

Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

May 22, 2012

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 $enum =$

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

end

2 Binary trees

theory $Binary-Trees$ **imports** $Main$ **begin**

2.1 Datatype definition

consts

$bt :: i \Rightarrow i$

datatype $bt(A) =$

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

declare $bt.intros [simp]$

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

$\langle proof \rangle$

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

$c \in C(Lf);$

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

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

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

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

$\langle proof \rangle$

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

consts $n\text{-}nodes :: i \Rightarrow i$

primrec

$n\text{-}nodes(Lf) = 0$

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

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

$\langle proof \rangle$

consts $n\text{-nodes-}aux :: i \Rightarrow i$

primrec

$n\text{-nodes-}aux(Lf) = (\lambda k \in nat. k)$

$n\text{-nodes-}aux(Br(a, l, r)) =$

$(\lambda k \in nat. n\text{-nodes-}aux(r) \text{ ‘ } (n\text{-nodes-}aux(l) \text{ ‘ } succ(k)))$

lemma $n\text{-nodes-}aux\text{-}eq$:

$t \in bt(A) \Rightarrow k \in nat \Rightarrow n\text{-nodes-}aux(t) \text{ ‘ } k = n\text{-nodes}(t) \text{ \#} + k$

$\langle proof \rangle$

definition

$n\text{-nodes-}tail :: i \Rightarrow i$ **where**

$n\text{-nodes-}tail(t) == n\text{-nodes-}aux(t) \text{ ‘ } 0$

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

$\langle proof \rangle$

2.3 Number of leaves

consts

$n\text{-leaves} :: i \Rightarrow i$

primrec

$n\text{-leaves}(Lf) = 1$

$n\text{-leaves}(Br(a, l, r)) = n\text{-leaves}(l) \text{ \#} + n\text{-leaves}(r)$

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

$\langle proof \rangle$

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]: $t \in bt(A) \Rightarrow bt\text{-reflect}(t) \in bt(A)$

$\langle proof \rangle$

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

$\langle proof \rangle$

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-elims *list-univ* [*THEN subsetD, elim-format*]

declare *Apply* [*TC*]

definition

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

term-rec(t, d) ==

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

definition

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

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

definition

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

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

definition

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

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

definition

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

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

definition

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

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

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

<proof>

lemma *term-induct2*:

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

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

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

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

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

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

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

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

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

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

term-rec – by *Vset* recursion.

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

$$\implies \text{map}(\lambda z. (\lambda x \in \text{Vset}(i). h(x)) ' z, l) = \text{map}(h, l)$$
 — *map* works correctly on the underlying list of terms.
 ⟨proof⟩

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*.
 ⟨proof⟩

lemma *term-rec-type*:

assumes $t: t \in \text{term}(A)$
and $a: !!x zs r. \llbracket x \in A; zs: \text{list}(\text{term}(A));$

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

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

shows $\text{term-rec}(t, d) \in C(t)$
 — Slightly odd typing condition on r in the second premise!
 ⟨proof⟩

lemma *def-term-rec*:

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

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

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

term-map.

lemma *term-map* [simp]:

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

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

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

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

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

term-size.

lemma *term-size* [simp]:

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

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

<proof>

reflect.

lemma *reflect* [simp]:

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

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

<proof>

preorder.

lemma *preorder* [simp]:

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

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

$\text{ntree} :: i \Rightarrow i$
 $\text{maptree} :: i \Rightarrow i$
 $\text{maptree2} :: [i, i] \Rightarrow i$

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

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

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

definition

$\text{ntree-rec} :: [[i, i, i] \Rightarrow i, i] \Rightarrow i$ **where**
 $\text{ntree-rec}(b) ==$
 $\text{Vrecursor}(\lambda \text{pr}. \text{ntree-case}(\lambda x h. b(x, h, \lambda i \in \text{domain}(h). \text{pr}'(h'i))))$

definition

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

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.
 ⟨proof⟩

lemma *ntree-induct-eqn* [consumes 1]:

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

and $f: f \in \text{ntree}(A) \rightarrow B$

and $g: g \in \text{ntree}(A) \rightarrow B$

and *step*: $!!x\ n\ h. [| x \in A; n \in \text{nat}; h \in n \rightarrow \text{ntree}(A); f\ O\ h = g\ O\ h |]$

$==>$

$f\ ' \text{Branch}(x,h) = g\ ' \text{Branch}(x,h)$

shows $f^t = g^t$

— Induction on $\text{ntree}(A)$ to prove an equation

⟨proof⟩

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

lemma *ntree-mono*: $A \subseteq B ==> \text{ntree}(A) \subseteq \text{ntree}(B)$

⟨proof⟩

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

— Easily provable by induction also

⟨proof⟩

lemma *ntree-subset-univ*: $A \subseteq \text{univ}(B) ==> \text{ntree}(A) \subseteq \text{univ}(B)$

