funrestrict}(M, \text{mset-of}(M) - \{a\}), x), A - \{a\}) = (\text{if } a \in A \text{ then } \text{setsum}(\%x. \#\ \text{mcount}(M, x), A) \# - \#\ M'a \text{ else } \text{setsum}(\%x. \#\ \text{mcount}(M, x), A))$
 $\langle \text{proof} \rangle$

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

lemma *succ-pred-eq-self*: $[[0 < n; n \in \text{nat}]] \implies \text{succ}(n \# - 1) = n$
 $\langle \text{proof} \rangle$

Specialized for use in the proof below.

lemma *multiset-funrestrict*:

$$\llbracket \forall a \in A. M \text{ ' } a \in \text{nat} \wedge 0 < M \text{ ' } a; \text{Finite}(A) \rrbracket$$

$$\implies \text{multiset}(\text{funrestrict}(M, A - \{a\}))$$

$$\langle \text{proof} \rangle$$

lemma *multiset-induct-aux*:

assumes *prem1*: $\llbracket M \text{ ' } a. \llbracket \text{multiset}(M); a \notin \text{mset-of}(M); P(M) \rrbracket \implies P(\text{cons}(<a, 1>, M))$
and *prem2*: $\llbracket M \text{ ' } b. \llbracket \text{multiset}(M); b \in \text{mset-of}(M); P(M) \rrbracket \implies P(M(b := M \text{ ' } b \# + 1))$
shows

$$\llbracket n \in \text{nat}; P(0) \rrbracket$$

$$\implies (\forall M. \text{multiset}(M) \dashv\dashv$$

$$(\text{setsum}(\%x. \$\# \text{mcount}(M, x), \{x \in \text{mset-of}(M). 0 < M \text{ ' } x\}) = \$\# n) \dashv\dashv$$

$$P(M))$$

$$\langle \text{proof} \rangle$$

lemma *multiset-induct2*:

$$\llbracket \text{multiset}(M); P(0);$$

$$(\llbracket M \text{ ' } a. \llbracket \text{multiset}(M); a \notin \text{mset-of}(M); P(M) \rrbracket \implies P(\text{cons}(<a, 1>, M));$$

$$(\llbracket M \text{ ' } b. \llbracket \text{multiset}(M); b \in \text{mset-of}(M); P(M) \rrbracket \implies P(M(b := M \text{ ' } b \# + 1)))$$

$$\rrbracket$$

$$\implies P(M)$$

$$\langle \text{proof} \rangle$$

lemma *munion-single-case1*:

$$\llbracket \text{multiset}(M); a \notin \text{mset-of}(M) \rrbracket \implies M + \# \{\#a\# \} = \text{cons}(<a, 1>, M)$$

$$\langle \text{proof} \rangle$$

lemma *munion-single-case2*:

$$\llbracket \text{multiset}(M); a \in \text{mset-of}(M) \rrbracket \implies M + \# \{\#a\# \} = M(a := M \text{ ' } a \# + 1)$$

$$\langle \text{proof} \rangle$$

lemma *multiset-induct*:

assumes *M*: $\text{multiset}(M)$
and *P0*: $P(0)$
and *step*: $\llbracket M \text{ ' } a. \llbracket \text{multiset}(M); P(M) \rrbracket \implies P(M + \# \{\#a\# \})$
shows $P(M)$

$$\langle \text{proof} \rangle$$

lemma *MCollect-multiset [simp]*:

$$\text{multiset}(M) \implies \text{multiset}(\{\# x \in M. P(x) \# \})$$

$$\langle \text{proof} \rangle$$

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

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

$\langle proof \rangle$

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

$$x \in mset-of(\{\#x \in M. P(x)\# \}) \leftrightarrow x \in mset-of(M) \ \& \ P(x)$$

$\langle proof \rangle$

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*: $multiset(M) ==> M = \{\#x \in M. P(x)\# \} +\# \{\#x \in M. \sim P(x)\# \}$

