

# Examples of Inductive and Coinductive Definitions in ZF

Lawrence C Paulson and others

December 17, 2025

## Contents

<b>1</b>	<b>Sample datatype definitions</b>	<b>2</b>
1.1	A type with four constructors . . . . .	3
1.2	Example of a big enumeration type . . . . .	3
<b>2</b>	<b>Binary trees</b>	<b>4</b>
2.1	Datatype definition . . . . .	4
2.2	Number of nodes, with an example of tail-recursion . . . . .	5
2.3	Number of leaves . . . . .	5
2.4	Reflecting trees . . . . .	6
<b>3</b>	<b>Terms over an alphabet</b>	<b>6</b>
<b>4</b>	<b>Datatype definition n-ary branching trees</b>	<b>11</b>
<b>5</b>	<b>Trees and forests, a mutually recursive type definition</b>	<b>14</b>
5.1	Datatype definition . . . . .	14
5.2	Operations . . . . .	16
<b>6</b>	<b>Infinite branching datatype definitions</b>	<b>19</b>
6.1	The Brouwer ordinals . . . . .	19
6.2	The Martin-Löf wellordering type . . . . .	19
<b>7</b>	<b>The Mutilated Chess Board Problem, formalized inductively</b>	<b>20</b>
7.1	Basic properties of <i>evnodd</i> . . . . .	21
7.2	Dominoes . . . . .	21
7.3	Tilings . . . . .	21
7.4	The Operator <i>setsum</i> . . . . .	28

<b>8</b>	<b>The accessible part of a relation</b>	<b>31</b>
8.1	Properties of the original "restrict" from ZF.thy . . . . .	34
8.2	Multiset Orderings . . . . .	47
8.3	Toward the proof of well-foundedness of multirell . . . . .	48
8.4	Ordinal Multisets . . . . .	56
<b>9</b>	<b>An operator to “map” a relation over a list</b>	<b>59</b>
<b>10</b>	<b>Meta-theory of propositional logic</b>	<b>60</b>
10.1	The datatype of propositions . . . . .	60
10.2	The proof system . . . . .	61
10.3	The semantics . . . . .	61
10.3.1	Semantics of propositional logic. . . . .	61
10.3.2	Logical consequence . . . . .	62
10.4	Proof theory of propositional logic . . . . .	62
10.4.1	Weakening, left and right . . . . .	62
10.4.2	The deduction theorem . . . . .	63
10.4.3	The cut rule . . . . .	63
10.4.4	Soundness of the rules wrt truth-table semantics . . .	63
10.5	Completeness . . . . .	63
10.5.1	Towards the completeness proof . . . . .	63
10.5.2	Completeness – lemmas for reducing the set of as- sumptions . . . . .	65
10.5.3	Completeness theorem . . . . .	66
<b>11</b>	<b>Lists of n elements</b>	<b>66</b>
<b>12</b>	<b>Combinatory Logic example: the Church-Rosser Theorem</b>	<b>67</b>
12.1	Definitions . . . . .	68
12.2	Transitive closure preserves the Church-Rosser property . . .	69
12.3	Results about Contraction . . . . .	69
12.4	Non-contraction results . . . . .	70
12.5	Results about Parallel Contraction . . . . .	71
12.6	Basic properties of parallel contraction . . . . .	71
<b>13</b>	<b>Primitive Recursive Functions: the inductive definition</b>	<b>72</b>
13.1	Basic definitions . . . . .	73
13.2	Inductive definition of the PR functions . . . . .	74
13.3	Ackermann’s function cases . . . . .	74
13.4	Main result . . . . .	77

## 1 Sample datatype definitions

`theory Datatypes imports ZF begin`

## 1.1 A type with four constructors

It has four constructors, of arities 0–3, and two parameters  $A$  and  $B$ .

**consts**

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

**datatype**  $data(A, B) =$

$Con0$   
 $| Con1 (a \in A)$   
 $| Con2 (a \in A, b \in B)$   
 $| Con3 (a \in A, b \in B, d \in data(A, B))$

**lemma**  $data-unfold: data(A, B) = (\{0\} + A) + (A \times B + A \times B \times data(A, B))$

**by** ( $fast\ intro! : data.intros [unfolded\ data.con-defs]$   
 $elim : data.cases [unfolded\ data.con-defs]$ )

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

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

**unfolding**  $data.defs$   
**apply** ( $rule\ lfp-mono$ )  
**apply** ( $rule\ data.bnd-mono$ )  
**apply** ( $rule\ univ-mono\ Un-mono\ basic-monos\ | assumption$ )  
**done**

**lemma**  $data-univ: data(univ(A), univ(A)) \subseteq univ(A)$

**unfolding**  $data.defs\ data.con-defs$   
**apply** ( $rule\ lfp-lowerbound$ )  
**apply** ( $rule-tac\ [2]\ subset-trans\ [OF\ A-subset-univ\ Un-upper1,\ THEN\ univ-mono]$ )  
**apply** ( $fast\ intro! : zero-in-univ\ Inl-in-univ\ Inr-in-univ\ Pair-in-univ$ )  
**done**

**lemma**  $data-subset-univ:$

$\llbracket A \subseteq univ(C); B \subseteq univ(C) \rrbracket \Longrightarrow data(A, B) \subseteq univ(C)$   
**by** ( $rule\ subset-trans\ [OF\ data-mono\ data-univ]$ )

## 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** *ZF* **begin**

### 2.1 Datatype definition

**consts**

*bt* :: *i*  $\Rightarrow$  *i*

**datatype** *bt*(*A*) =

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

**declare** *bt.intros* [*simp*]

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

**by** (*induct arbitrary: x r set: bt*) *auto*

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

— Proving a freeness theorem.

**by** (*fast elim!: bt.free-elim*s)

**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*:  $A \subseteq B \implies bt(A) \subseteq bt(B)$

**unfolding** *bt.defs*

**apply** (*rule lfp-mono*)

**apply** (*rule bt.bnd-mono*)**+**

**apply** (*rule univ-mono basic-monos | assumption*)**+**

**done**

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

**unfolding** *bt.defs bt.con-defs*

**apply** (*rule lfp-lowerbound*)

**apply** (*rule-tac* [2] *A-subset-univ [THEN univ-mono]*)

**apply** (*fast intro!: zero-in-univ Inl-in-univ Inr-in-univ Pair-in-univ*)

**done**

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

**apply** (*rule subset-trans*)

**apply** (*erule bt-mono*)

**apply** (*rule bt-univ*)

**done**

**lemma** *bt-rec-type*:

```

   $\llbracket t \in bt(A);$ 
     $c \in C(Lf);$ 
     $\bigwedge x\ y\ z\ r\ s. \llbracket x \in A; y \in bt(A); z \in bt(A); r \in C(y); s \in C(z) \rrbracket \implies$ 
     $h(x, y, z, r, s) \in C(Br(x, y, z))$ 
 $\rrbracket \implies bt-rec(c, h, t) \in C(t)$ 
  — Type checking for recursor – example only; not really needed.
  apply (induct-tac t)
  apply simp-all
done

```

## 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*]:  $t \in bt(A) \implies n-nodes(t) \in nat$

**by** (*induct set: bt*) *auto*

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

**primrec**

```

  n-nodes-aux(Lf) = ( $\lambda k \in nat. k$ )
  n-nodes-aux(Br(a, l, r)) =
    ( $\lambda k \in nat. n-nodes-aux(r) \text{ ' } (n-nodes-aux(l) \text{ ' } succ(k))$ )

```

**lemma** *n-nodes-aux-eq*:

$t \in bt(A) \implies k \in nat \implies n-nodes-aux(t) \text{ ' } k = n-nodes(t) \# + k$

**apply** (*induct arbitrary: k set: bt*)

**apply** *simp*

**apply** (*atomize, simp*)

**done**

**definition**

*n-nodes-tail* ::  $i \Rightarrow i$  **where**

$n-nodes-tail(t) \equiv n-nodes-aux(t) \text{ ' } 0$

**lemma**  $t \in bt(A) \implies n-nodes-tail(t) = n-nodes(t)$

**by** (*simp add: n-nodes-tail-def n-nodes-aux-eq*)

## 2.3 Number of leaves

**consts**

*n-leaves* ::  $i \Rightarrow i$

**primrec**

```

  n-leaves(Lf) = 1
  n-leaves(Br(a, l, r)) = n-leaves(l) #+ n-leaves(r)

```

**lemma** *n-leaves-type* [simp]:  $t \in \text{bt}(A) \implies \text{n-leaves}(t) \in \text{nat}$   
**by** (induct set: *bt*) *auto*

## 2.4 Reflecting trees

**consts**

*bt-reflect* ::  $i \Rightarrow i$

**primrec**

*bt-reflect*(*Lf*) = *Lf*

*bt-reflect*(*Br*(*a*, *l*, *r*)) = *Br*(*a*, *bt-reflect*(*r*), *bt-reflect*(*l*))

**lemma** *bt-reflect-type* [simp]:  $t \in \text{bt}(A) \implies \text{bt-reflect}(t) \in \text{bt}(A)$   
**by** (induct set: *bt*) *auto*

Theorems about *n-leaves*.

**lemma** *n-leaves-reflect*:  $t \in \text{bt}(A) \implies \text{n-leaves}(\text{bt-reflect}(t)) = \text{n-leaves}(t)$   
**by** (induct set: *bt*) (*simp-all add: add-commute*)

**lemma** *n-leaves-nodes*:  $t \in \text{bt}(A) \implies \text{n-leaves}(t) = \text{succ}(\text{n-nodes}(t))$   
**by** (induct set: *bt*) *simp-all*

Theorems about *bt-reflect*.

**lemma** *bt-reflect-bt-reflect-ident*:  $t \in \text{bt}(A) \implies \text{bt-reflect}(\text{bt-reflect}(t)) = t$   
**by** (induct set: *bt*) *simp-all*

**end**

## 3 Terms over an alphabet

**theory** *Term* **imports** *ZF* **begin**

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

**consts**

*term* ::  $i \Rightarrow i$

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

**monos** *list-mono*

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

**declare** *Apply* [*TC*]

**definition**

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

*term-rec*(*t*, *d*)  $\equiv$

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

**definition**

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

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

**definition**

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

**definition**

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

**definition**

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

**definition**

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

**lemma**  $term\text{-}unfold$ :  $term(A) = A * list(term(A))$   
**by** ( $fast\ intro!$ :  $term.intros$  [ $unfolded\ term.con\text{-}defs$ ]  
 $elim$ :  $term.cases$  [ $unfolded\ term.con\text{-}defs$ ])

**lemma**  $term\text{-}induct2$ :

$\llbracket t \in term(A);$   
 $\quad \bigwedge x. \llbracket x \in A \rrbracket \implies P(Apply(x, Nil));$   
 $\quad \bigwedge x \text{ } z \text{ } zs. \llbracket x \in A; \ z \in term(A); \ zs: list(term(A)); \ P(Apply(x, zs))$   
 $\rrbracket \implies P(Apply(x, Cons(z, zs)))$   
 $\rrbracket \implies P(t)$   
 — Induction on  $term(A)$  followed by induction on  $list$ .  
**apply** ( $induct\text{-}tac\ t$ )  
**apply** ( $erule\ list.induct$ )  
**apply** ( $auto\ dest: list\text{-}CollectD$ )  
**done**

**lemma**  $term\text{-}induct\text{-}eqn$  [ $consumes\ 1$ ,  $case\text{-}names\ Apply$ ]:

$\llbracket t \in term(A);$   
 $\quad \bigwedge x \text{ } zs. \llbracket x \in A; \ zs: list(term(A)); \ map(f, zs) = map(g, zs) \rrbracket \implies$   
 $\quad \quad \quad f(Apply(x, zs)) = g(Apply(x, zs))$   
 $\rrbracket \implies f(t) = g(t)$   
 — Induction on  $term(A)$  to prove an equation.  
**apply** ( $induct\text{-}tac\ t$ )  
**apply** ( $auto\ dest: map\text{-}list\text{-}Collect\ list\text{-}CollectD$ )  
**done**

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

**lemma**  $term\text{-}mono$ :  $A \subseteq B \implies term(A) \subseteq term(B)$   
**unfolding**  $term.defs$   
**apply** ( $rule\ lfp\text{-}mono$ )  
**apply** ( $rule\ term.bnd\text{-}mono$ )  
**+**

**apply** (*rule univ-mono basic-monos* | *assumption*) +  
**done**

**lemma** *term-univ*:  $\text{term}(\text{univ}(A)) \subseteq \text{univ}(A)$   
— Easily provable by induction also  
**unfolding** *term.defs term.con-defs*  
**apply** (*rule lfp-lowerbound*)  
**apply** (*rule-tac* [2] *A-subset-univ [THEN univ-mono]*)  
**apply** *safe*  
**apply** (*assumption* | *rule Pair-in-univ list-univ [THEN subsetD]*) +  
**done**

**lemma** *term-subset-univ*:  $A \subseteq \text{univ}(B) \implies \text{term}(A) \subseteq \text{univ}(B)$   
**apply** (*rule subset-trans*)  
**apply** (*erule term-mono*)  
**apply** (*rule term-univ*)  
**done**

**lemma** *term-into-univ*:  $\llbracket t \in \text{term}(A); A \subseteq \text{univ}(B) \rrbracket \implies t \in \text{univ}(B)$   
**by** (*rule term-subset-univ [THEN subsetD]*)

*term-rec* – by *Vset* recursion.