⟨proof⟩

ntree recursion.

lemma *ntree-rec-Branch*:

function(h) $==>$

$\text{ntree-rec}(b, \text{Branch}(x,h)) = b(x, h, \lambda i \in \text{domain}(h). \text{ntree-rec}(b, h^i))$

⟨proof⟩

lemma *ntree-copy-Branch* [*simp*]:

function(h) $==>$

$\text{ntree-copy}(\text{Branch}(x, h)) = \text{Branch}(x, \lambda i \in \text{domain}(h). \text{ntree-copy}(h^i))$

⟨proof⟩

lemma *ntree-copy-is-ident*: $z \in \text{ntree}(A) ==> \text{ntree-copy}(z) = z$

⟨proof⟩

maptree

lemma *maptree-unfold*: $\text{maptree}(A) = A \times (\text{maptree}(A) \dashv\vdash \text{maptree}(A))$

⟨proof⟩

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

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

and *step*: $!!x\ n\ h. [| x \in A; h \in \text{maptree}(A) \dashv\vdash \text{maptree}(A);$

$\forall y \in \text{field}(h). P(y)$

```

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

declare tree-forest.intros [simp, TC]

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

```

lemma *forest-def*: $forest(A) == Part(tree-forest(A), Inr)$
 ⟨*proof*⟩

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

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

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

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

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

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

lemma

notes *rews* = *tree-forest.con-defs tree-def forest-def*

shows

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

⟨*proof*⟩

lemma *tree-forest-unfold'*:

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

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

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

Type checking for recursor: Not needed; possibly interesting?

lemma *TF-rec-type*:

[[$z \in tree-forest(A)$;
 !! $x f r$. [[$x \in A$; $f \in forest(A)$; $r \in C(f)$
]] ==> $b(x,f,r) \in C(Tcons(x,f))$;
 $c \in C(Fnil)$;
 !! $t f r1 r2$. [[$t \in tree(A)$; $f \in forest(A)$; $r1 \in C(t)$; $r2 \in C(f)$
]] ==> $d(t,f,r1,r2) \in C(Fcons(t,f))$
]] ==> $tree-forest-rec(b,c,d,z) \in C(z)$

<proof>

lemma *tree-forest-rec-type*:

$$\begin{aligned} & \llbracket \! \! \! x f r. \llbracket x \in A; f \in \text{forest}(A); r \in D(f) \\ & \quad \quad \quad \rrbracket \implies b(x,f,r) \in C(T\text{cons}(x,f)); \\ & \quad c \in D(F\text{nil}); \\ & \quad \! \! \! t f r1 r2. \llbracket t \in \text{tree}(A); f \in \text{forest}(A); r1 \in C(t); r2 \in D(f) \\ & \quad \quad \quad \rrbracket \implies d(t,f,r1,r2) \in D(F\text{cons}(t,f)) \\ & \rrbracket \implies (\forall t \in \text{tree}(A). \text{tree-forest-rec}(b,c,d,t) \in C(t)) \wedge \\ & \quad (\forall f \in \text{forest}(A). \text{tree-forest-rec}(b,c,d,f) \in D(f)) \\ & \text{--- Mutually recursive version.} \\ & \text{\textit{<proof>}} \end{aligned}$$

5.2 Operations

consts

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

primrec

$$\begin{aligned} \text{list-of-TF } (T\text{cons}(x,f)) &= [T\text{cons}(x,f)] \\ \text{list-of-TF } (F\text{nil}) &= [] \\ \text{list-of-TF } (F\text{cons}(t,tf)) &= \text{Cons } (t, \text{list-of-TF}(tf)) \end{aligned}$$

primrec

$$\begin{aligned} \text{of-list}([]) &= F\text{nil} \\ \text{of-list}(\text{Cons}(t,l)) &= F\text{cons}(t, \text{of-list}(l)) \end{aligned}$$

primrec

$$\begin{aligned} \text{map } (h, T\text{cons}(x,f)) &= T\text{cons}(h(x), \text{map}(h,f)) \\ \text{map } (h, F\text{nil}) &= F\text{nil} \\ \text{map } (h, F\text{cons}(t,tf)) &= F\text{cons } (\text{map}(h, t), \text{map}(h, tf)) \end{aligned}$$

primrec

$$\begin{aligned} \text{size } (T\text{cons}(x,f)) &= \text{succ}(\text{size}(f)) \\ \text{size } (F\text{nil}) &= 0 \\ \text{size } (F\text{cons}(t,tf)) &= \text{size}(t) \# + \text{size}(tf) \end{aligned}$$

primrec

$$\begin{aligned} \text{preorder } (T\text{cons}(x,f)) &= \text{Cons}(x, \text{preorder}(f)) \\ \text{preorder } (F\text{nil}) &= \text{Nil} \\ \text{preorder } (F\text{cons}(t,tf)) &= \text{preorder}(t) @ \text{preorder}(tf) \end{aligned}$$

primrec

$$\text{reflect } (T\text{cons}(x,f)) = T\text{cons}(x, \text{reflect}(f))$$

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

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

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

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

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

⟨proof⟩

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

⟨proof⟩

size.