$\langle proof \rangle$

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

$$[\mid multiset(M); a \in mset-of(M) \mid] ==> natify(M'a) = M'a$$

$\langle proof \rangle$

lemma *munion-eq-conv-diff*: $[\mid multiset(M); multiset(N) \mid]$

$$==> (M +\# \{\#a\# \} = N +\# \{\#b\# \}) \leftrightarrow (M = N \ \& \ a = b \mid$$

$$M = N -\# \{\#a\# \} +\# \{\#b\# \} \ \& \ N = M -\# \{\#b\# \} +\# \{\#a\# \})$$

$\langle proof \rangle$

lemma *melem-diff-single*:

$$multiset(M) ==>$$

$$k \in mset-of(M -\# \{\#a\# \}) \leftrightarrow (k=a \ \& \ 1 < mcount(M, a)) \mid (k \neq a \ \& \ k \in mset-of(M))$$

$\langle proof \rangle$

lemma *munion-eq-conv-exist*:

$$[\mid M \in Mult(A); N \in Mult(A) \mid]$$

$$==> (M +\# \{\#a\# \} = N +\# \{\#b\# \}) \leftrightarrow$$

$$(M=N \ \& \ a=b \mid (\exists K \in Mult(A). M = K +\# \{\#b\# \} \ \& \ N = K +\# \{\#a\# \}))$$

$\langle proof \rangle$

8.2 Multiset Orderings

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

$\langle proof \rangle$

lemma *multirel1-0* [simp]: $multirel1(0, r) = 0$

$\langle proof \rangle$

lemma *multirel1-iff*:

$$\langle N, M \rangle \in multirel1(A, r) \leftrightarrow$$

$$(\exists a. a \in A \ \&$$

$$(\exists M0. M0 \in Mult(A) \ \& \ (\exists K. K \in Mult(A) \ \&$$

$M = M0 + \# \{ \#a\# \} \ \& \ N = M0 + \# K \ \& \ (\forall b \in \text{mset-of}(K). \langle b, a \rangle \in r))$
 $\langle \text{proof} \rangle$

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*:
 $[\mid A \subseteq B; r \subseteq s \mid] \implies \text{multirel1}(A, r) \subseteq \text{multirel1}(B, s)$
 $\langle \text{proof} \rangle$

8.3 Toward the proof of well-foundedness of *multirel1*

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

lemma *less-munion*: $[\mid \langle N, M0 + \# \{ \#a\# \} \rangle \in \text{multirel1}(A, r); M0 \in \text{Mult}(A)$
 $\mid] \implies$
 $(\exists M. \langle M, M0 \rangle \in \text{multirel1}(A, r) \ \& \ N = M + \# \{ \#a\# \}) \mid$
 $(\exists K. K \in \text{Mult}(A) \ \& \ (\forall b \in \text{mset-of}(K). \langle b, a \rangle \in r) \ \& \ N = M0 + \# K)$
 $\langle \text{proof} \rangle$

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

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

lemma *lemma1*: $[\mid \forall b \in A. \langle b, a \rangle \in r \implies$
 $(\forall M \in \text{acc}(\text{multirel1}(A, r)). M + \# \{ \#b\# \} : \text{acc}(\text{multirel1}(A, r)))$;
 $M0 \in \text{acc}(\text{multirel1}(A, r)); a \in A$;
 $\forall M. \langle M, M0 \rangle \in \text{multirel1}(A, r) \implies M + \# \{ \#a\# \} \in \text{acc}(\text{multirel1}(A, r))$
 $\mid]$
 $\implies M0 + \# \{ \#a\# \} \in \text{acc}(\text{multirel1}(A, r))$
 $\langle \text{proof} \rangle$

lemma *lemma2*: $[\mid \forall b \in A. \langle b, a \rangle \in r$
 $\implies (\forall M \in \text{acc}(\text{multirel1}(A, r)). M + \# \{ \#b\# \} : \text{acc}(\text{multirel1}(A, r)))$;
 $M \in \text{acc}(\text{multirel1}(A, r)); a \in A \mid] \implies M + \# \{ \#a\# \} \in \text{acc}(\text{multirel1}(A,$
 $r))$
 $\langle \text{proof} \rangle$

lemma *lemma3*: $[\mid \text{wf}[A](r); a \in A \mid]$
 $\implies \forall M \in \text{acc}(\text{multirel1}(A, r)). M + \# \{ \#a\# \} \in \text{acc}(\text{multirel1}(A, r))$
 $\langle \text{proof} \rangle$

lemma *lemma4*: $\text{multiset}(M) \implies \text{mset-of}(M) \subseteq A \dashv\dashv$
 $\text{wf}[A](r) \dashv\dashv M \in \text{field}(\text{multirel1}(A, r)) \dashv\dashv M \in \text{acc}(\text{multirel1}(A, r))$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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*:
 $[| \langle M, N \rangle \in \text{multirel}(A, r); \langle N, K \rangle \in \text{multirel}(A, r) |] \implies \langle M, K \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