**lemma** *map-lemma*:  $\llbracket l \in \text{list}(A); \text{Ord}(i); \text{rank}(l) < i \rrbracket$   
 $\implies \text{map}(\lambda z. (\lambda x \in \text{Vset}(i). h(x)) \text{ ‘ } z, l) = \text{map}(h, l)$   
— *map* works correctly on the underlying list of terms.  
**apply** (*induct set: list*)  
**apply** *simp*  
**apply** (*subgoal-tac rank (a) < i  $\wedge$  rank (l) < i*)  
**apply** (*simp add: rank-of-Ord*)  
**apply** (*simp add: list.con-defs*)  
**apply** (*blast dest: rank-rls [THEN lt-trans]*)  
**done**

**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*.  
**apply** (*rule term-rec-def [THEN def-Vrec, THEN trans]*)  
**unfolding** *term.con-defs*  
**apply** (*simp add: rank-pair2 map-lemma*)  
**done**

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



```

apply induct
apply (frule list-CollectD)
apply (subst term-rec)
  apply (assumption | rule a) +
apply (erule list.induct)
apply auto
done

```

```

lemma def-term-rec:
   $\llbracket \bigwedge t. j(t) \equiv \text{term-rec}(t, d); \text{ } ts: \text{list}(A) \rrbracket \implies$ 
   $j(\text{Apply}(a, ts)) = d(a, ts, \text{map}(\lambda Z. j(Z), ts))$ 
apply (simp only:)
apply (erule term-rec)
done

```

```

lemma term-rec-simple-type [TC]:
   $\llbracket t \in \text{term}(A);$ 
   $\bigwedge x \text{ } zs \text{ } r. \llbracket x \in A; \text{ } zs: \text{list}(\text{term}(A)); \text{ } r \in \text{list}(C) \rrbracket$ 
   $\implies d(x, zs, r): C$ 
 $\rrbracket \implies \text{term-rec}(t, d) \in C$ 
apply (erule term-rec-type)
apply (drule subset-refl [THEN UN-least, THEN list-mono, THEN subsetD])
apply simp
done

```

*term-map.*

```

lemma term-map [simp]:
   $ts \in \text{list}(A) \implies$ 
   $\text{term-map}(f, \text{Apply}(a, ts)) = \text{Apply}(f(a), \text{map}(\text{term-map}(f), ts))$ 
by (rule term-map-def [THEN def-term-rec])

```

```

lemma term-map-type [TC]:
   $\llbracket t \in \text{term}(A); \bigwedge x. x \in A \implies f(x): B \rrbracket \implies \text{term-map}(f, t) \in \text{term}(B)$ 
unfolding term-map-def
apply (erule term-rec-simple-type)
apply fast
done

```

```

lemma term-map-type2 [TC]:
   $t \in \text{term}(A) \implies \text{term-map}(f, t) \in \text{term}(\{f(u). u \in A\})$ 
apply (erule term-map-type)
apply (erule RepFunI)
done

```

*term-size.*

```

lemma term-size [simp]:
   $ts \in \text{list}(A) \implies \text{term-size}(\text{Apply}(a, ts)) = \text{succ}(\text{list-add}(\text{map}(\text{term-size}, ts)))$ 
by (rule term-size-def [THEN def-term-rec])

```

**lemma** *term-size-type* [TC]:  $t \in \text{term}(A) \implies \text{term-size}(t) \in \text{nat}$   
**by** (*auto simp add: term-size-def*)

*reflect.*

**lemma** *reflect* [simp]:  
 $ts \in \text{list}(A) \implies \text{reflect}(\text{Apply}(a, ts)) = \text{Apply}(a, \text{rev}(\text{map}(\text{reflect}, ts)))$   
**by** (*rule reflect-def [THEN def-term-rec]*)

**lemma** *reflect-type* [TC]:  $t \in \text{term}(A) \implies \text{reflect}(t) \in \text{term}(A)$   
**by** (*auto simp add: reflect-def*)

*preorder.*

**lemma** *preorder* [simp]:  
 $ts \in \text{list}(A) \implies \text{preorder}(\text{Apply}(a, ts)) = \text{Cons}(a, \text{flat}(\text{map}(\text{preorder}, ts)))$   
**by** (*rule preorder-def [THEN def-term-rec]*)

**lemma** *preorder-type* [TC]:  $t \in \text{term}(A) \implies \text{preorder}(t) \in \text{list}(A)$   
**by** (*simp add: preorder-def*)

*postorder.*

**lemma** *postorder* [simp]:  
 $ts \in \text{list}(A) \implies \text{postorder}(\text{Apply}(a, ts)) = \text{flat}(\text{map}(\text{postorder}, ts)) @ [a]$   
**by** (*rule postorder-def [THEN def-term-rec]*)

**lemma** *postorder-type* [TC]:  $t \in \text{term}(A) \implies \text{postorder}(t) \in \text{list}(A)$   
**by** (*simp add: postorder-def*)

Theorems about *term-map*.

**declare** *map-compose* [simp]

**lemma** *term-map-ident*:  $t \in \text{term}(A) \implies \text{term-map}(\lambda u. u, t) = t$   
**by** (*induct rule: term-induct-eqn simp*)

**lemma** *term-map-compose*:  
 $t \in \text{term}(A) \implies \text{term-map}(f, \text{term-map}(g, t)) = \text{term-map}(\lambda u. f(g(u)), t)$   
**by** (*induct rule: term-induct-eqn simp*)

**lemma** *term-map-reflect*:  
 $t \in \text{term}(A) \implies \text{term-map}(f, \text{reflect}(t)) = \text{reflect}(\text{term-map}(f, t))$   
**by** (*induct rule: term-induct-eqn (simp add: rev-map-distrib [symmetric])*)

Theorems about *term-size*.

**lemma** *term-size-term-map*:  $t \in \text{term}(A) \implies \text{term-size}(\text{term-map}(f, t)) = \text{term-size}(t)$   
**by** (*induct rule: term-induct-eqn simp*)

**lemma** *term-size-reflect*:  $t \in \text{term}(A) \implies \text{term-size}(\text{reflect}(t)) = \text{term-size}(t)$   
**by** (*induct rule*: *term-induct-eqn*) (*simp add*: *rev-map-distrib [symmetric] list-add-rev*)

**lemma** *term-size-length*:  $t \in \text{term}(A) \implies \text{term-size}(t) = \text{length}(\text{preorder}(t))$   
**by** (*induct rule*: *term-induct-eqn*) (*simp add*: *length-flat*)

Theorems about *reflect*.

**lemma** *reflect-reflect-ident*:  $t \in \text{term}(A) \implies \text{reflect}(\text{reflect}(t)) = t$   
**by** (*induct rule*: *term-induct-eqn*) (*simp add*: *rev-map-distrib*)

Theorems about *preorder*.

**lemma** *preorder-term-map*:  
 $t \in \text{term}(A) \implies \text{preorder}(\text{term-map}(f, t)) = \text{map}(f, \text{preorder}(t))$   
**by** (*induct rule*: *term-induct-eqn*) (*simp add*: *map-flat*)

**lemma** *preorder-reflect-eq-rev-postorder*:  
 $t \in \text{term}(A) \implies \text{preorder}(\text{reflect}(t)) = \text{rev}(\text{postorder}(t))$   
**by** (*induct rule*: *term-induct-eqn*)  
(*simp add*: *rev-app-distrib rev-flat rev-map-distrib [symmetric]*)

**end**

## 4 Datatype definition n-ary branching trees

**theory** *Ntree* **imports** *ZF* **begin**

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

**consts**

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

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

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

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

**definition**

```

ntree-rec :: [[i, i, i] ⇒ i, i] ⇒ i where
ntree-rec(b) ≡
  Vrecursor(λpr. ntree-case(λx h. b(x, h, λi ∈ domain(h). pr'(h'i))))

```

**definition**

```

ntree-copy :: i ⇒ i where
ntree-copy(z) ≡ ntree-rec(λx h r. Branch(x,r), z)

```

*ntree*

**lemma** *ntree-unfold*:  $ntree(A) = A \times (\bigcup n \in nat. n \rightarrow ntree(A))$   
**by** (*blast intro*: *ntree.intros* [*unfolded ntree.con-defs*]  
*elim*: *ntree.cases* [*unfolded ntree.con-defs*])

**lemma** *ntree-induct* [*consumes 1*, *case-names Branch*, *induct set*: *ntree*]:  
**assumes** *t*:  $t \in ntree(A)$   
**and step**:  $\bigwedge x n h. \llbracket x \in A; n \in nat; h \in n \rightarrow ntree(A); \forall i \in n. P(h'i) \rrbracket \implies P(Branch(x,h))$   
**shows**  $P(t)$   
— A nicer induction rule than the standard one.  
**using** *t*  
**apply** *induct*  
**apply** (*erule UN-E*)  
**apply** (*assumption* | *rule step*) +  
**apply** (*fast elim*: *fun-weaken-type*)  
**apply** (*fast dest*: *apply-type*)  
**done**

**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**:  $\bigwedge x n h. \llbracket x \in A; n \in nat; h \in n \rightarrow ntree(A); f \circ h = g \circ h \rrbracket \implies f \circ Branch(x,h) = g \circ Branch(x,h)$   
**shows**  $f \circ t = g \circ t$   
— Induction on *ntree(A)* to prove an equation  
**using** *t*  
**apply** *induct*  
**apply** (*assumption* | *rule step*) +  
**apply** (*insert f g*)  
**apply** (*rule fun-extension*)  
**apply** (*assumption* | *rule comp-fun*) +  
**apply** (*simp add*: *comp-fun-apply*)  
**done**

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

**lemma** *ntree-mono*:  $A \subseteq B \implies ntree(A) \subseteq ntree(B)$   
**unfolding** *ntree.defs*  
**apply** (*rule lfp-mono*)

```

apply (rule ntree.bnd-mono)+
apply (assumption | rule univ-mono basic-monos)+
done

```

```

lemma ntree-univ:  $ntree(univ(A)) \subseteq univ(A)$ 
— Easily provable by induction also
unfolding ntree.defs ntree.con-defs
apply (rule lfp-lowerbound)
apply (rule-tac [2] A-subset-univ [THEN univ-mono])
apply (blast intro: Pair-in-univ nat-fun-univ [THEN subsetD])
done

```

```

lemma ntree-subset-univ:  $A \subseteq univ(B) \implies ntree(A) \subseteq univ(B)$ 
by (rule subset-trans [OF ntree-mono ntree-univ])

```

*ntree* recursion.

```

lemma ntree-rec-Branch:
  function(h)  $\implies$ 
     $ntree-rec(b, Branch(x, h)) = b(x, h, \lambda i \in domain(h). ntree-rec(b, h'i))$ 
apply (rule ntree-rec-def [THEN def-Vrecursor, THEN trans])
apply (simp add: ntree.con-defs rank-pair2 [THEN [2] lt-trans] rank-apply)
done