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

⟨proof⟩

preorder.

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

⟨proof⟩

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

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

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

$R(Fnil);$

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

$]] ==> R(f)$

— Essentially the same as list induction.

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

Theorems about *map*.

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

⟨proof⟩

lemma *map-compose*:

$z \in \text{tree-forest}(A) \implies \text{map}(h, \text{map}(j, z)) = \text{map}(\lambda u. h(j(u)), z)$
<proof>

Theorems about *size*.

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

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

Theorems about *preorder*.

lemma *preorder-map*:

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

end

6 Infinite branching datatype definitions

theory *Brouwer* imports *Main-ZFC* begin

6.1 The Brouwer ordinals

consts

brouwer :: *i*

datatype $\subseteq V$ from(*0*, *csucc*(*nat*))

brouwer = *Zero* | *Suc* (*b* ∈ *brouwer*) | *Lim* (*h* ∈ *nat* → *brouwer*)

monos *Pi-mono*

type-intros *inf-datatype-intros*

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

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

assumes *b*: *b* ∈ *brouwer*

and cases:

P(*Zero*)

!!*b*. [*b* ∈ *brouwer*; *P*(*b*)] \implies *P*(*Suc*(*b*))

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

shows *P*(*b*)

— A nicer induction rule than the standard one.

<proof>

6.2 The Martin-Löf wellordering type

consts

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

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

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

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

monos $Pi-mono$

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

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

$\langle proof \rangle$

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.

$\langle proof \rangle$

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.

$\langle proof \rangle$

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

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

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

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

inductive**domains** $tiling(A) \subseteq Pow(\bigcup(A))$ **intros***empty*: $0 \in tiling(A)$ *Un*: $[[a \in A; t \in tiling(A); a \cap t = 0]] ==> a \cup t \in tiling(A)$ **type-intros** *empty-subsetI Union-upper Un-least PowI***type-elims** *PowD [elim-format]***definition***evnodd* :: $[i, i] ==> i$ **where** $evnodd(A, b) == \{z \in A. \exists i j. z = \langle i, j \rangle \wedge (i \# + j) \bmod 2 = b\}$ **7.1 Basic properties of evnodd****lemma** *evnodd-iff*: $\langle i, j \rangle: evnodd(A, b) \longleftrightarrow \langle i, j \rangle: A \ \& \ (i \# + j) \bmod 2 = b$
*<proof>***lemma** *evnodd-subset*: $evnodd(A, b) \subseteq A$
*<proof>***lemma** *Finite-evnodd*: $Finite(X) ==> Finite(evnodd(X, b))$
*<proof>***lemma** *evnodd-Un*: $evnodd(A \cup B, b) = evnodd(A, b) \cup evnodd(B, b)$
*<proof>***lemma** *evnodd-Diff*: $evnodd(A - B, b) = evnodd(A, b) - evnodd(B, b)$
*<proof>***lemma** *evnodd-cons* [*simp*]:
 $evnodd(cons(\langle i, j \rangle, C), b) =$
 $(if (i \# + j) \bmod 2 = b then cons(\langle i, j \rangle, evnodd(C, b)) else evnodd(C, b))$
*<proof>***lemma** *evnodd-0* [*simp*]: $evnodd(0, b) = 0$
*<proof>***7.2 Dominoes****lemma** *domino-Finite*: $d \in domino ==> Finite(d)$
*<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*:

$t \in \text{tiling}(A) \implies u \in \text{tiling}(A) \implies t \cap u = 0 \implies t \cup u \in \text{tiling}(A)$
<proof>

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

<proof>

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

<proof>

lemma *dominoes-tile-row*:

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

lemma *dominoes-tile-matrix*:

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

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

<proof>

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

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

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

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

<proof>

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

intros

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

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

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

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

lemma *cons-lemma1*: $[[x \notin C; x \notin B]] \implies \text{cons}(x, B) = \text{cons}(x, C) \iff B = C$
 <proof>

lemma *cons-lemma2*: $[[\text{cons}(x, B) = \text{cons}(y, C); x \neq y; x \notin B; y \notin C]] \implies B - \{y\} = C - \{x\} \ \& \ x \in C \ \& \ y \in B$
 <proof>

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

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

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

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

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

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

lemma *Diff-sing-imp*:

$\llbracket C - \{b\} = D - \{a\}; a \neq b; b \in C \rrbracket \implies C = \text{cons}(b, D) - \{a\}$
 <proof>

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

$n \in \text{nat}$

$\implies \forall C. |C| < n \longrightarrow$
 $(\forall x. \langle C, x \rangle \in \text{fold-set}(A, B, f, e) \longrightarrow$
 $(\forall y. \langle C, y \rangle \in \text{fold-set}(A, B, f, e) \longrightarrow y = x))$
 <proof>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