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*:
 $[\langle M, N \rangle \in \text{multirel1}(A, r); K \in \text{Mult}(A)] \implies \langle K +\# M, K +\# N \rangle \in \text{multirel1}(A, r)$
 $\langle \text{proof} \rangle$

lemma *munion-multirel-mono2*:
 $[\langle M, N \rangle \in \text{multirel}(A, r); K \in \text{Mult}(A)] \implies \langle K +\# M, K +\# N \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

lemma *munion-multirel-mono1*:
 $[\langle M, N \rangle \in \text{multirel}(A, r); K \in \text{Mult}(A)] \implies \langle M +\# K, N +\# K \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

lemma *munion-multirel-mono*:
 $[\langle M, K \rangle \in \text{multirel}(A, r); \langle N, L \rangle \in \text{multirel}(A, r)] \implies \langle M +\# N, K +\# L \rangle \in \text{multirel}(A, r)$
 $\langle \text{proof} \rangle$

8.4 Ordinal Multisets

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

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]: $[\text{omultiset}(M); \text{omultiset}(N)] \implies \text{omultiset}(M +\# N)$
 $\langle \text{proof} \rangle$

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*: $\text{trans}(r) \iff \text{trans}[\text{field}(r)](r)$

$\langle proof \rangle$

lemma *part-ord-Memrel*: $Ord(i) \implies part-ord(field(Memrel(i)), Memrel(i))$
 $\langle proof \rangle$

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

lemma *mless-not-refl*: $\sim(M <\# M)$
 $\langle proof \rangle$

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

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

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

lemma *mless-asy*: $[M <\# N; \sim P \implies N <\# M] \implies P$
 $\langle proof \rangle$

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

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

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

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

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

lemma *munion-less-mono1*: $[| M <\# N; \text{omultiset}(K) |] \implies M +\# K <\# N +\# K$
 $\langle \text{proof} \rangle$

lemma *mless-imp-omultiset*: $M <\# N \implies \text{omultiset}(M) \ \& \ \text{omultiset}(N)$
 $\langle \text{proof} \rangle$

lemma *munion-less-mono*: $[| M <\# K; N <\# L |] \implies M +\# N <\# K +\# L$
 $\langle \text{proof} \rangle$

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

lemma *mle-mono*: $[| M <\# = K; N <\# = L |] \implies M +\# N <\# = K +\# L$
 $\langle \text{proof} \rangle$

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

lemma *empty-leI* [simp]: $\text{omultiset}(M) \implies 0 <\# = M$
 $\langle \text{proof} \rangle$

lemma *munion-upper1*: $[| \text{omultiset}(M); \text{omultiset}(N) |] \implies M <\# = M +\# N$
 $\langle \text{proof} \rangle$

end

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

theory *Rmap* **imports** *Main* **begin**

consts

rmap :: $i \implies i$

inductive

domains $rmap(r) \subseteq \text{list}(\text{domain}(r)) \times \text{list}(\text{range}(r))$

intros

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

ConsI: $[| \langle x, y \rangle: r; \langle xs, ys \rangle \in rmap(r) |] \implies \langle Cons(x, xs), Cons(y, ys) \rangle \in rmap(r)$

type-intros *domainI rangeI list.intros*

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

inductive-cases

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

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

declare *rmap.intros* [intro]

lemma *rmap-rel-type*: $r \subseteq A \times B \implies rmap(r) \subseteq list(A) \times list(B)$
 $\langle proof \rangle$

lemma *rmap-total*: $A \subseteq domain(r) \implies list(A) \subseteq domain(rmap(r))$
 $\langle proof \rangle$

lemma *rmap-functional*: $function(r) \implies function(rmap(r))$
 $\langle proof \rangle$