```

```

lemma ntree-copy-Branch [simp]:
  function(h)  $\implies$ 
     $ntree-copy (Branch(x, h)) = Branch(x, \lambda i \in domain(h). ntree-copy (h'i))$ 
by (simp add: ntree-copy-def ntree-rec-Branch)

```

```

lemma ntree-copy-is-ident:  $z \in ntree(A) \implies ntree-copy(z) = z$ 
by (induct z set: ntree)
  (auto simp add: domain-of-fun Pi-Collect-iff fun-is-function)

```

*maptree*

```

lemma maptree-unfold:  $maptree(A) = A \times (maptree(A) -||> maptree(A))$ 
by (fast intro!: maptree.intros [unfolded maptree.con-defs])
  (elim: maptree.cases [unfolded maptree.con-defs])

```

```

lemma maptree-induct [consumes 1, induct set: maptree]:
  assumes t:  $t \in maptree(A)$ 
  and step:  $\bigwedge x \ n \ h. \llbracket x \in A; \ h \in maptree(A) -||> maptree(A);$ 
     $\forall y \in field(h). P(y)$ 
]]  $\implies P(Sons(x, h))$ 
shows  $P(t)$ 
— A nicer induction rule than the standard one.
using t
apply induct
apply (assumption | rule step)+
apply (erule Collect-subset [THEN FiniteFun-mono1, THEN subsetD])

```

```

apply (drule FiniteFun.dom-subset [THEN subsetD])
apply (drule Fin.dom-subset [THEN subsetD])
apply fast
done

maptree2

lemma maptree2-unfold:  $\text{maptree2}(A, B) = A \times (B -||> \text{maptree2}(A, B))$ 
  by (fast intro!: maptree2.intros [unfolded maptree2.con-defs]
    elim: maptree2.cases [unfolded maptree2.con-defs])

lemma maptree2-induct [consumes 1, induct set: maptree2]:
  assumes  $t: t \in \text{maptree2}(A, B)$ 
  and step:  $\bigwedge x \ n \ h. \llbracket x \in A; \ h \in B -||> \text{maptree2}(A, B); \ \forall y \in \text{range}(h). \ P(y) \rrbracket \implies P(\text{Sons2}(x, h))$ 
  shows  $P(t)$ 
  using  $t$ 
  apply induct
  apply (assumption | rule step)+
  apply (erule FiniteFun-mono [OF subset-refl Collect-subset, THEN subsetD])
  apply (drule FiniteFun.dom-subset [THEN subsetD])
  apply (drule Fin.dom-subset [THEN subsetD])
  apply fast
  done

end

```

## 5 Trees and forests, a mutually recursive type definition

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

### 5.1 Datatype definition

```

consts
  tree ::  $i \Rightarrow i$ 
  forest ::  $i \Rightarrow i$ 
  tree-forest ::  $i \Rightarrow i$ 

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

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

```

```

for  $t$   $f$ 

declare tree-forest.intros [simp, TC]

lemma tree-def:  $tree(A) \equiv Part(tree-forest(A), Inl)$ 
  by (simp only: tree-forest.defs)

lemma forest-def:  $forest(A) \equiv Part(tree-forest(A), Inr)$ 
  by (simp only: tree-forest.defs)

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

lemma tree-subset-TF:  $tree(A) \subseteq tree-forest(A)$ 
  unfolding tree-forest.defs
  apply (rule Part-subset)
  done

lemma treeI [TC]:  $x \in tree(A) \implies x \in tree-forest(A)$ 
  by (rule tree-subset-TF [THEN subsetD])

lemma forest-subset-TF:  $forest(A) \subseteq tree-forest(A)$ 
  unfolding tree-forest.defs
  apply (rule Part-subset)
  done

lemma treeI' [TC]:  $x \in forest(A) \implies x \in tree-forest(A)$ 
  by (rule forest-subset-TF [THEN subsetD])

lemma TF-equals-Un:  $tree(A) \cup forest(A) = tree-forest(A)$ 
  apply (insert tree-subset-TF forest-subset-TF)
  apply (auto intro!: equalityI tree-forest.intros elim: tree-forest.cases)
  done

lemma tree-forest-unfold:
   $tree-forest(A) = (A \times forest(A)) + (\{0\} + tree(A) \times forest(A))$ 
  — NOT useful, but interesting ...
  supply rews = tree-forest.con-defs tree-def forest-def
  unfolding tree-def forest-def
  apply (fast intro!: tree-forest.intros [unfolded rews, THEN PartD1])
  elim: tree-forest.cases [unfolded rews]
  done

lemma tree-forest-unfold':
   $tree-forest(A) =$ 
   $A \times Part(tree-forest(A), \lambda w. Inr(w)) +$ 
   $\{0\} + Part(tree-forest(A), \lambda w. Inl(w)) * Part(tree-forest(A), \lambda w. Inr(w))$ 
  by (rule tree-forest-unfold [unfolded tree-def forest-def])

lemma tree-unfold:  $tree(A) = \{Inl(x). x \in A \times forest(A)\}$ 
  unfolding tree-def forest-def

```

```

apply (rule Part-Inl [THEN subst])
apply (rule tree-forest-unfold' [THEN subst-context])
done

```

```

lemma forest-unfold: forest(A) = {Inr(x). x ∈ {0} + tree(A)*forest(A)}
  unfolding tree-def forest-def
  apply (rule Part-Inr [THEN subst])
  apply (rule tree-forest-unfold' [THEN subst-context])
done

```

Type checking for recursor: Not needed; possibly interesting?

```

lemma TF-rec-type:
  ⌊z ∈ tree-forest(A);
    ∧ x f r. ⌊x ∈ A; f ∈ forest(A); r ∈ C(f)
  ⌋ ⇒ b(x,f,r) ∈ C(Tcons(x,f));
    c ∈ C(Fnil);
    ∧ t f r1 r2. ⌊t ∈ tree(A); f ∈ forest(A); r1 ∈ C(t); r2 ∈ C(f)
  ⌋ ⇒ d(t,f,r1,r2) ∈ C(Fcons(t,f))
  ⌋ ⇒ tree-forest-rec(b,c,d,z) ∈ C(z)
  by (induct-tac z) simp-all

```

```

lemma tree-forest-rec-type:
  ⌊∧ x f r. ⌊x ∈ A; f ∈ forest(A); r ∈ D(f)
  ⌋ ⇒ b(x,f,r) ∈ C(Tcons(x,f));
    c ∈ D(Fnil);
    ∧ t f r1 r2. ⌊t ∈ tree(A); f ∈ forest(A); r1 ∈ C(t); r2 ∈ D(f)
  ⌋ ⇒ d(t,f,r1,r2) ∈ D(Fcons(t,f))
  ⌋ ⇒ (∀ t ∈ tree(A). tree-forest-rec(b,c,d,t) ∈ C(t)) ∧
    (∀ f ∈ forest(A). tree-forest-rec(b,c,d,f) ∈ D(f))
  — Mutually recursive version.
  unfolding Ball-def
  apply (rule tree-forest.mutual-induct)
  apply simp-all
done

```

## 5.2 Operations

**consts**

```

map :: [i ⇒ i, i] ⇒ i
size :: i ⇒ i
preorder :: i ⇒ i
list-of-TF :: i ⇒ i
of-list :: i ⇒ i
reflect :: i ⇒ i

```

**primrec**

```

list-of-TF (Tcons(x,f)) = [Tcons(x,f)]
list-of-TF (Fnil) = []
list-of-TF (Fcons(t,tf)) = Cons (t, list-of-TF(tf))

```



**primrec**

$of\_list([]) = Fnil$   
 $of\_list(Cons(t,l)) = Fcons(t, of\_list(l))$

**primrec**

$map(h, Tcons(x,f)) = Tcons(h(x), map(h,f))$   
 $map(h, Fnil) = Fnil$   
 $map(h, Fcons(t,tf)) = Fcons(map(h, t), map(h, tf))$

**primrec**

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

**primrec**

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

**primrec**

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

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

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

$z \in tree\_forest(A) \implies list\_of\_TF(z) \in list(tree(A))$   
**by** (*induct set: tree-forest*) *simp-all*

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

**by** (*induct set: list*) *simp-all*

*map*.

**lemma**

**assumes**  $\bigwedge x. x \in A \implies h(x) \in B$   
**shows** *map-tree-type*:  $t \in tree(A) \implies map(h,t) \in tree(B)$   
**and** *map-forest-type*:  $f \in forest(A) \implies map(h,f) \in forest(B)$   
**using** *assms*  
**by** (*induct rule: tree'induct forest'induct*) *simp-all*

*size*.

**lemma** *size-type* [TC]:  $z \in tree\_forest(A) \implies size(z) \in nat$

**by** (*induct set: tree-forest*) *simp-all*

*preorder*.

**lemma** *preorder-type* [TC]:  $z \in \text{tree-forest}(A) \implies \text{preorder}(z) \in \text{list}(A)$   
**by** (*induct set: tree-forest*) *simp-all*

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

**lemma** *forest-induct* [*consumes 1, case-names Fnil Fcons*]:  
 $\llbracket f \in \text{forest}(A);$   
 $R(\text{Fnil});$   
 $\bigwedge t f. \llbracket t \in \text{tree}(A); f \in \text{forest}(A); R(f) \rrbracket \implies R(\text{Fcons}(t,f))$   
 $\rrbracket \implies R(f)$   
— Essentially the same as list induction.  
**apply** (*erule tree-forest.mutual-induct*  
[*THEN conjunct2, THEN spec, THEN [2] rev-mp*])  
**apply** (*rule TrueI*)  
**apply** *simp*  
**apply** *simp*  
**done**

**lemma** *forest-iso*:  $f \in \text{forest}(A) \implies \text{of-list}(\text{list-of-TF}(f)) = f$   
**by** (*induct rule: forest-induct*) *simp-all*

**lemma** *tree-list-iso*:  $ts: \text{list}(\text{tree}(A)) \implies \text{list-of-TF}(\text{of-list}(ts)) = ts$   
**by** (*induct set: list*) *simp-all*

Theorems about *map*.

**lemma** *map-ident*:  $z \in \text{tree-forest}(A) \implies \text{map}(\lambda u. u, z) = z$   
**by** (*induct set: tree-forest*) *simp-all*

**lemma** *map-compose*:  
 $z \in \text{tree-forest}(A) \implies \text{map}(h, \text{map}(j, z)) = \text{map}(\lambda u. h(j(u)), z)$   
**by** (*induct set: tree-forest*) *simp-all*

Theorems about *size*.

**lemma** *size-map*:  $z \in \text{tree-forest}(A) \implies \text{size}(\text{map}(h, z)) = \text{size}(z)$   
**by** (*induct set: tree-forest*) *simp-all*

**lemma** *size-length*:  $z \in \text{tree-forest}(A) \implies \text{size}(z) = \text{length}(\text{preorder}(z))$   
**by** (*induct set: tree-forest*) (*simp-all add: length-app*)

Theorems about *preorder*.

**lemma** *preorder-map*:  
 $z \in \text{tree-forest}(A) \implies \text{preorder}(\text{map}(h, z)) = \text{List.map}(h, \text{preorder}(z))$   
**by** (*induct set: tree-forest*) (*simp-all add: map-app-distrib*)

**end**

## 6 Infinite branching datatype definitions

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

### 6.1 The Brouwer ordinals

**consts**

*brouwer* :: *i*

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

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

**monos** *Pi-mono*

**type-intros** *inf-datatype-intros*

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

**by** (*fast intro!*: *brouwer.intros* [*unfolded* *brouwer.con-defs*])

*elim*: *brouwer.cases* [*unfolded* *brouwer.con-defs*])

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

**assumes** *b*: *b*  $\in$  *brouwer*

**and cases**:

*P*(*Zero*)

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

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

**shows** *P*(*b*)

— A nicer induction rule than the standard one.