<proof>

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

$$\begin{aligned} & \llbracket C \in \text{Fin}(A); D \in \text{Fin}(A); C \cap D = 0 \rrbracket \\ & \implies \text{fold}[B](f, e, C \cup D) = \text{fold}[B](f, \text{fold}[B](f, e, D), C) \end{aligned}$$

<proof>

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

<proof>

7.4 The Operator *setsum*

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

<proof>

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

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

<proof>

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

<proof>

lemma *setsum-Un-Int*:

$$\begin{aligned} & \llbracket \text{Finite}(C); \text{Finite}(D) \rrbracket \\ & \implies \text{setsum}(g, C \cup D) \$+ \text{setsum}(g, C \cap D) \\ & \quad = \text{setsum}(g, C) \$+ \text{setsum}(g, D) \end{aligned}$$

<proof>

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

<proof>

lemma *setsum-Un-disjoint*:

$$\begin{aligned} & \llbracket \text{Finite}(C); \text{Finite}(D); C \cap D = 0 \rrbracket \\ & \implies \text{setsum}(g, C \cup D) = \text{setsum}(g, C) \$+ \text{setsum}(g, D) \end{aligned}$$

<proof>

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

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

<proof>

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

$Finite(I)$
 $\implies (\forall i \in I. Finite(C(i))) \longrightarrow$
 $(\forall i \in I. \forall j \in I. i \neq j \longrightarrow C(i) \cap C(j) = 0) \longrightarrow$
 $setsum(f, \bigcup i \in I. C(i)) = setsum (\%i. setsum(f, C(i)), I)$
 <proof>

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

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

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

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

lemma *setsum-Un*:
 $[[Finite(A); Finite(B)]]$
 $\implies setsum(f, A \cup B) =$
 $setsum(f, A) \$+ setsum(f, B) \$- setsum(f, A \cap B)$
 <proof>

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

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

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

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

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

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

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

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

end

8 The accessible part of a relation

theory *Acc* imports *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*: $[[!!b. <b,a>:r ==> b \in acc(r); a \in field(r)]] ==> a \in acc(r)$
 <proof>

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

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

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

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

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

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

end

theory *Multiset*
imports *FoldSet Acc*
begin

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

definition

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

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**
 $M + \# N == \lambda x \in mset-of(M) \cup mset-of(N).$
 if $x \in mset-of(M) \cap mset-of(N)$ *then* $(M'x) \# + (N'x)$
 else (*if* $x \in mset-of(M)$ *then* $M'x$ *else* $N'x$)

definition

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

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

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

definition

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

definition

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

definition

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

definition

$\text{msize} :: i \Rightarrow i$ **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**) **where**
 $a :# M == a \in \text{mset-of}(M)$

syntax

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

syntax (*xsymbols*)

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

translations

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

definition

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

definition

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

definition

omultiset :: *i* => *o* **where**
omultiset(*M*) == ∃ *i*. *Ord*(*i*) & *M* ∈ *Mult*(*field*(*Memrel*(*i*)))

definition

mless :: [*i*, *i*] => *o* (**infixl** <# 50) **where**
M <# *N* == ∃ *i*. *Ord*(*i*) & <*M*, *N*> ∈ *multirel*(*field*(*Memrel*(*i*)), *Memrel*(*i*))

definition

mle :: [*i*, *i*] => *o* (**infixl** <#= 50) **where**
M <#= *N* == (*omultiset*(*M*) & *M* = *N*) | *M* <# *N*

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

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

lemma *funrestrict-type*:

[| !!*x*. *x* ∈ *A* ==> *f*'*x* ∈ *B*(*x*) |] ==> *funrestrict*(*f*,*A*) ∈ *Pi*(*A*,*B*)
 <proof>

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

lemma *funrestrict [simp]*: *a* ∈ *A* ==> *funrestrict*(*f*,*A*) ' *a* = *f*'*a*
 <proof>

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

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

lemma *fun-cons-funrestrict-eq*:

f ∈ *cons*(*a*, *b*) -> *B* ==> *f* = *cons*(<*a*, *f* ' *a*>, *funrestrict*(*f*, *b*))
 <proof>

declare *domain-of-fun [simp]*

declare *domainE [rule del]*

A useful simplification rule

lemma *multiset-fun-iff*:

(*f* ∈ *A* -> *nat*-{*0*}) ↔ *f* ∈ *A*->*nat*&(∀ *a* ∈ *A*. *f*'*a* ∈ *nat* & 0 < *f*'*a*)
 <proof>