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

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

lemma *rmap-Nil*: $rmap(f)'Nil = Nil$
 $\langle proof \rangle$

lemma *rmap-Cons*: $[f \in A \multimap B; x \in A; xs: list(A)]$
 $\implies rmap(f)'Cons(x, xs) = Cons(f'x, rmap(f)'xs)$
 $\langle proof \rangle$

end

10 Meta-theory of propositional logic

theory *PropLog* **imports** *Main* **begin**

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

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

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

10.1 The datatype of propositions

consts

propn :: i

datatype *propn* =

FIs

| *Var* ($n \in \text{nat}$) ($\#$ - [100] 100)
| *Imp* ($p \in \text{propn}$, $q \in \text{propn}$) (**infixr** \Rightarrow 90)

10.2 The proof system

consts *thms* :: $i \Rightarrow i$
syntax *-thms* :: $[i, i] \Rightarrow o$ (**infixl** \vdash 50)
translations $H \vdash p == p \in \text{thms}(H)$

inductive

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

intros

H : $\llbracket p \in H; p \in \text{propn} \rrbracket \Rightarrow H \vdash p$
 K : $\llbracket p \in \text{propn}; q \in \text{propn} \rrbracket \Rightarrow H \vdash p \Rightarrow q \Rightarrow p$
 S : $\llbracket p \in \text{propn}; q \in \text{propn}; r \in \text{propn} \rrbracket$
 $\Rightarrow H \vdash (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$
 DN : $p \in \text{propn} \Rightarrow H \vdash ((p \Rightarrow \text{Fls}) \Rightarrow \text{Fls}) \Rightarrow p$
 MP : $\llbracket H \vdash p \Rightarrow q; H \vdash p; p \in \text{propn}; q \in \text{propn} \rrbracket \Rightarrow H \vdash q$
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) == *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) \leftrightarrow *False*
 $\langle \text{proof} \rangle$

lemma *is-true-Var* [*simp*]: *is-true*($\#v$, t) \leftrightarrow $v \in t$
 $\langle \text{proof} \rangle$

lemma *is-true-Imp* [*simp*]: *is-true*($p \Rightarrow q$, t) \leftrightarrow (*is-true*(p , t) \rightarrow *is-true*(q , t))
 $\langle \text{proof} \rangle$

10.3.2 Logical consequence

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

definition

$logcon :: [i,i] ==> o$ (**infixl** $|= 50$) **where**
 $H \models p == \forall t. (\forall q \in H. is_true(q,t)) \dashv\dashv is_true(p,t)$

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

consts

$hyps :: [i,i] ==> i$

primrec

$hyps(Fls, t) = 0$
 $hyps(Var(v), t) = (if\ v \in t\ then\ \{\#v\}\ else\ \{\#v ==> Fls\})$
 $hyps(p ==> q, t) = hyps(p,t) \cup hyps(q,t)$

10.4 Proof theory of propositional logic

lemma *thms-mono*: $G \subseteq H ==> thms(G) \subseteq thms(H)$
 $\langle proof \rangle$

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

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

lemma *thms-MP*: $[| H \vdash p ==> q; H \vdash p |] ==> H \vdash q$
 — Stronger Modus Ponens rule: no typechecking!
 $\langle proof \rangle$

lemma *thms-I*: $p \in propn ==> H \vdash p ==> p$
 — Rule is called *I* for Identity Combinator, not for Introduction.
 $\langle proof \rangle$

10.4.1 Weakening, left and right

lemma *weaken-left*: $[| G \subseteq H; G \vdash p |] ==> H \vdash p$
 — Order of premises is convenient with *THEN*
 $\langle proof \rangle$

lemma *weaken-left-cons*: $H \vdash p ==> cons(a,H) \vdash p$
 $\langle proof \rangle$

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

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

lemma *weaken-right*: $[| H \vdash q; p \in propn |] ==> H \vdash p ==> q$
 $\langle proof \rangle$

10.4.2 The deduction theorem

theorem *deduction*: $[| cons(p,H) \vdash q; p \in propn |] ==> H \vdash p ==> q$
 $\langle proof \rangle$