**using** *b*

**apply** *induct*

**apply** (*rule cases*(1))

**apply** (*erule* (1) *cases*(2))

**apply** (*rule cases*(3))

**apply** (*fast elim*: *fun-weaken-type*)

**apply** (*fast dest*: *apply-type*)

**done**

### 6.2 The Martin-Löf wellordering type

**consts**

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

**datatype**  $\subseteq V$  *from* (*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*) = ( $\sum x \in A. B(x)$   $\rightarrow$  *Well*(*A*, *B*))

**by** (*fast intro!*: *Well.intros* [*unfolded* *Well.con-defs*])

*elim*: *Well.cases* [*unfolded* *Well.con-defs*])

```

lemma Well-induct2 [consumes 1, case-names step]:
  assumes w:  $w \in \text{Well}(A, B)$ 
    and step:  $\bigwedge a f. \llbracket a \in A; f \in B(a) \rightarrow \text{Well}(A, B); \forall y \in B(a). P(f'y) \rrbracket \implies$ 
 $P(\text{Sup}(a, f))$ 
  shows  $P(w)$ 
  — A nicer induction rule than the standard one.
  using w
  apply induct
  apply (assumption | rule step) +
  apply (fast elim: fun-weaken-type)
  apply (fast dest: apply-type)
  done

lemma Well-bool-unfold:  $\text{Well}(\text{bool}, \lambda x. x) = 1 + (1 \rightarrow \text{Well}(\text{bool}, \lambda x. x))$ 
  — In fact it's isomorphic to nat, but we need a recursion operator
  — for Well to prove this.
  apply (rule Well-unfold [THEN trans])
  apply (simp add: Sigma-bool succ-def)
  done

end

```

## 7 The Mutilated Chess Board Problem, formalized inductively

**theory** *Mutil* **imports** *ZF* **begin**

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

**consts**

*domino* :: *i*  
*tiling* :: *i*  $\Rightarrow$  *i*

**inductive**

**domains** *domino*  $\subseteq \text{Pow}(\text{nat} \times \text{nat})$

**intros**

*horiz*:  $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \{\langle i, j \rangle, \langle i, \text{succ}(j) \rangle\} \in \text{domino}$

*vertl*:  $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \{\langle i, j \rangle, \langle \text{succ}(i), j \rangle\} \in \text{domino}$

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

**inductive**

**domains** *tiling*(*A*)  $\subseteq \text{Pow}(\bigcup(A))$

**intros**

*empty*:  $0 \in \text{tiling}(A)$

*Un*:  $\llbracket a \in A; t \in \text{tiling}(A); a \cap t = 0 \rrbracket \implies a \cup t \in \text{tiling}(A)$

**type-intros** *empty-subsetI Union-upper Un-least PowI*

**type-elims** *PowD* [*elim-format*]

**definition**

$evnodd :: [i, i] \Rightarrow i$  **where**  
 $evnodd(A, b) \equiv \{z \in A. \exists i j. z = \langle i, j \rangle \wedge (i \# + j) \bmod 2 = b\}$

**7.1 Basic properties of evnodd**

**lemma** *evnodd-iff*:  $\langle i, j \rangle : evnodd(A, b) \longleftrightarrow \langle i, j \rangle : A \wedge (i \# + j) \bmod 2 = b$   
**by** (*unfold evnodd-def*) *blast*

**lemma** *evnodd-subset*:  $evnodd(A, b) \subseteq A$   
**by** (*unfold evnodd-def*) *blast*

**lemma** *Finite-evnodd*:  $Finite(X) \Longrightarrow Finite(evnodd(X, b))$   
**by** (*rule lepoll-Finite, rule subset-imp-lepoll, rule evnodd-subset*)

**lemma** *evnodd-Un*:  $evnodd(A \cup B, b) = evnodd(A, b) \cup evnodd(B, b)$   
**by** (*simp add: evnodd-def Collect-Un*)

**lemma** *evnodd-Diff*:  $evnodd(A - B, b) = evnodd(A, b) - evnodd(B, b)$   
**by** (*simp add: evnodd-def Collect-Diff*)

**lemma** *evnodd-cons* [*simp*]:  
 $evnodd(cons(\langle i, j \rangle, C), b) =$   
 $(if (i \# + j) \bmod 2 = b then cons(\langle i, j \rangle, evnodd(C, b)) else evnodd(C, b))$   
**by** (*simp add: evnodd-def Collect-cons*)

**lemma** *evnodd-0* [*simp*]:  $evnodd(0, b) = 0$   
**by** (*simp add: evnodd-def*)

**7.2 Dominoes**

**lemma** *domino-Finite*:  $d \in domino \Longrightarrow Finite(d)$   
**by** (*blast intro!: Finite-cons Finite-0 elim: domino.cases*)

**lemma** *domino-singleton*:  
 $\llbracket d \in domino; b < 2 \rrbracket \Longrightarrow \exists i' j'. evnodd(d, b) = \{\langle i', j' \rangle\}$   
**apply** (*erule domino.cases*)  
**apply** (*rule-tac [2] k1 = i# + j in mod2-cases [THEN disjE]*)  
**apply** (*rule-tac k1 = i# + j in mod2-cases [THEN disjE]*)  
**apply** (*rule add-type | assumption*) +  
**apply** (*auto simp add: mod-succ succ-neq-self dest: ltD*)  
**done**

**7.3 Tilings**

The union of two disjoint tilings is a tiling

**lemma** *tiling-UnI*:

```

     $t \in \text{tiling}(A) \implies u \in \text{tiling}(A) \implies t \cap u = 0 \implies t \cup u \in \text{tiling}(A)$ 
  apply (induct set: tiling)
  apply (simp add: tiling.intros)
  apply (simp add: Un-assoc subset-empty-iff [THEN iff-sym])
  apply (blast intro: tiling.intros)
done

lemma tiling-domino-Finite:  $t \in \text{tiling}(\text{domino}) \implies \text{Finite}(t)$ 
  apply (induct set: tiling)
  apply (rule Finite-0)
  apply (blast intro!: Finite-Un intro: domino-Finite)
done

lemma tiling-domino-0-1:  $t \in \text{tiling}(\text{domino}) \implies |\text{evnodd}(t,0)| = |\text{evnodd}(t,1)|$ 
  apply (induct set: tiling)
  apply (simp add: evnodd-def)
  apply (rule-tac b1 = 0 in domino-singleton [THEN exE])
  prefer 2
  apply simp
  apply assumption
  apply (rule-tac b1 = 1 in domino-singleton [THEN exE])
  prefer 2
  apply simp
  apply assumption
  apply safe
  apply (subgoal-tac  $\forall p b. p \in \text{evnodd}(a,b) \longrightarrow p \notin \text{evnodd}(t,b)$ )
  apply (simp add: evnodd-Un Un-cons tiling-domino-Finite
    evnodd-subset [THEN subset-Finite] Finite-imp-cardinal-cons)
  apply (blast dest!: evnodd-subset [THEN subsetD] elim: equalityE)
done

lemma dominoes-tile-row:
   $\llbracket i \in \text{nat}; n \in \text{nat} \rrbracket \implies \{i\} * (n \# + n) \in \text{tiling}(\text{domino})$ 
  apply (induct-tac n)
  apply (simp add: tiling.intros)
  apply (simp add: Un-assoc [symmetric] Sigma-succ2)
  apply (rule tiling.intros)
  prefer 2 apply assumption
  apply (rename-tac n')
  apply (subgoal-tac
     $\{i\} * \{\text{succ}(n' \# + n')\} \cup \{i\} * \{n' \# + n'\} =$ 
 $\{<i, n' \# + n'>, <i, \text{succ}(n' \# + n')>\}$ )
  prefer 2 apply blast
  apply (simp add: domino.horiz)
  apply (blast elim: mem-irrefl mem-asym)
done

lemma dominoes-tile-matrix:
   $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies m * (n \# + n) \in \text{tiling}(\text{domino})$ 

```

```

apply (induct-tac m)
  apply (simp add: tiling.intros)
  apply (simp add: Sigma-succ1)
  apply (blast intro: tiling-UnI dominoes-tile-row elim: mem-irrefl)
done

```

```

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

```

```

theorem mutil-not-tiling:  $\llbracket m \in \text{nat}; n \in \text{nat};$ 
   $t = (\text{succ}(m)\# + \text{succ}(m)) * (\text{succ}(n)\# + \text{succ}(n));$ 
   $t' = t - \{ \langle 0, 0 \rangle \} - \{ \langle \text{succ}(m\# + m), \text{succ}(n\# + n) \rangle \}$ 
 $\implies t' \notin \text{tiling}(\text{domino})$ 
apply (rule notI)
apply (drule tiling-domino-0-1)
apply (erule-tac  $x = |A|$  for A in eq-lt-E)
apply (subgoal-tac  $t \in \text{tiling}(\text{domino})$ )
prefer 2
apply (simp only: nat-succI add-type dominoes-tile-matrix)
apply (simp add: evnodd-Diff mod2-add-self mod2-succ-succ
  tiling-domino-0-1 [symmetric])
apply (rule lt-trans)
apply (rule Finite-imp-cardinal-Diff,
  simp add: tiling-domino-Finite Finite-evnodd Finite-Diff,
  simp add: evnodd-iff nat-0-le [THEN ltD] mod2-add-self) +
done

```

**end**

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

```

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

```

**inductive**

```

domains fold-set(A, B, f, e)  $\subseteq \text{Fin}(A) * B$ 

```

**intros**

```

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

```

```

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

```

**type-intros** *Fin.intros*

**definition**

```

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

```

**definition**

```

setsum ::  $[i \Rightarrow i, i] \Rightarrow i$  where
setsum(g, C)  $\equiv \text{if } \text{Finite}(C) \text{ then}$ 

```

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

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

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

**lemma** *cons-lemma1*:  $\llbracket x \notin C; x \notin B \rrbracket \implies cons(x, B) = cons(x, C) \longleftrightarrow B = C$   
**by** (*auto elim: equalityE*)

**lemma** *cons-lemma2*:  $\llbracket cons(x, B) = cons(y, C); x \neq y; x \notin B; y \notin C \rrbracket$   
 $\implies B - \{y\} = C - \{x\} \wedge x \in C \wedge y \in B$   
**apply** (*auto elim: equalityE*)  
**done**

**lemma** *fold-set-mono-lemma*:  
 $\langle C, x \rangle \in fold\text{-}set(A, B, f, e)$   
 $\implies \forall D. A \leq D \longrightarrow \langle C, x \rangle \in fold\text{-}set(D, B, f, e)$   
**apply** (*erule fold-set.induct*)  
**apply** (*auto intro: fold-set.intros*)  
**done**

**lemma** *fold-set-mono*:  $C \leq A \implies fold\text{-}set(C, B, f, e) \subseteq fold\text{-}set(A, B, f, e)$   
**apply** *clarify*  
**apply** (*frule fold-set.dom-subset [THEN subsetD], clarify*)  
**apply** (*auto dest: fold-set-mono-lemma*)  
**done**

**lemma** *fold-set-lemma*:  
 $\langle C, x \rangle \in fold\text{-}set(A, B, f, e) \implies \langle C, x \rangle \in fold\text{-}set(C, B, f, e) \wedge C \leq A$   
**apply** (*erule fold-set.induct*)  
**apply** (*auto intro!: fold-set.intros intro: fold-set-mono [THEN subsetD]*)  
**done**

**lemma** *Diff1-fold-set*:  
 $\llbracket \langle C - \{x\}, y \rangle \in fold\text{-}set(A, B, f, e); x \in C; x \in A; f(x, y) \in B \rrbracket$   
 $\implies \langle C, f(x, y) \rangle \in fold\text{-}set(A, B, f, e)$   
**apply** (*frule fold-set.dom-subset [THEN subsetD]*)  
**apply** (*erule cons-Diff [THEN subst], rule fold-set.intros, auto*)  
**done**

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



**and** *etype* [*intro,simp*]:  $e \in B$   
**and** *fcomm*:  $\llbracket x \in A; y \in A; z \in B \rrbracket \implies f(x, f(y, z)) = f(y, f(x, z))$

**lemma** (*in fold-typing*) *Fin-imp-fold-set*:  
 $C \in \text{Fin}(A) \implies (\exists x. \langle C, x \rangle \in \text{fold-set}(A, B, f, e))$   
**apply** (*erule Fin-induct*)  
**apply** (*auto dest: fold-set.dom-subset [THEN subsetD]*  
*intro: fold-set.intros etype ftype*)  
**done**