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

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

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

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

The *multiset* operator

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

The *mset-of* operator

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

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

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

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

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

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

lemma *msingle-not-0* [*iff*]: $\{\#a\} \neq 0 \ \& \ 0 \neq \{\#a\}$
<proof>

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

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

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

lemma *normalize-idem [simp]: normalize(normalize(f)) = normalize(f)*
<proof>

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

lemma *multiset-normalize [simp]: multiset(normalize(f))*
<proof>

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

lemma *mdiff-multiset [simp]: multiset(M -# N)*
<proof>

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

lemma *munion-commute: M +# N = N +# M*
<proof>

lemma *munion-assoc: (M +# N) +# K = M +# (N +# K)*
<proof>

lemma *munion-lcommute: M +# (N +# K) = N +# (M +# K)*
<proof>

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

lemma *mdiff-self-eq-0 [simp]: M -# M = 0*
<proof>

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

<proof>

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

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

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

lemma *mcoun-0* [*simp*]: $\text{mcoun}(0, a) = 0$
<proof>

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

lemma *mcoun-union* [*simp*]: $[\text{multiset}(M); \text{multiset}(N)] \implies \text{mcoun}(M +\# N, a) = \text{mcoun}(M, a) \#+ \text{mcoun}(N, a)$
<proof>

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

lemma *mcoun-elim*: $[\text{multiset}(M); a \in \text{mset-of}(M)] \implies 0 < \text{mcoun}(M, a)$
<proof>

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

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

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

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

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

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

$[[M \neq 0; \text{multiset}(M)]] \implies \exists a \in \text{mset-of}(M). 0 < \text{mcount}(M, a)$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

lemma *munion-is-single*:

$[[\text{multiset}(M); \text{multiset}(N)]] \implies (M \# + N = \{\#a\}) \longleftrightarrow (M = \{\#a\} \& N = 0) \mid (M = 0 \& N = \{\#a\})$
 <proof>

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

$\implies (\{\#a\} = M \# + N) \longleftrightarrow (\{\#a\} = M \& N = 0 \mid M = 0 \& \{\#a\} = N)$
 <proof>

lemma *setsum-decr*:

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

lemma *setsum-decr2*:

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

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

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

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

<proof>

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

<proof>

Specialized for use in the proof below.

lemma *multiset-funrestrict*:

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

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

lemma *multiset-induct-aux*:

assumes *prem1*: $\llbracket M \text{ a. } \llbracket \text{multiset}(M); a \notin \text{mset-of}(M); P(M) \rrbracket \implies P(\text{cons}(\langle a, 1 \rangle, M))$

and *prem2*: $\llbracket M \text{ b. } \llbracket \text{multiset}(M); b \in \text{mset-of}(M); P(M) \rrbracket \implies P(M(b := M \text{ ' } b \text{ \# } + 1))$

shows

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

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

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

<proof>

lemma *multiset-induct2*:

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

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

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

\llbracket

$\implies P(M)$

<proof>

lemma *munion-single-case1*:

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

<proof>

lemma *munion-single-case2*:

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

<proof>

lemma *multiset-induct*:

assumes *M*: *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*)

<proof>

lemma *MCollect-multiset* [*simp*]:

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

<proof>

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

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

<proof>

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

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

<proof>

lemma *mcount-MCollect* [*simp*]:

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

<proof>

lemma *multiset-partition*: $\text{multiset}(M) \implies M = \{\#x \in M. P(x)\# \} +\# \{\#x \in M. \sim P(x)\# \}$

<proof>

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

$$[[\text{multiset}(M); a \in \text{mset-of}(M)]] \implies \text{natify}(M'a) = M'a$$

<proof>

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

$$\implies (M +\# \{\#a\# \} = N +\# \{\#b\# \}) \longleftrightarrow (M = N \ \& \ a = b \mid$$

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

<proof>

lemma *melem-diff-single*:

$\text{multiset}(M) \implies$

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

<proof>

lemma *munion-eq-conv-exist*:

$[[M \in \text{Mult}(A); N \in \text{Mult}(A)]]$

$$\implies (M +\# \{\#a\# \} = N +\# \{\#b\# \}) \longleftrightarrow$$

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

<proof>

8.2 Multiset Orderings

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

<proof>

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

<proof>

lemma *multirel1-iff*:

$$\langle N, M \rangle \in \text{multirel1}(A, r) \longleftrightarrow$$

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

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

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

Monotonicity of *multirel1*

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

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

lemma *multirel1-mono*:
 $[\![A \subseteq B; r \subseteq s \!]\!] \implies \text{multirel1}(A, r) \subseteq \text{multirel1}(B, s)$
 <proof>

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

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

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

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

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

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

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

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

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

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: $\text{wf}[A](r) \implies \text{wf}[A - || > \text{nat} - \{0\}](\text{multirel1}(A, r))$
 $\langle \text{proof} \rangle$

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

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

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

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

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

lemma melem-imp-eq-diff-union [simp]: $[[a \in \text{mset-of}(M); \text{multiset}(M)]] \implies$