10.4.3 The cut rule

lemma *cut*: $[| H |-p; \text{cons}(p,H) |- q |] ==> H |- q$
 $\langle \text{proof} \rangle$

lemma *thms-FlsE*: $[| H |- Fls; p \in \text{propn} |] ==> H |- p$
 $\langle \text{proof} \rangle$

lemma *thms-notE*: $[| H |- p=>Fls; H |- p; q \in \text{propn} |] ==> H |- q$
 $\langle \text{proof} \rangle$

10.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: $H |- p ==> H |= p$
 $\langle \text{proof} \rangle$

10.5 Completeness

10.5.1 Towards the completeness proof

lemma *Fls-Imp*: $[| H |- p=>Fls; q \in \text{propn} |] ==> H |- p=>q$
 $\langle \text{proof} \rangle$

lemma *Imp-Fls*: $[| H |- p; H |- q=>Fls |] ==> H |- (p=>q)=>Fls$
 $\langle \text{proof} \rangle$

lemma *hyps-thms-if*:
 $p \in \text{propn} ==> \text{hyps}(p,t) |- (\text{if is-true}(p,t) \text{ then } p \text{ else } p=>Fls)$
— Typical example of strengthening the induction statement.
 $\langle \text{proof} \rangle$

lemma *logcon-thms-p*: $[| p \in \text{propn}; 0 |= p |] ==> \text{hyps}(p,t) |- p$
— Key lemma for completeness; yields a set of assumptions satisfying p
 $\langle \text{proof} \rangle$

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*:
 $[| p \in \text{propn}; q \in \text{propn} |] ==> H |- (p=>q) ==> ((p=>Fls)=>q) ==> q$
 $\langle \text{proof} \rangle$

lemma *thms-excluded-middle-rule*:
 $[| \text{cons}(p,H) |- q; \text{cons}(p=>Fls,H) |- q; p \in \text{propn} |] ==> H |- q$
— Hard to prove directly because it requires cuts
 $\langle \text{proof} \rangle$

10.5.2 Completeness – lemmas for reducing the set of assumptions

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

lemma *hyps-Diff*:

$$p \in \text{propn} \implies \text{hyps}(p, t - \{v\}) \subseteq \text{cons}(\#v \Rightarrow \text{Fls}, \text{hyps}(p, t) - \{\#v\})$$

<proof>

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

lemma *hyps-cons*:

$$p \in \text{propn} \implies \text{hyps}(p, \text{cons}(v, t)) \subseteq \text{cons}(\#v, \text{hyps}(p, t) - \{\#v \Rightarrow \text{Fls}\})$$

<proof>

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]:

$$[\mid p \in \text{propn}; \ 0 \models p \mid] \implies \forall t. \text{hyps}(p, t) - \text{hyps}(p, t0) \vdash p$$

<proof>

10.5.3 Completeness theorem

lemma *completeness-0*: $[\mid p \in \text{propn}; \ 0 \models p \mid] \implies 0 \vdash p$

— The base case for completeness

<proof>

lemma *logcon-Imp*: $[\mid \text{cons}(p, H) \models q \mid] \implies H \models p \Rightarrow q$

— A semantic analogue of the Deduction Theorem

<proof>

lemma *completeness*:

$$H \in \text{Fin}(\text{propn}) \implies p \in \text{propn} \implies H \models p \implies H \vdash p$$

$$\langle \text{proof} \rangle$$

theorem *thms-iff*: $H \in \text{Fin}(\text{propn}) \implies H \vdash p \iff H \models p \wedge p \in \text{propn}$

$$\langle \text{proof} \rangle$$

end

11 Lists of n elements

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

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

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

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

$$\langle \text{proof} \rangle$$

lemma *listn-iff*: $\langle n, l \rangle \in \text{listn}(A) \iff l \in \text{list}(A) \ \& \ \text{length}(l) = n$

$$\langle \text{proof} \rangle$$

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

$$\langle \text{proof} \rangle$$

lemma *listn-mono*: $A \subseteq B \implies \text{listn}(A) \subseteq \text{listn}(B)$

$$\langle \text{proof} \rangle$$

lemma *listn-append*:

$$[\mid \langle n, l \rangle \in \text{listn}(A); \langle n', l' \rangle \in \text{listn}(A) \mid] \implies \langle n \# + n', l @ l' \rangle \in \text{listn}(A)$$

$$\langle \text{proof} \rangle$$

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

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

end

12 Combinatory Logic example: the Church-Rosser Theorem

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

Curiously, combinators do not include free variables.