**lemma** *Diff-sing-imp*:  
 $\llbracket C - \{b\} = D - \{a\}; a \neq b; b \in C \rrbracket \implies C = \text{cons}(b, D) - \{a\}$   
**by** (*blast elim: equalityE*)

**lemma** (*in fold-typing*) *fold-set-determ-lemma* [*rule-format*]:  
 $n \in \text{nat}$   
 $\implies \forall C. |C| < n \longrightarrow$   
 $(\forall x. \langle C, x \rangle \in \text{fold-set}(A, B, f, e) \longrightarrow$   
 $(\forall y. \langle C, y \rangle \in \text{fold-set}(A, B, f, e) \longrightarrow y = x))$   
**apply** (*erule nat-induct*)  
**apply** (*auto simp add: le-iff*)  
**apply** (*erule fold-set.cases*)  
**apply** (*force elim!: empty-fold-setE*)  
**apply** (*erule fold-set.cases*)  
**apply** (*force elim!: empty-fold-setE, clarify*)

**apply** (*frule-tac a = Ca in fold-set.dom-subset [THEN subsetD, THEN SigmaD1]*)  
**apply** (*frule-tac a = Cb in fold-set.dom-subset [THEN subsetD, THEN SigmaD1]*)  
**apply** (*simp add: Fin-into-Finite [THEN Finite-imp-cardinal-cons]*)  
**apply** (*case-tac x = xb, auto*)  
**apply** (*simp add: cons-lemma1, blast*)

**case**  $x \neq xb$

**apply** (*drule cons-lemma2, safe*)  
**apply** (*frule Diff-sing-imp, assumption+*)

\* LEVEL 17

**apply** (*subgoal-tac |Ca|  $\leq$  |Cb|*)  
**prefer** 2  
**apply** (*rule succ-le-imp-le*)  
**apply** (*simp add: Fin-into-Finite Finite-imp-succ-cardinal-Diff*  
*Fin-into-Finite [THEN Finite-imp-cardinal-cons]*)  
**apply** (*rule-tac C1 = Ca - {xb} in Fin-imp-fold-set [THEN exE]*)  
**apply** (*blast intro: Diff-subset [THEN Fin-subset]*)

\* LEVEL 24 \*

**apply** (*frule Diff1-fold-set, blast, blast*)  
**apply** (*blast dest!: ftype fold-set.dom-subset [THEN subsetD]*)

```

apply (subgoal-tac  $ya = f(xb, xa)$  )
  prefer 2 apply (blast del: equalityCE)
apply (subgoal-tac  $\langle Cb - \{x\}, xa \rangle \in \text{fold-set}(A, B, f, e)$ )
  prefer 2 apply simp
apply (subgoal-tac  $yb = f(x, xa)$  )
  apply (drule-tac [2]  $C = Cb$  in Diff1-fold-set, simp-all)
    apply (blast intro: fcomm dest!: fold-set.dom-subset [THEN subsetD])
  apply (blast intro: ftype dest!: fold-set.dom-subset [THEN subsetD], blast)
done

```

```

lemma (in fold-typing) fold-set-determ:
   $\llbracket \langle C, x \rangle \in \text{fold-set}(A, B, f, e);$ 
   $\langle C, y \rangle \in \text{fold-set}(A, B, f, e) \rrbracket \implies y = x$ 
apply (frule fold-set.dom-subset [THEN subsetD], clarify)
apply (drule Fin-into-Finite)
apply (unfold Finite-def, clarify)
apply (rule-tac  $n = \text{succ}(n)$  in fold-set-determ-lemma)
apply (auto intro: eqpoll-imp-lepoll [THEN lepoll-cardinal-le])
done

```

```

lemma (in fold-typing) fold-equality:
   $\langle C, y \rangle \in \text{fold-set}(A, B, f, e) \implies \text{fold}[B](f, e, C) = y$ 
  unfolding fold-def
apply (frule fold-set.dom-subset [THEN subsetD], clarify)
apply (rule the-equality)
  apply (rule-tac [2]  $A = C$  in fold-typing.fold-set-determ)
apply (force dest: fold-set-lemma)
apply (auto dest: fold-set-lemma)
apply (simp add: fold-typing-def, auto)
apply (auto dest: fold-set-lemma intro: ftype etype fcomm)
done

```

```

lemma fold-0 [simp]:  $e \in B \implies \text{fold}[B](f, e, 0) = e$ 
  unfolding fold-def
apply (blast elim!: empty-fold-setE intro: fold-set.intros)
done

```

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

```

lemma (in fold-typing) fold-set-imp-cons:
   $\llbracket \langle C, y \rangle \in \text{fold-set}(C, B, f, e); C \in \text{Fin}(A); c \in A; c \notin C \rrbracket$ 
   $\implies \langle \text{cons}(c, C), f(c, y) \rangle \in \text{fold-set}(\text{cons}(c, C), B, f, e)$ 
apply (frule FinD [THEN fold-set-mono, THEN subsetD])
  apply assumption
apply (frule fold-set.dom-subset [of A, THEN subsetD])
apply (blast intro!: fold-set.consI intro: fold-set-mono [THEN subsetD])
done

```

```

lemma (in fold-typing) fold-cons-lemma [rule-format]:
   $\llbracket C \in \text{Fin}(A); c \in A; c \notin C \rrbracket$ 
     $\implies \langle \text{cons}(c, C), v \rangle \in \text{fold-set}(\text{cons}(c, C), B, f, e) \longleftrightarrow$ 
       $(\exists y. \langle C, y \rangle \in \text{fold-set}(C, B, f, e) \wedge v = f(c, y))$ 
apply auto
prefer 2 apply (blast intro: fold-set-imp-cons)
apply (frule-tac Fin.consI [of c, THEN FinD, THEN fold-set-mono, THEN sub-
setD], assumption+)
apply (frule-tac fold-set.dom-subset [of A, THEN subsetD])
apply (drule FinD)
apply (rule-tac A1 = cons(c,C) and f1=f and B1=B and C1=C and e1=e in
fold-typing.Fin-imp-fold-set [THEN exE])
apply (blast intro: fold-typing.intro ftype etype fcomm)
apply (blast intro: Fin-subset [of - cons(c,C)] Finite-into-Fin
  dest: Fin-into-Finite)
apply (rule-tac x = x in exI)
apply (auto intro: fold-set.intros)
apply (drule-tac fold-set-lemma [of C], blast)
apply (blast intro!: fold-set.consI
  intro: fold-set-determ fold-set-mono [THEN subsetD]
  dest: fold-set.dom-subset [THEN subsetD])
done

lemma (in fold-typing) fold-cons:
   $\llbracket C \in \text{Fin}(A); c \in A; c \notin C \rrbracket$ 
     $\implies \text{fold}[B](f, e, \text{cons}(c, C)) = f(c, \text{fold}[B](f, e, C))$ 
unfolding fold-def
apply (simp add: fold-cons-lemma)
apply (rule the-equality, auto)
apply (subgoal-tac [2]  $\langle C, y \rangle \in \text{fold-set}(A, B, f, e)$ )
apply (drule Fin-imp-fold-set)
apply (auto dest: fold-set-lemma simp add: fold-def [symmetric] fold-equality)
apply (blast intro: fold-set-mono [THEN subsetD] dest!: FinD)
done

lemma (in fold-typing) fold-type [simp, TC]:
   $C \in \text{Fin}(A) \implies \text{fold}[B](f, e, C) : B$ 
apply (erule Fin-induct)
apply (simp-all add: fold-cons ftype etype)
done

lemma (in fold-typing) fold-commute [rule-format]:
   $\llbracket C \in \text{Fin}(A); c \in A \rrbracket$ 
     $\implies (\forall y \in B. f(c, \text{fold}[B](f, y, C)) = \text{fold}[B](f, f(c, y), C))$ 
apply (erule Fin-induct)
apply (simp-all add: fold-typing.fold-cons [of A B - f]
  fold-typing.fold-type [of A B - f]
  fold-typing-def fcomm)
done

```

**lemma** (in *fold-typing*) *fold-nest-Un-Int*:  
 $\llbracket C \in \text{Fin}(A); D \in \text{Fin}(A) \rrbracket$   
 $\implies \text{fold}[B](f, \text{fold}[B](f, e, D), C) =$   
 $\text{fold}[B](f, \text{fold}[B](f, e, (C \cap D)), C \cup D)$   
**apply** (erule *Fin-induct*, auto)  
**apply** (simp add: *Un-cons Int-cons-left fold-type fold-commute*  
*fold-typing.fold-cons [of A - - f]*  
*fold-typing-def fcomm cons-absorb*)  
**done**

**lemma** (in *fold-typing*) *fold-nest-Un-disjoint*:  
 $\llbracket C \in \text{Fin}(A); D \in \text{Fin}(A); C \cap D = \emptyset \rrbracket$   
 $\implies \text{fold}[B](f, e, C \cup D) = \text{fold}[B](f, \text{fold}[B](f, e, D), C)$   
**by** (simp add: *fold-nest-Un-Int*)

**lemma** *Finite-cons-lemma*:  $\text{Finite}(C) \implies C \in \text{Fin}(\text{cons}(c, C))$   
**apply** (erule *Finite-into-Fin*)  
**apply** (blast intro: *Fin-mono [THEN subsetD]*)  
**done**

## 7.4 The Operator *setsum*

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

**lemma** *setsum-cons* [simp]:  
 $\text{Finite}(C) \implies$   
 $\text{setsum}(g, \text{cons}(c, C)) =$   
 $(\text{if } c \in C \text{ then } \text{setsum}(g, C) \text{ else } g(c) \$+ \text{setsum}(g, C))$   
**apply** (auto simp add: *setsum-def Finite-cons cons-absorb*)  
**apply** (rule-tac  $A = \text{cons}(c, C)$  in *fold-typing.fold-cons*)  
**apply** (auto intro: *fold-typing.intro Finite-cons-lemma*)  
**done**

**lemma** *setsum-K0*:  $\text{setsum}((\lambda i. \#0), C) = \#0$   
**apply** (case-tac *Finite* (C) )  
**prefer** 2 **apply** (simp add: *setsum-def*)  
**apply** (erule *Finite-induct*, auto)  
**done**

**lemma** *setsum-Un-Int*:  
 $\llbracket \text{Finite}(C); \text{Finite}(D) \rrbracket$   
 $\implies \text{setsum}(g, C \cup D) \$+ \text{setsum}(g, C \cap D)$   
 $= \text{setsum}(g, C) \$+ \text{setsum}(g, D)$   
**apply** (erule *Finite-induct*)  
**apply** (simp-all add: *Int-cons-right cons-absorb Un-cons Int-commute Finite-Un*  
*Int-lower1 [THEN subset-Finite]*)

done

**lemma** *setsum-type* [*simp*, *TC*]: *setsum*(*g*, *C*):*int*  
**apply** (*case-tac* *Finite* (*C*) )  
**prefer** 2 **apply** (*simp* *add*: *setsum-def*)  
**apply** (*erule* *Finite-induct*, *auto*)  
**done**

**lemma** *setsum-Un-disjoint*:  

$$\llbracket \text{Finite}(C); \text{Finite}(D); C \cap D = 0 \rrbracket$$

$$\implies \text{setsum}(g, C \cup D) = \text{setsum}(g, C) \$+ \text{setsum}(g, D)$$
  
**apply** (*subst* *setsum-Un-Int* [*symmetric*])  
**apply** (*subgoal-tac* [*?*] *Finite* (*C*  $\cup$  *D*) )  
**apply** (*auto* *intro*: *Finite-Un*)  
**done**

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

$$\text{Finite}(I) \implies (\forall i \in I. \text{Finite}(C(i))) \longrightarrow \text{Finite}(\text{RepFun}(I, C))$$
  
**apply** (*erule* *Finite-induct*, *auto*)  
**done**

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

$$\text{Finite}(I)$$

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

$$(\forall i \in I. \forall j \in I. i \neq j \longrightarrow C(i) \cap C(j) = 0) \longrightarrow$$

$$\text{setsum}(f, \bigcup i \in I. C(i)) = \text{setsum}(\lambda i. \text{setsum}(f, C(i)), I)$$
  