$M -\# \{ \#a\# \} +\# \{ \#a\# \} = M$
 ⟨proof⟩

lemma *msize-eq-succ-imp-eq-union*:

[[$msize(M) = \# \# succ(n)$; $M \in Mult(A)$; $n \in nat$]]
 $\implies \exists a N. M = N +\# \{ \#a\# \} \ \& \ N \in Mult(A) \ \& \ a \in A$
 ⟨proof⟩

lemma *one-step-implies-multirel-lemma* [rule-format (no-asm)]:

$n \in nat \implies$
 $(\forall I J K.$
 $I \in Mult(A) \ \& \ J \in Mult(A) \ \& \ K \in Mult(A) \ \&$
 $(msize(J) = \# \# n \ \& \ J \neq 0 \ \& \ (\forall k \in mset-of(K). \exists j \in mset-of(J). \langle k, j \rangle \in$
 $r))$
 $\longrightarrow \langle I +\# K, I +\# J \rangle \in multirel(A, r))$
 ⟨proof⟩

lemma *one-step-implies-multirel*:

[[$J \neq 0$; $\forall k \in mset-of(K). \exists j \in mset-of(J). \langle k, j \rangle \in r$;
 $I \in Mult(A)$; $J \in Mult(A)$; $K \in Mult(A)$]]
 $\implies \langle I +\# K, I +\# J \rangle \in multirel(A, r)$
 ⟨proof⟩

lemma *multirel-irrefl-lemma*:

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

lemma *irrefl-on-multirel*:

$part-ord(A, r) \implies irrefl(Mult(A), multirel(A, r))$
 ⟨proof⟩

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

⟨proof⟩

lemma *multirel-trans*:

[[$\langle M, N \rangle \in multirel(A, r)$; $\langle N, K \rangle \in multirel(A, r)$]] $\implies \langle M, K \rangle \in$
 $multirel(A, r)$
 ⟨proof⟩

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

⟨proof⟩

lemma *part-ord-multirel*: $part-ord(A, r) \implies part-ord(Mult(A), multirel(A, r))$

<proof>

lemma *munion-multirel1-mono*:

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

<proof>

lemma *munion-multirel-mono2*:

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

<proof>

lemma *munion-multirel-mono1*:

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

<proof>

lemma *munion-multirel-mono*:

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

<proof>

8.4 Ordinal Multisets

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

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

lemma *omultiset-is-multiset [simp]*: $\text{omultiset}(M) \implies \text{multiset}(M)$

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

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

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

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

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

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

lemma *mless-asym*: $[| M <\# N; \sim P \implies N <\# M |] \implies P$
<proof>

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

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

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

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

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

lemma *munion-less-mono1*: $[| M <\# N; omultiset(K) |] \implies M +\# K <\# N$

$+ \# K$
 $\langle proof \rangle$

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

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

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

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

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

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

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

end

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

theory *Rmap* imports *Main* begin

consts

$rmap :: i \Rightarrow i$

inductive

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

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*: $r \subseteq s \implies rmap(r) \subseteq rmap(s)$
 $\langle proof \rangle$

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*: $r \subseteq A \times B \implies rmap(r) \subseteq list(A) \times list(B)$
<proof>

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

lemma *rmap-functional*: $function(r) \implies function(rmap(r))$
<proof>

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

lemma *rmap-fun-type*: $f \in A \rightarrow B \implies rmap(f): list(A) \rightarrow list(B)$
<proof>

lemma *rmap-Nil*: $rmap(f) ' Nil = Nil$
<proof>

lemma *rmap-Cons*: $[| f \in A \rightarrow B; x \in A; xs: list(A) |]$
 $\implies rmap(f) ' Cons(x, xs) = Cons(f'x, rmap(f) ' xs)$
<proof>

end

10 Meta-theory of propositional logic

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

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

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

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

10.1 The datatype of propositions

consts

propn :: i

datatype *propn* =

Fls

| *Var* ($n \in nat$) ($\#$ - [100] 100)

| *Imp* ($p \in propn, q \in propn$) (**infixr** \implies 90)

10.2 The proof system

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

abbreviation

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

inductive

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

intros

H: $[\mid p \in H; p \in \text{propn} \mid] \Rightarrow H \vdash p$
K: $[\mid p \in \text{propn}; q \in \text{propn} \mid] \Rightarrow H \vdash p \Rightarrow q \Rightarrow p$
S: $[\mid p \in \text{propn}; q \in \text{propn}; r \in \text{propn} \mid]$
 $\Rightarrow H \vdash (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$
DN: $p \in \text{propn} \Rightarrow H \vdash ((p \Rightarrow \text{Fls}) \Rightarrow \text{Fls}) \Rightarrow p$
MP: $[\mid H \vdash p \Rightarrow q; H \vdash p; p \in \text{propn}; q \in \text{propn} \mid] \Rightarrow H \vdash q$
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*]: $\text{is-true}(\text{Fls}, t) \longleftrightarrow \text{False}$
(*proof*)