Example taken from [?].

12.1 Definitions

Datatype definition of combinators S and K .

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

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

```
consts
  contract :: i
syntax
  -contract      :: [i,i] => o    (infixl -1-> 50)
  -contract-multi :: [i,i] => o    (infixl ----> 50)
translations
  p -1-> q == <p,q> ∈ contract
  p ----> q == <p,q> ∈ contract^*
```

```
syntax (xsymbols)
  comb.app    :: [i, i] => i      (infixl · 90)
```

inductive

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

intros

```
K: [| p ∈ comb; q ∈ comb |] ==> K·p·q -1-> p
S: [| p ∈ comb; q ∈ comb; r ∈ comb |] ==> S·p·q·r -1-> (p·r)·(q·r)
Ap1: [| p -1-> q; r ∈ comb |] ==> p·r -1-> q·r
Ap2: [| p -1-> q; r ∈ comb |] ==> r·p -1-> r·q
```

type-intros *comb.intros*

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

consts

parcontract :: *i*

syntax

```
-parcontract :: [i,i] => o    (infixl =1=> 50)
-parcontract-multi :: [i,i] => o    (infixl ===> 50)
```

translations

$p = 1 \Rightarrow q == \langle p, q \rangle \in \text{parcontract}$
 $p \Rightarrow \Rightarrow q == \langle p, q \rangle \in \text{parcontract}^+ +$

inductive

domains $\text{parcontract} \subseteq \text{comb} \times \text{comb}$

intros

refl: $[| p \in \text{comb} |] \Rightarrow p = 1 \Rightarrow p$
K: $[| p \in \text{comb}; q \in \text{comb} |] \Rightarrow K \cdot p \cdot q = 1 \Rightarrow p$
S: $[| p \in \text{comb}; q \in \text{comb}; r \in \text{comb} |] \Rightarrow S \cdot p \cdot q \cdot r = 1 \Rightarrow (p \cdot r) \cdot (q \cdot r)$
Ap: $[| p = 1 \Rightarrow q; r = 1 \Rightarrow s |] \Rightarrow p \cdot r = 1 \Rightarrow q \cdot s$
type-intros comb.intros

Misc definitions.

definition

$I :: i$ **where**
 $I == S \cdot K \cdot K$

definition

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

12.2 Transitive closure preserves the Church-Rosser property

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

$[| \text{diamond}(r); \langle x, y \rangle : r^+ + |] \Rightarrow$
 $\forall y'. \langle x, y' \rangle : r \dashrightarrow (\exists z. \langle y', z \rangle : r^+ + \ \& \ \langle y, z \rangle : r)$
<proof>

lemma *diamond-trancl*: $\text{diamond}(r) \Rightarrow \text{diamond}(r^+)$
<proof>

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

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

12.3 Results about Contraction

For type checking: replaces $a - 1 -> b$ 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*: $\text{field}(\text{contract}) = \text{comb}$
<proof>

lemmas *reduction-refl* =
field-contract-eq [*THEN equalityD2*, *THEN subsetD*, *THEN rtranc1-refl*]

lemmas *rtranc1-into-rtranc2* =
r-into-rtranc1 [*THEN trans-rtranc1* [*THEN transD*]]

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

lemmas *reduction-rls* =
contract.K [*THEN rtranc1-into-rtranc2*]
contract.S [*THEN rtranc1-into-rtranc2*]
contract.Ap1 [*THEN rtranc1-into-rtranc2*]
contract.Ap2 [*THEN rtranc1-into-rtranc2*]

lemma $p \in \text{comb} \implies I \cdot p \dashv\dashv\dashv p$
— 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!*]: $K -1-\> r$
and *S-contractE* [*elim!*]: $S -1-\> r$
and *Ap-contractE* [*elim!*]: $p \cdot q -1-\> r$