**apply** (*erule* *Finite-induct*, *auto*)  
**apply** (*subgoal-tac*  $\forall i \in B. x \neq i$ )  
**prefer** 2 **apply** *blast*  
**apply** (*subgoal-tac*  $C(x) \cap (\bigcup i \in B. C(i)) = 0$ )  
**prefer** 2 **apply** *blast*  
**apply** (*subgoal-tac*  $\text{Finite}(\bigcup i \in B. C(i)) \wedge \text{Finite}(C(x)) \wedge \text{Finite}(B)$ )  
**apply** (*simp* (*no-asm-simp*) *add*: *setsum-Un-disjoint*)  
**apply** (*auto* *intro*: *Finite-Union* *Finite-RepFun*)  
**done**

**lemma** *setsum-addf*:  $\text{setsum}(\lambda x. f(x) \$+ g(x), C) = \text{setsum}(f, C) \$+ \text{setsum}(g, C)$   
**apply** (*case-tac* *Finite* (*C*) )  
**prefer** 2 **apply** (*simp* *add*: *setsum-def*)  
**apply** (*erule* *Finite-induct*, *auto*)  
**done**

**lemma** *fold-set-cong*:  

$$\llbracket A=A'; B=B'; e=e'; (\forall x \in A'. \forall y \in B'. f(x,y) = f'(x,y)) \rrbracket$$

$$\implies \text{fold-set}(A, B, f, e) = \text{fold-set}(A', B', f', e')$$
  
**apply** (*simp* *add*: *fold-set-def*)

**apply** (*intro refl iff-refl lfp-cong Collect-cong disj-cong ex-cong, auto*)  
**done**

**lemma** *fold-cong*:  
 $\llbracket B=B'; A=A'; e=e' \rrbracket \implies \bigwedge x y. \llbracket x \in A'; y \in B' \rrbracket \implies f(x,y) = f'(x,y) \implies$   
 $\text{fold}[B](f,e,A) = \text{fold}[B'](f', e', A')$   
**apply** (*simp add: fold-def*)  
**apply** (*subst fold-set-cong*)  
**apply** (*rule-tac [5] refl, simp-all*)  
**done**

**lemma** *setsum-cong*:  
 $\llbracket A=B; \bigwedge x. x \in B \implies f(x) = g(x) \rrbracket \implies$   
 $\text{setsum}(f, A) = \text{setsum}(g, B)$   
**by** (*simp add: setsum-def cong add: fold-cong*)

**lemma** *setsum-Un*:  
 $\llbracket \text{Finite}(A); \text{Finite}(B) \rrbracket$   
 $\implies \text{setsum}(f, A \cup B) =$   
 $\text{setsum}(f, A) \$+ \text{setsum}(f, B) \$- \text{setsum}(f, A \cap B)$   
**apply** (*subst setsum-Un-Int [symmetric], auto*)  
**done**

**lemma** *setsum-zneg-or-0* [*rule-format (no-asm)*]:  
 $\text{Finite}(A) \implies (\forall x \in A. g(x) \$\leq \#0) \longrightarrow \text{setsum}(g, A) \$\leq \#0$   
**apply** (*erule Finite-induct*)  
**apply** (*auto intro: zneg-or-0-add-zneg-or-0-imp-zneg-or-0*)  
**done**

**lemma** *setsum-succD-lemma* [*rule-format*]:  
 $\text{Finite}(A)$   
 $\implies \forall n \in \text{nat}. \text{setsum}(f, A) = \$\# \text{succ}(n) \longrightarrow (\exists a \in A. \#0 \$< f(a))$   
**apply** (*erule Finite-induct*)  
**apply** (*auto simp del: int-of-0 int-of-succ simp add: not-zless-iff-zle int-of-0 [symmetric]*)  
**apply** (*subgoal-tac setsum (f, B) \\$\leq \#0*)  
**apply** *simp-all*  
**prefer** 2 **apply** (*blast intro: setsum-zneg-or-0*)  
**apply** (*subgoal-tac \\$\# 1 \\$\leq f (x) \\$+ setsum (f, B) )*)  
**apply** (*drule zdiff-zle-iff [THEN iffD2]*)  
**apply** (*subgoal-tac \\$\# 1 \\$\leq \\$\# 1 \\$- setsum (f,B) )*)  
**apply** (*drule-tac x = \\$\# 1 in zle-trans*)  
**apply** (*rule-tac [2] j = \#1 in zless-zle-trans, auto*)  
**done**

**lemma** *setsum-succD*:  
 $\llbracket \text{setsum}(f, A) = \$\# \text{succ}(n); n \in \text{nat} \rrbracket \implies \exists a \in A. \#0 \$< f(a)$   
**apply** (*case-tac Finite (A) )*)

```

apply (blast intro: setsum-succD-lemma)
  unfolding setsum-def
apply (auto simp del: int-of-0 int-of-succ simp add: int-succ-int-1 [symmetric]
int-of-0 [symmetric])
done

lemma g-zpos-imp-setsum-zpos [rule-format]:
   $Finite(A) \implies (\forall x \in A. \#0 \leq g(x)) \longrightarrow \#0 \leq setsum(g, A)$ 
apply (erule Finite-induct)
apply (simp (no-asm))
apply (auto intro: zpos-add-zpos-imp-zpos)
done

lemma g-zpos-imp-setsum-zpos2 [rule-format]:
   $\llbracket Finite(A); \forall x. \#0 \leq g(x) \rrbracket \implies \#0 \leq setsum(g, A)$ 
apply (erule Finite-induct)
apply (auto intro: zpos-add-zpos-imp-zpos)
done

lemma g-zpos-imp-setsum-zspos [rule-format]:
   $Finite(A) \implies (\forall x \in A. \#0 < g(x)) \longrightarrow A \neq 0 \longrightarrow (\#0 < setsum(g, A))$ 
apply (erule Finite-induct)
apply (auto intro: zspos-add-zspos-imp-zspos)
done

lemma setsum-Diff [rule-format]:
   $Finite(A) \implies \forall a. M(a) = \#0 \longrightarrow setsum(M, A) = setsum(M, A - \{a\})$ 
apply (erule Finite-induct)
apply (simp-all add: Diff-cons-eq Finite-Diff)
done

end

```

## 8 The accessible part of a relation

**theory** *Acc imports ZF begin*

Inductive definition of  $acc(r)$ ; see [3].

**consts**

*acc* ::  $i \Rightarrow i$

**inductive**

**domains**  $acc(r) \subseteq field(r)$

**intros**

*vmage*:  $\llbracket r - \{a\}: Pow(acc(r)); a \in field(r) \rrbracket \implies a \in acc(r)$

**monos** *Pow-mono*

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

a proper class!

The intended introduction rule:

**lemma** *accI*:  $\llbracket \bigwedge b. \langle b, a \rangle : r \implies b \in \text{acc}(r); a \in \text{field}(r) \rrbracket \implies a \in \text{acc}(r)$   
**by** (*blast intro: acc.intros*)

**lemma** *acc-downward*:  $\llbracket b \in \text{acc}(r); \langle a, b \rangle : r \rrbracket \implies a \in \text{acc}(r)$   
**by** (*erule acc.cases blast*)

**lemma** *acc-induct* [*consumes 1, case-names vimage, induct set: acc*]:  
 $\llbracket a \in \text{acc}(r);$   
 $\bigwedge x. \llbracket x \in \text{acc}(r); \forall y. \langle y, x \rangle : r \longrightarrow P(y) \rrbracket \implies P(x)$   
 $\rrbracket \implies P(a)$   
**by** (*erule acc.induct*) (*blast intro: acc.intros*)

**lemma** *wf-on-acc*:  $\text{wf}[\text{acc}(r)](r)$   
**apply** (*rule wf-onI2*)  
**apply** (*erule acc-induct*)  
**apply** *fast*  
**done**

**lemma** *acc-wfI*:  $\text{field}(r) \subseteq \text{acc}(r) \implies \text{wf}(r)$   
**by** (*erule wf-on-acc [THEN wf-on-subset-A, THEN wf-on-field-imp-wf]*)

**lemma** *acc-wfD*:  $\text{wf}(r) \implies \text{field}(r) \subseteq \text{acc}(r)$   
**apply** (*rule subsetI*)  
**apply** (*erule wf-induct2, assumption*)  
**apply** (*blast intro: accI*)  
**done**

**lemma** *wf-acc-iff*:  $\text{wf}(r) \longleftrightarrow \text{field}(r) \subseteq \text{acc}(r)$   
**by** (*rule iffI, erule acc-wfD, erule acc-wfI*)

**end**

**theory** *Multiset*  
**imports** *FoldSet Acc*  
**begin**

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

**definition**

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



**definition**

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

**definition**

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

**definition**

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

**definition**

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

**definition**

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

**definition**

$msingle :: i \Rightarrow i$  ( $\langle \langle open-block\ notation = \langle mixfix\ multiset \rangle \{ \# - \# \} \rangle \rangle$ ) **where**  
 $\{ \# a \# \} \equiv \{ \langle a, 1 \rangle \}$

**definition**

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

**definition**

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

**definition**

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

**abbreviation**

$melem :: [i, i] \Rightarrow o$  ( $\langle \langle notation = \langle infix : \# \rangle - / : \# - \rangle [50, 51] 50 \rangle$ ) **where**

$a : \# M \equiv a \in mset-of(M)$

#### **syntax**

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

#### **syntax-consts**

$-MColl \equiv MCollect$

#### **translations**

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

#### **definition**

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

#### **definition**

$multirel :: [i, i] \Rightarrow i \text{ where}$   
 $multirel(A, r) \equiv multirel1(A, r) ^+ +$

#### **definition**

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

#### **definition**

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

#### **definition**

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

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

**lemma** *funrestrict-subset*:  $\llbracket f \in Pi(C, B); A \subseteq C \rrbracket \Longrightarrow funrestrict(f, A) \subseteq f$   
**by** (*auto simp add: funrestrict-def lam-def intro: apply-Pair*)

**lemma** *funrestrict-type*:

$\llbracket \bigwedge x. x \in A \Longrightarrow f'x \in B(x) \rrbracket \Longrightarrow funrestrict(f, A) \in Pi(A, B)$   
**by** (*simp add: funrestrict-def lam-type*)

**lemma** *funrestrict-type2*:  $\llbracket f \in Pi(C, B); A \subseteq C \rrbracket \Longrightarrow funrestrict(f, A) \in Pi(A, B)$   
**by** (*blast intro: apply-type funrestrict-type*)

**lemma** *funrestrict* [*simp*]:  $a \in A \implies \text{funrestrict}(f, A) \text{ ` } a = f'a$   
**by** (*simp add: funrestrict-def*)

**lemma** *funrestrict-empty* [*simp*]:  $\text{funrestrict}(f, 0) = 0$   
**by** (*simp add: funrestrict-def*)

**lemma** *domain-funrestrict* [*simp*]:  $\text{domain}(\text{funrestrict}(f, C)) = C$   
**by** (*auto simp add: funrestrict-def lam-def*)

**lemma** *fun-cons-funrestrict-eq*:  
 $f \in \text{cons}(a, b) \rightarrow B \implies f = \text{cons}(\langle a, f \text{ ` } a \rangle, \text{funrestrict}(f, b))$   
**apply** (*rule equalityI*)  
**prefer** 2 **apply** (*blast intro: apply-Pair funrestrict-subset [THEN subsetD]*)  
**apply** (*auto dest!: Pi-memberD simp add: funrestrict-def lam-def*)  
**done**

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

A useful simplification rule

**lemma** *multiset-fun-iff*:  
 $(f \in A \rightarrow \text{nat} - \{0\}) \longleftrightarrow f \in A \rightarrow \text{nat} \wedge (\forall a \in A. f'a \in \text{nat} \wedge 0 < f'a)$   
**apply** *safe*  
**apply** (*rule-tac B1 = range (f) in Pi-mono [THEN subsetD]*)  
**apply** (*auto intro!: Ord-0-lt*  
 $\text{dest: apply-type Diff-subset [THEN Pi-mono, THEN subsetD]$   
 $\text{simp add: range-of-fun apply-iff}$ )  
**done**