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

lemma *is-true-Imp* [*simp*]: $\text{is-true}(p \Rightarrow q, t) \longleftrightarrow (\text{is-true}(p, t) \longrightarrow \text{is-true}(q, t))$
(*proof*)

10.3.2 Logical consequence

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

definition

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

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

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

10.4.1 Weakening, left and right

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

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

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

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

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

10.4.2 The deduction theorem

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

10.4.3 The cut rule

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

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

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

10.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: $H |- p ==> H |= p$
<proof>

10.5 Completeness

10.5.1 Towards the completeness proof

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

lemma *Imp-Fls*: $[[H |- p; H |- q=>Fls]] ==> H |- (p=>q)=>Fls$
<proof>

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.
<proof>

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
<proof>

For proving certain theorems in our new propositional logic.

lemmas *propn-SIs* = *propn.intros deduction*
and *propn-Is* = *thms-in-pl thms.H thms.H [THEN thms-MP]*

The excluded middle in the form of an elimination rule.

lemma *thms-excluded-middle*:
 $[[p \in \text{propn}; q \in \text{propn}]] ==> H |- (p=>q) => ((p=>Fls)=>q) => q$
<proof>

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
<proof>

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]:

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

<proof>

10.5.3 Completeness theorem

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

— The base case for completeness

<proof>

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

— A semantic analogue of the Deduction Theorem

<proof>

lemma *completeness*:

$H \in \text{Fin}(\text{propn}) \implies p \in \text{propn} \implies H \models p \implies H \Vdash p$
 ⟨proof⟩

theorem *thms-iff*: $H \in \text{Fin}(\text{propn}) \implies H \Vdash p \longleftrightarrow H \models p \wedge p \in \text{propn}$
 ⟨proof⟩

end

11 Lists of n elements

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

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

consts *listn* :: $i \implies i$

inductive

domains $\text{listn}(A) \subseteq \text{nat} \times \text{list}(A)$

intros

NilI: $\langle 0, \text{Nil} \rangle \in \text{listn}(A)$

ConsI: $\llbracket a \in A; \langle n, l \rangle \in \text{listn}(A) \rrbracket \implies \langle \text{succ}(n), \text{Cons}(a, l) \rangle \in \text{listn}(A)$

type-intros *nat-typechecks list.intros*

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

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

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

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

lemma *listn-append*:

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

⟨proof⟩

inductive-cases

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

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

inductive-cases

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

and *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)
```

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

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

```
consts
  contract :: i
```

```
abbreviation
  contract-syntax :: [i,i] => o  (infixl -1-> 50)
  where p -1-> q == <p,q> ∈ contract
```

```
abbreviation
  contract-multi :: [i,i] => o  (infixl -> 50)
  where p -> q == <p,q> ∈ contract^*
```

```
inductive
  domains contract ⊆ comb × comb
```

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

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