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

lemma *K1-contractD*: $K \cdot p -1-\> r \implies (\exists q. r = K \cdot q \ \& \ p -1-\> q)$
 $\langle \text{proof} \rangle$

lemma *Ap-reduce1*: $[p \dashv\dashv\dashv q; r \in \text{comb}] \implies p \cdot r \dashv\dashv\dashv q \cdot r$
 $\langle \text{proof} \rangle$

lemma *Ap-reduce2*: $[p \dashv\dashv\dashv q; r \in \text{comb}] \implies r \cdot p \dashv\dashv\dashv r \cdot q$
 $\langle \text{proof} \rangle$

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

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

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

lemma *KIII-contract3*: $K \cdot I \cdot ((K \cdot I) \cdot (K \cdot I)) - 1 -> I$
 $\langle proof \rangle$

lemma *not-diamond-contract*: $\neg diamond(contract)$
 $\langle proof \rangle$

12.5 Results about Parallel Contraction

For type checking: replaces $a = 1 => b$ by $a, b \in comb$

lemmas *parcontract-combE2* = *parcontract.dom-subset* [*THEN subsetD*, *THEN SigmaE2*]

and *parcontract-combD1* = *parcontract.dom-subset* [*THEN subsetD*, *THEN SigmaD1*]

and *parcontract-combD2* = *parcontract.dom-subset* [*THEN subsetD*, *THEN SigmaD2*]

lemma *field-parcontract-eq*: $field(parcontract) = comb$
 $\langle proof \rangle$

Derive a case for each combinator constructor.

inductive-cases

K-parcontractE [*elim!*]: $K = 1 => r$

and *S-parcontractE* [*elim!*]: $S = 1 => r$

and *Ap-parcontractE* [*elim!*]: $p \cdot q = 1 => r$

declare *parcontract.intros* [*intro*]

12.6 Basic properties of parallel contraction

lemma *K1-parcontractD* [*dest!*]:

$K \cdot p = 1 => r ==> (\exists p'. r = K \cdot p' \ \& \ p = 1 => p')$

$\langle proof \rangle$

lemma *S1-parcontractD* [*dest!*]:

$S \cdot p = 1 => r ==> (\exists p'. r = S \cdot p' \ \& \ p = 1 => p')$

$\langle proof \rangle$

lemma *S2-parcontractD* [*dest!*]:

$S \cdot p \cdot q = 1 => r ==> (\exists p' q'. r = S \cdot p' \cdot q' \ \& \ p = 1 => p' \ \& \ q = 1 => q')$

$\langle proof \rangle$

lemma *diamond-parcontract*: $diamond(parcontract)$

— Church-Rosser property for parallel contraction

$\langle proof \rangle$

Equivalence of $p ---> q$ and $p ==> q$.

lemma *contract-imp-parcontract*: $p - 1 -> q ==> p = 1 => q$

$\langle proof \rangle$

lemma *reduce-imp-parreduce*: $p \dashv\dashv \Rightarrow q \implies p \implies q$
 <proof>

lemma *parcontract-imp-reduce*: $p = 1 \Rightarrow q \implies p \dashv\dashv \Rightarrow q$
 <proof>

lemma *parreduce-imp-reduce*: $p \implies q \implies p \dashv\dashv \Rightarrow q$
 <proof>

lemma *parreduce-iff-reduce*: $p \implies q \iff p \dashv\dashv \Rightarrow q$
 <proof>

end

13 Primitive Recursive Functions: the inductive definition

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

Proof adopted from [?].

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

13.1 Basic definitions

definition

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

definition

$CONSTANT :: i \Rightarrow i$ **where**
 $CONSTANT(k) == \lambda l \in list(nat). k$

definition

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

definition

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

definition

$PREC :: [i, i] \Rightarrow i$ **where**
 $PREC(f, g) ==$
 $\lambda l \in list(nat). list-case(0,$
 $\lambda x xs. rec(x, f^l xs, \lambda y r. g \text{ ' } Cons(r, Cons(y, xs))), l)$

— Note that g is applied first to $PREC(f, g) \text{ ' } y$ and then to $y!$

consts

$ACK :: i \Rightarrow i$

primrec

$ACK(0) = SC$

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

abbreviation

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

$ack(x, y) == ACK(x) \text{ ' } [y]$

Useful special cases of evaluation.