**lemma** *multiset-into-Mult*:  $\llbracket \text{multiset}(M); \text{mset-of}(M) \subseteq A \rrbracket \implies M \in \text{Mult}(A)$   
**apply** (*simp add: multiset-def*)  
**apply** (*auto simp add: multiset-fun-iff mset-of-def*)  
**apply** (*rule-tac B1 = nat - {0} in FiniteFun-mono [THEN subsetD], simp-all*)  
**apply** (*rule Finite-into-Fin [THEN [2] Fin-mono [THEN subsetD], THEN fun-FiniteFunI]*)  
**apply** (*simp-all (no-asm-simp) add: multiset-fun-iff*)  
**done**

**lemma** *Mult-into-multiset*:  $M \in \text{Mult}(A) \implies \text{multiset}(M) \wedge \text{mset-of}(M) \subseteq A$   
**apply** (*simp add: multiset-def mset-of-def*)  
**apply** (*frule FiniteFun-is-fun*)  
**apply** (*drule FiniteFun-domain-Fin*)  
**apply** (*frule FinD, clarify*)  
**apply** (*rule-tac x = domain (M) in exI*)  
**apply** (*blast intro: Fin-into-Finite*)  
**done**

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

**by** (*blast dest: Mult-into-multiset intro: multiset-into-Mult*)

**lemma** *multiset-iff-Mult-mset-of*:  $\text{multiset}(M) \longleftrightarrow M \in \text{Mult}(\text{mset-of}(M))$   
**by** (*auto simp add: Mult-iff-multiset*)

The *multiset* operator

**lemma** *multiset-0* [*simp*]:  $\text{multiset}(0)$   
**by** (*auto intro: FiniteFun.intros simp add: multiset-iff-Mult-mset-of*)

The *mset-of* operator

**lemma** *multiset-set-of-Finite* [*simp*]:  $\text{multiset}(M) \implies \text{Finite}(\text{mset-of}(M))$   
**by** (*simp add: multiset-def mset-of-def, auto*)

**lemma** *mset-of-0* [*iff*]:  $\text{mset-of}(0) = 0$   
**by** (*simp add: mset-of-def*)

**lemma** *mset-is-0-iff*:  $\text{multiset}(M) \implies \text{mset-of}(M)=0 \longleftrightarrow M=0$   
**by** (*auto simp add: multiset-def mset-of-def*)

**lemma** *mset-of-single* [*iff*]:  $\text{mset-of}(\{\#a\}) = \{a\}$   
**by** (*simp add: msingle-def mset-of-def*)

**lemma** *mset-of-union* [*iff*]:  $\text{mset-of}(M +\# N) = \text{mset-of}(M) \cup \text{mset-of}(N)$   
**by** (*simp add: mset-of-def munion-def*)

**lemma** *mset-of-diff* [*simp*]:  $\text{mset-of}(M) \subseteq A \implies \text{mset-of}(M -\# N) \subseteq A$   
**by** (*auto simp add: mdiff-def multiset-def normalize-def mset-of-def*)

**lemma** *msingle-not-0* [*iff*]:  $\{\#a\} \neq 0 \wedge 0 \neq \{\#a\}$   
**by** (*simp add: msingle-def*)

**lemma** *msingle-eq-iff* [*iff*]:  $(\{\#a\} = \{\#b\}) \longleftrightarrow (a = b)$   
**by** (*simp add: msingle-def*)

**lemma** *msingle-multiset* [*iff, TC*]:  $\text{multiset}(\{\#a\})$   
**apply** (*simp add: multiset-def msingle-def*)  
**apply** (*rule-tac x = \{a\} in exI*)  
**apply** (*auto intro: Finite-cons Finite-0 fun-extend3*)  
**done**

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

**lemma** *normalize-idem* [*simp*]:  $\text{normalize}(\text{normalize}(f)) = \text{normalize}(f)$   
**apply** (*simp add: normalize-def funrestrict-def mset-of-def*)  
**apply** (*case-tac  $\exists A. f \in A \rightarrow \text{nat} \wedge \text{Finite}(A)$* )

```

apply clarify
apply (drule-tac  $x = \{x \in \text{domain } (f) . 0 < f \text{ ' } x\}$  in spec)
apply auto
apply (auto intro!: lam-type simp add: Collect-Finite)
done

```

```

lemma normalize-multiset [simp]:  $\text{multiset}(M) \implies \text{normalize}(M) = M$ 
by (auto simp add: multiset-def normalize-def mset-of-def funrestrict-def multiset-fun-iff)

```

```

lemma multiset-normalize [simp]:  $\text{multiset}(\text{normalize}(f))$ 
apply (simp add: normalize-def)
apply (simp add: normalize-def mset-of-def multiset-def, auto)
apply (rule-tac  $x = \{x \in A . 0 < f \text{ ' } x\}$  in exI)
apply (auto intro: Collect-subset [THEN subset-Finite] funrestrict-type)
done

```

```

lemma munion-multiset [simp]:  $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies \text{multiset}(M \text{ +\# } N)$ 
apply (unfold multiset-def munion-def mset-of-def, auto)
apply (rule-tac  $x = A \cup Aa$  in exI)
apply (auto intro!: lam-type intro: Finite-Un simp add: multiset-fun-iff zero-less-add)
done

```

```

lemma mdiff-multiset [simp]:  $\text{multiset}(M \text{ -\# } N)$ 
by (simp add: mdiff-def)

```

```

lemma munion-0 [simp]:  $\text{multiset}(M) \implies M \text{ +\# } 0 = M \wedge 0 \text{ +\# } M = M$ 
apply (simp add: multiset-def)
apply (auto simp add: munion-def mset-of-def)
done

```

```

lemma munion-commute:  $M \text{ +\# } N = N \text{ +\# } M$ 
by (auto intro!: lam-cong simp add: munion-def)

```

```

lemma munion-assoc:  $(M \text{ +\# } N) \text{ +\# } K = M \text{ +\# } (N \text{ +\# } K)$ 
unfolding munion-def mset-of-def
apply (rule lam-cong, auto)
done

```

**lemma** *munion-lcommute*:  $M +\# (N +\# K) = N +\# (M +\# K)$   
**unfolding** *munion-def mset-of-def*  
**apply** (*rule lam-cong, auto*)  
**done**

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

**lemma** *mdiff-self-eq-0* [*simp*]:  $M -\# M = 0$   
**by** (*simp add: mdiff-def normalize-def mset-of-def*)

**lemma** *mdiff-0* [*simp*]:  $0 -\# M = 0$   
**by** (*simp add: mdiff-def normalize-def*)

**lemma** *mdiff-0-right* [*simp*]:  $\text{multiset}(M) \implies M -\# 0 = M$   
**by** (*auto simp add: multiset-def mdiff-def normalize-def multiset-fun-iff mset-of-def funrestrict-def*)

**lemma** *mdiff-union-inverse2* [*simp*]:  $\text{multiset}(M) \implies M +\# \{\#a\# \} -\# \{\#a\# \} = M$   
**unfolding** *multiset-def munion-def mdiff-def msingle-def normalize-def mset-of-def*  
**apply** (*auto cong add: if-cong simp add: ltD multiset-fun-iff funrestrict-def subset-Un-iff2 [THEN iffD1]*)  
**prefer** 2 **apply** (*force intro!: lam-type*)  
**apply** (*subgoal-tac [2]  $\{x \in A \cup \{a\} . x \neq a \wedge x \in A\} = A$* )  
**apply** (*rule fun-extension, auto*)  
**apply** (*drule-tac  $x = A \cup \{a\}$  in spec*)  
**apply** (*simp add: Finite-Un*)  
**apply** (*force intro!: lam-type*)  
**done**

**lemma** *mcount-type* [*simp, TC*]:  $\text{multiset}(M) \implies \text{mcount}(M, a) \in \text{nat}$   
**by** (*auto simp add: multiset-def mcount-def mset-of-def multiset-fun-iff*)

**lemma** *mcount-0* [*simp*]:  $\text{mcount}(0, a) = 0$   
**by** (*simp add: mcount-def*)

**lemma** *mcount-single* [*simp*]:  $\text{mcount}(\{\#b\# \}, a) = (\text{if } a=b \text{ then } 1 \text{ else } 0)$   
**by** (*simp add: mcount-def mset-of-def msingle-def*)

**lemma** *mcount-union* [*simp*]:  $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket$   
 $\implies \text{mcount}(M +\# N, a) = \text{mcount}(M, a) \# + \text{mcount}(N, a)$   
**apply** (*auto simp add: multiset-def multiset-fun-iff mcount-def munion-def mset-of-def*)  
**done**

```

lemma mcount-diff [simp]:
  multiset(M)  $\implies$  mcount(M -# N, a) = mcount(M, a) #- mcount(N, a)
apply (simp add: multiset-def)
apply (auto dest!: not-lt-imp-le
  simp add: mdiff-def multiset-fun-iff mcount-def normalize-def mset-of-def)
apply (force intro!: lam-type)
apply (force intro!: lam-type)
done

```

```

lemma mcount-elem:  $\llbracket \text{multiset}(M); a \in \text{mset-of}(M) \rrbracket \implies 0 < \text{mcount}(M, a)$ 
apply (simp add: multiset-def, clarify)
apply (simp add: mcount-def mset-of-def)
apply (simp add: multiset-fun-iff)
done

```

```

lemma msize-0 [simp]: msize(0) = #0
by (simp add: msize-def)

```

```

lemma msize-single [simp]: msize({#a#}) = #1
by (simp add: msize-def)

```

```

lemma msize-type [simp, TC]: msize(M)  $\in$  int
by (simp add: msize-def)

```

```

lemma msize-zpositive: multiset(M)  $\implies$  #0  $\leq$  msize(M)
by (auto simp add: msize-def intro: g-zpos-imp-setsum-zpos)

```

```

lemma msize-int-of-nat: multiset(M)  $\implies \exists n \in \text{nat}. \text{msize}(M) = \#n$ 
apply (rule not-zneg-int-of)
apply (simp-all (no-asm-simp) add: msize-type [THEN znegative-iff-zless-0] not-zless-iff-zle
  msize-zpositive)
done

```

```

lemma not-empty-multiset-imp-exist:
   $\llbracket M \neq 0; \text{multiset}(M) \rrbracket \implies \exists a \in \text{mset-of}(M). 0 < \text{mcount}(M, a)$ 
apply (simp add: multiset-def)
apply (erule not-emptyE)
apply (auto simp add: mset-of-def mcount-def multiset-fun-iff)
apply (blast dest!: fun-is-rel)
done

```

```

lemma msize-eq-0-iff: multiset(M)  $\implies \text{msize}(M) = \#0 \iff M = 0$ 
apply (simp add: msize-def, auto)
apply (rule-tac P = setsum (u,v)  $\neq$  #0 for u v in swap)
apply blast
apply (drule not-empty-multiset-imp-exist, assumption, clarify)
apply (subgoal-tac Finite (mset-of (M) - {a}) )

```

```

prefer 2 apply (simp add: Finite-Diff)
apply (subgoal-tac setsum (λx. $# mcount (M, x), cons (a, mset-of (M) - {a}))=#0)
prefer 2 apply (simp add: cons-Diff, simp)
apply (subgoal-tac #0 $# ≤ setsum (λx. $# mcount (M, x), mset-of (M) - {a}) )
apply (rule-tac [2] g-zpos-imp-setsum-zpos)
apply (auto simp add: Finite-Diff not-zless-iff-zle [THEN iff-sym] znegative-iff-zless-0
[THEN iff-sym])
apply (rule not-zneg-int-of [THEN bexE])
apply (auto simp del: int-of-0 simp add: int-of-add [symmetric] int-of-0 [symmetric])
done

```

```

lemma setsum-mcount-Int:
  Finite(A) ⇒ setsum(λa. $# mcount(N, a), A ∩ mset-of(N))
    = setsum(λa. $# mcount(N, a), A)
apply (induct rule: Finite-induct)
apply auto
apply (subgoal-tac Finite (B ∩ mset-of (N)))
prefer 2 apply (blast intro: subset-Finite)
apply (auto simp add: mcount-def Int-cons-left)
done