```
consts
  parcontract :: i
```

abbreviation

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

abbreviation

parcontract-multi :: $[i,i] \Rightarrow o$ (**infixl** ===> 50)
where $p ===> q == \langle p,q \rangle \in \text{parcontract}^{\wedge+}$

inductive

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

intros

refl: $[[p \in \text{comb}]] \implies p =1=> p$
K: $[[p \in \text{comb}; q \in \text{comb}]] \implies K \cdot p \cdot q =1=> p$
S: $[[p \in \text{comb}; q \in \text{comb}; r \in \text{comb}]] \implies S \cdot p \cdot q \cdot r =1=> (p \cdot r) \cdot (q \cdot r)$
Ap: $[[p =1=> q; r =1=> s]] \implies p \cdot r =1=> q \cdot s$

type-intros *comb.intros*

Misc definitions.

definition

I :: *i* **where**
I == *S* · *K* · *K*

definition

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

12.2 Transitive closure preserves the Church-Rosser property

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

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

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

<proof>

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

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*: $field(contract) = comb$
 ⟨*proof*⟩

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

lemmas *rtrancl-into-rtrancl2* =
r-into-rtrancl [THEN *trans-rtrancl* [THEN *transD*]]

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

lemmas *reduction-rls* =
contract.K [THEN *rtrancl-into-rtrancl2*]
contract.S [THEN *rtrancl-into-rtrancl2*]
contract.Ap1 [THEN *rtrancl-into-rtrancl2*]
contract.Ap2 [THEN *rtrancl-into-rtrancl2*]

lemma $p \in comb \implies I \cdot p \longrightarrow p$
 — Example only: not used
 ⟨*proof*⟩

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

12.4 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases

K-contractE [elim!]: $K -1-\> r$
and *S-contractE* [elim!]: $S -1-\> r$
and *Ap-contractE* [elim!]: $p \cdot q -1-\> r$

lemma *I-contract-E*: $I -1-\> r \implies P$
 ⟨*proof*⟩

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

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

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

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

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

lemma *KIII-contract2*: $K \cdot I \cdot (I \cdot I) -1-\> K \cdot I \cdot ((K \cdot I) \cdot (K \cdot I))$

<proof>

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

lemma *not-diamond-contract*: $\neg \text{diamond}(\text{contract})$
<proof>

12.5 Results about Parallel Contraction

For type checking: replaces $a = 1 \Rightarrow b$ by $a, b \in \text{comb}$

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*: $\text{field}(\text{parcontract}) = \text{comb}$
<proof>

Derive a case for each combinator constructor.

inductive-cases

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

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

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

declare *parcontract.intros* [*intro*]

12.6 Basic properties of parallel contraction

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

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

<proof>

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

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

<proof>

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

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

<proof>

lemma *diamond-parcontract*: $\text{diamond}(\text{parcontract})$

— Church-Rosser property for parallel contraction

<proof>

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

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

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

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

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

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

end

13 Primitive Recursive Functions: the inductive definition

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

Proof adopted from [?].

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

13.1 Basic definitions

definition

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

definition

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

definition

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

definition

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

definition

PREC :: $[i,i]=>i$ **where**
 $PREC(f,g) ==$
 $\lambda l \in list(nat). list-case(0,$

$\lambda x xs. \text{rec}(x, f'xs, \lambda y r. g \text{ ' } \text{Cons}(r, \text{Cons}(y, xs))), l)$

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

consts

$ACK :: i \Rightarrow i$

primrec

$ACK(0) = SC$

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

abbreviation

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

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

Useful special cases of evaluation.

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

lemma *CONSTANT*: $l \in \text{list}(\text{nat}) \Rightarrow \text{CONSTANT}(k) \text{ ' } l = k$
<proof>

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

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

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

lemma *PREC-succ*:

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

13.2 Inductive definition of the PR functions

consts

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

inductive

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

intros

$SC \in \text{prim-rec}$

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

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

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

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

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

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

declare *prim-rec.intros* [TC]
declare *nat-into-Ord* [TC]
declare *rec-type* [TC]

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

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

13.3 Ackermann's function cases

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

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

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

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
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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
<proof>

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
<proof>

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
<proof>

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 <
<proof>

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
<proof>

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

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
<proof>

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
<proof>

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
<proof>

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 $k \# + \text{integ-of}(Pls$

BIT 1 BIT 0 BIT 0).

— Quantified version must be nested $\exists k'. \forall i, j \dots$

<proof>

13.4 Main result

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

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

<proof>

lemma *lt-ack1*: $[[i \in \text{nat}; j \in \text{nat}]] \implies i < \text{ack}(i, j)$

— PROPERTY A 4? Extra lemma needed for *CONSTANT* case, constant functions.

<proof>

lemma *CONSTANT-case*:

$[[l \in \text{list}(\text{nat}); k \in \text{nat}]] \implies \text{CONSTANT}(k) \text{ } l < \text{ack}(k, \text{list-add}(l))$

<proof>

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

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

<proof>

COMP case.

lemma *COMP-map-lemma*:

$fs \in \text{list}(\{f \in \text{prim-rec}. \exists kf \in \text{nat}. \forall l \in \text{list}(\text{nat}). f^l < \text{ack}(kf, \text{list-add}(l))\})$

$\implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}).$

$\text{list-add}(\text{map}(\lambda f. f \text{ } l, fs)) < \text{ack}(k, \text{list-add}(l))$

<proof>

lemma *COMP-case*:

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

$\forall l \in \text{list}(\text{nat}). g^l < \text{ack}(kg, \text{list-add}(l));$

$fs \in \text{list}(\{f \in \text{prim-rec} .$

$\exists kf \in \text{nat}. \forall l \in \text{list}(\text{nat}).$

$f^l < \text{ack}(kf, \text{list-add}(l))\})]]$

$\implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}). \text{COMP}(g, fs)^l < \text{ack}(k, \text{list-add}(l))$

<proof>

PREC case.

lemma *PREC-case-lemma*:

$[[\forall l \in \text{list}(\text{nat}). f^l \# + \text{list-add}(l) < \text{ack}(kf, \text{list-add}(l));$

$\forall l \in \text{list}(\text{nat}). g^l \# + \text{list-add}(l) < \text{ack}(kg, \text{list-add}(l));$

$f \in \text{prim-rec}; kf \in \text{nat};$

$g \in \text{prim-rec}; kg \in \text{nat};$

$l \in \text{list}(\text{nat})]]$

$\implies \text{PREC}(f, g)^l \# + \text{list-add}(l) < \text{ack}(\text{succ}(kf \# + kg), \text{list-add}(l))$

<proof>

lemma *PREC-case:*

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

lemma *ack-bounds-prim-rec:*

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

theorem *ack-not-prim-rec:*

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

end