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

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

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

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

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

lemma $PREC-succ$:

$[[x \in nat; l \in list(nat)]]$
 $\Rightarrow PREC(f, g) \text{ ' } (Cons(succ(x), l)) =$
 $g \text{ ' } Cons(PREC(f, g) \text{ ' } (Cons(x, l)), Cons(x, l))$
 $\langle proof \rangle$

13.2 Inductive definition of the PR functions

consts

$prim-rec :: i$

inductive

domains $prim-rec \subseteq list(nat) \rightarrow nat$

intros

$SC \in prim-rec$

$k \in nat \Rightarrow CONSTANT(k) \in prim-rec$

$i \in nat \Rightarrow PROJ(i) \in prim-rec$

$[[g \in prim-rec; fs \in list(prim-rec)]] \Rightarrow COMP(g, fs) \in prim-rec$

$[[f \in prim-rec; g \in prim-rec]] \Rightarrow PREC(f, g) \in prim-rec$

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]: $c \in \text{prim-rec} \implies c \in \text{list}(\text{nat}) \rightarrow \text{nat}$
 ⟨proof⟩

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

declare *prim-rec.intros* [TC]

declare *nat-into-Ord* [TC]

declare *rec-type* [TC]

lemma *ACK-in-prim-rec* [TC]: $i \in \text{nat} \implies \text{ACK}(i) \in \text{prim-rec}$
 ⟨proof⟩

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

13.3 Ackermann's function cases

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *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*:
 $[i1 \in \text{nat}; i2 \in \text{nat}; j \in \text{nat}]$
 $\implies \text{ack}(i1, \text{ack}(i2, j)) < \text{ack}(\text{succ}(\text{succ}(i1 \# + i2)), j)$
 — PROPERTY A 10
 $\langle \text{proof} \rangle$

lemma *ack-add-bound*:
 $[i1 \in \text{nat}; i2 \in \text{nat}; j \in \text{nat}]$
 $\implies \text{ack}(i1, j) \# + \text{ack}(i2, j) < \text{ack}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(i1 \# + i2)))), j)$
 — PROPERTY A 11
 $\langle \text{proof} \rangle$

lemma *ack-add-bound2*:
 $[i < \text{ack}(k, j); j \in \text{nat}; k \in \text{nat}]$
 $\implies i \# + j < \text{ack}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(k)))), j)$
 — PROPERTY A 12.
 — Article uses existential quantifier but the ALF proof used $k \# + \text{integ-of}(Pls \text{ BIT } 1 \text{ BIT } 0 \text{ BIT } 0)$.
 — Quantified version must be nested $\exists k'. \forall i, j \dots$

$\langle proof \rangle$

13.4 Main result

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

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

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

lemma *CONSTANT-case*:
 $[\mid l \in \text{list}(\text{nat}); k \in \text{nat} \mid] \implies \text{CONSTANT}(k) \text{ ' } l < \text{ack}(k, \text{list-add}(l))$
 $\langle proof \rangle$

lemma *PROJ-case* [*rule-format*]:
 $l \in \text{list}(\text{nat}) \implies \forall i \in \text{nat}. \text{PROJ}(i) \text{ ' } l < \text{ack}(0, \text{list-add}(l))$
 $\langle proof \rangle$

COMP case.

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

lemma *COMP-case*:
 $[\mid kg \in \text{nat};$
 $\quad \forall l \in \text{list}(\text{nat}). g'l < \text{ack}(kg, \text{list-add}(l));$
 $\quad fs \in \text{list}(\{f \in \text{prim-rec} .$
 $\quad \quad \exists kf \in \text{nat}. \forall l \in \text{list}(\text{nat}).$
 $\quad \quad \quad f'l < \text{ack}(kf, \text{list-add}(l))\}) \mid]$
 $\implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}). \text{COMP}(g, fs) \text{ ' } l < \text{ack}(k, \text{list-add}(l))$
 $\langle proof \rangle$

PREC case.

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

lemma *PREC-case*:

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

lemma *ack-bounds-prim-rec*:

$$f \in \text{prim-rec} \implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}). f'l < \text{ack}(k, \text{list-add}(l))$$

 $\langle \text{proof} \rangle$

theorem *ack-not-prim-rec*:

$$(\lambda l \in \text{list}(\text{nat}). \text{list-case}(0, \lambda x \, xs. \text{ack}(x,x), l)) \notin \text{prim-rec}$$

 $\langle \text{proof} \rangle$

end