```

```

lemma msize-union [simp]:
  [multiset(M); multiset(N)] ⇒ msize(M +# N) = msize(M) $+ msize(N)
apply (simp add: msize-def setsum-Un setsum-addf int-of-add setsum-mcount-Int)
apply (subst Int-commute)
apply (simp add: setsum-mcount-Int)
done

```

```

lemma msize-eq-succ-imp-elem: [msize(M) = $# succ(n); n ∈ nat] ⇒ ∃ a. a ∈
mset-of(M)
  unfolding msize-def
apply (blast dest: setsum-succD)
done

```

```

lemma equality-lemma:
  [multiset(M); multiset(N); ∀ a. mcount(M, a)=mcount(N, a)]
    ⇒ mset-of(M)=mset-of(N)
apply (simp add: multiset-def)
apply (rule sym, rule equalityI)
apply (auto simp add: multiset-fun-iff mcount-def mset-of-def)
apply (drule-tac [!] x=x in spec)
apply (case-tac [2] x ∈ Aa, case-tac x ∈ A, auto)
done

```

```

lemma multiset-equality:
  [multiset(M); multiset(N)] ⇒ M=N ↔ (∀ a. mcount(M, a)=mcount(N, a))
apply auto

```



```

apply (subgoal-tac mset-of (M) = mset-of (N) )
prefer 2 apply (blast intro: equality-lemma)
apply (simp add: multiset-def mset-of-def)
apply (auto simp add: multiset-fun-iff)
apply (rule fun-extension)
apply (blast, blast)
apply (drule-tac x = x in spec)
apply (auto simp add: mcount-def mset-of-def)
done

```

```

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

```

```

lemma empty-eq-munion-iff [simp]:  $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies (0 = M + \# N) \longleftrightarrow (M=0 \wedge N=0)$ 
apply (rule iffI, drule sym)
apply (simp-all add: multiset-equality)
done

```

```

lemma munion-right-cancel [simp]:
 $\llbracket \text{multiset}(M); \text{multiset}(N); \text{multiset}(K) \rrbracket \implies (M + \# K = N + \# K) \longleftrightarrow (M=N)$ 
by (auto simp add: multiset-equality)

```

```

lemma munion-left-cancel [simp]:
 $\llbracket \text{multiset}(K); \text{multiset}(M); \text{multiset}(N) \rrbracket \implies (K + \# M = K + \# N) \longleftrightarrow (M = N)$ 
by (auto simp add: multiset-equality)

```

```

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

```

```

lemma munion-is-single:
 $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket \implies (M + \# N = \{\#a\# \}) \longleftrightarrow (M=\{\#a\# \} \wedge N=0) \mid (M = 0 \wedge N = \{\#a\# \})$ 
apply (simp (no-asm-simp) add: multiset-equality)
apply safe
apply simp-all
apply (case-tac aa=a)
apply (drule-tac [2] x = aa in spec)
apply (drule-tac x = a in spec)
apply (simp add: nat-add-eq-1-cases, simp)
apply (case-tac aaa=aa, simp)
apply (drule-tac x = aa in spec)
apply (simp add: nat-add-eq-1-cases)

```

```

apply (case-tac aaa=a)
apply (drule-tac [4] x = aa in spec)
apply (drule-tac [3] x = a in spec)
apply (drule-tac [2] x = aaa in spec)
apply (drule-tac x = aa in spec)
apply (simp-all add: nat-add-eq-1-cases)
done

```

```

lemma msingle-is-union:  $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket$ 
 $\implies (\{ \#a\# \} = M +\# N) \longleftrightarrow (\{ \#a\# \} = M \wedge N=0 \mid M = 0 \wedge \{ \#a\# \} = N)$ 
apply (subgoal-tac ( $\{ \#a\# \} = M +\# N \longleftrightarrow (M +\# N = \{ \#a\# \})$ ) )
apply (simp (no-asm-simp) add: munion-is-single)
apply blast
apply (blast dest: sym)
done

```

```

lemma setsum-decr:
 $\text{Finite}(A)$ 
 $\implies (\forall M. \text{multiset}(M) \longrightarrow$ 
 $(\forall a \in \text{mset-of}(M). \text{setsum}(\lambda z. \$\# \text{mcount}(M(a:=M'a \#- 1), z), A) =$ 
 $(\text{if } a \in A \text{ then } \text{setsum}(\lambda z. \$\# \text{mcount}(M, z), A) \$- \#1$ 
 $\text{ else } \text{setsum}(\lambda z. \$\# \text{mcount}(M, z), A))))$ 
unfolding multiset-def
apply (erule Finite-induct)
apply (auto simp add: multiset-fun-iff)
unfolding mset-of-def mcount-def
apply (case-tac  $x \in A, \text{auto}$ )
apply (subgoal-tac  $\$ \# M ' x \$+ \#-1 = \$ \# M ' x \$- \$ \# 1$ )
apply (erule ssubst)
apply (rule int-of-diff, auto)
done

```

```

lemma setsum-decr2:
 $\text{Finite}(A)$ 
 $\implies \forall M. \text{multiset}(M) \longrightarrow (\forall a \in \text{mset-of}(M).$ 
 $\text{setsum}(\lambda x. \$\# \text{mcount}(\text{funrestrict}(M, \text{mset-of}(M)-\{a\}), x), A) =$ 
 $(\text{if } a \in A \text{ then } \text{setsum}(\lambda x. \$\# \text{mcount}(M, x), A) \$- \$ \# M'a$ 
 $\text{ else } \text{setsum}(\lambda x. \$\# \text{mcount}(M, x), A)))$ 
apply (simp add: multiset-def)
apply (erule Finite-induct)
apply (auto simp add: multiset-fun-iff mcount-def mset-of-def)
done

```

```

lemma setsum-decr3:  $\llbracket \text{Finite}(A); \text{multiset}(M); a \in \text{mset-of}(M) \rrbracket$ 
 $\implies \text{setsum}(\lambda x. \$\# \text{mcount}(\text{funrestrict}(M, \text{mset-of}(M)-\{a\}), x), A - \{a\})$ 
 $=$ 
 $(\text{if } a \in A \text{ then } \text{setsum}(\lambda x. \$\# \text{mcount}(M, x), A) \$- \$ \# M'a$ 

```

```

      else setsum( $\lambda x. \$\# \text{ mcount}(M, x), A$ )
apply (subgoal-tac setsum ( $\lambda x. \$\# \text{ mcount}(\text{funrestrict}(M, \text{mset-of}(M) - \{a\}), x), A - \{a\}$ )
= setsum ( $\lambda x. \$\# \text{ mcount}(\text{funrestrict}(M, \text{mset-of}(M) - \{a\}), x), A$ ) )
apply (rule-tac [2] setsum-Diff [symmetric])
apply (rule sym, rule ssubst, blast)
apply (rule sym, drule setsum-decr2, auto)
apply (simp add: mcount-def mset-of-def)
done

```

```

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

```

```

lemma succ-pred-eq-self:  $\llbracket 0 < n; n \in \text{nat} \rrbracket \implies \text{succ}(n \# - 1) = n$ 
apply (subgoal-tac  $1 \leq n$ )
apply (drule add-diff-inverse2, auto)
done

```

Specialized for use in the proof below.

```

lemma multiset-funrestrict:
   $\llbracket \forall a \in A. M \text{ ' } a \in \text{nat} \wedge 0 < M \text{ ' } a; \text{Finite}(A) \rrbracket$ 
 $\implies \text{multiset}(\text{funrestrict}(M, A - \{a\}))$ 
apply (simp add: multiset-def multiset-fun-iff)
apply (rule-tac  $x = A - \{a\}$  in exI)
apply (auto intro: Finite-Diff funrestrict-type)
done

```

```

lemma multiset-induct-aux:
  assumes prem1:  $\bigwedge M a. \llbracket \text{multiset}(M); a \notin \text{mset-of}(M); P(M) \rrbracket \implies P(\text{cons}(\langle a, 1 \rangle, M))$ 
  and prem2:  $\bigwedge M b. \llbracket \text{multiset}(M); b \in \text{mset-of}(M); P(M) \rrbracket \implies P(M(b := M \text{ ' } b \# + 1))$ 
  shows
     $\llbracket n \in \text{nat}; P(0) \rrbracket$ 
 $\implies (\forall M. \text{multiset}(M) \longrightarrow$ 
  (setsum( $\lambda x. \$\# \text{ mcount}(M, x), \{x \in \text{mset-of}(M). 0 < M \text{ ' } x\} = \$\# n$ )  $\longrightarrow P(M)$ )
apply (erule nat-induct, clarify)
apply (frule msize-eq-0-iff)
apply (auto simp add: mset-of-def multiset-def multiset-fun-iff msize-def)
apply (subgoal-tac setsum ( $\lambda x. \$\# \text{ mcount}(M, x), A = \$\# \text{ succ}(x)$ ) )
apply (drule setsum-succD, auto)
apply (case-tac  $1 < M \text{ ' } a$ )
apply (drule-tac [2] not-lt-imp-le)
apply (simp-all add: nat-le-1-cases)
apply (subgoal-tac  $M = (M (a := M \text{ ' } a \# - 1)) (a := (M (a := M \text{ ' } a \# - 1)) \text{ ' } a \# + 1)$ 
)
apply (rule-tac [2]  $A = A$  and  $B = \lambda x. \text{nat}$  and  $D = \lambda x. \text{nat}$  in fun-extension)
apply (rule-tac [3] update-type) +
apply (simp-all (no-asm-simp))
apply (rule-tac [2] impI)

```

```

  apply (rule-tac [2] succ-pred-eq-self [symmetric])
  apply (simp-all (no-asm-simp))
  apply (rule subst, rule sym, blast, rule prem2)
  apply (simp (no-asm) add: multiset-def multiset-fun-iff)
  apply (rule-tac  $x = A$  in exI)
  apply (force intro: update-type)
  apply (simp (no-asm-simp) add: mset-of-def mcount-def)
  apply (drule-tac  $x = M$  ( $a := M$  '  $a \# - 1$ ) in spec)
  apply (drule mp, drule-tac [2] mp, simp-all)
  apply (rule-tac  $x = A$  in exI)
  apply (auto intro: update-type)
  apply (subgoal-tac Finite ( $\{x \in \text{cons } (a, A) . x \neq a \longrightarrow 0 < M'x\}$ ))
  prefer 2 apply (blast intro: Collect-subset [THEN subset-Finite] Finite-cons)
  apply (drule-tac  $A = \{x \in \text{cons } (a, A) . x \neq a \longrightarrow 0 < M'x\}$  in setsum-decr)
  apply (drule-tac  $x = M$  in spec)
  apply (subgoal-tac multiset ( $M$ ))
  prefer 2
  apply (simp add: multiset-def multiset-fun-iff)
  apply (rule-tac  $x = A$  in exI, force)
  apply (simp-all add: mset-of-def)
  apply (drule-tac  $\psi = \forall x \in A. u(x)$  for  $u$  in asm-rl)
  apply (drule-tac  $x = a$  in bspec)
  apply (simp (no-asm-simp))
  apply (subgoal-tac cons ( $a, A$ ) =  $A$ )
  prefer 2 apply blast
  apply simp
  apply (subgoal-tac  $M = \text{cons } (<a, M'a>, \text{funrestrict } (M, A - \{a\}))$ )
  prefer 2
  apply (rule fun-cons-funrestrict-eq)
  apply (subgoal-tac cons ( $a, A - \{a\}$ ) =  $A$ )
  apply force
  apply force
  apply (rule-tac  $a = \text{cons } (<a, 1>, \text{funrestrict } (M, A - \{a\}))$  in ssubst)
  apply simp
  apply (frule multiset-funrestrict, assumption)
  apply (rule prem1, assumption)
  apply (simp add: mset-of-def)
  apply (drule-tac  $x = \text{funrestrict } (M, A - \{a\})$  in spec)
  apply (drule mp)
  apply (rule-tac  $x = A - \{a\}$  in exI)
  apply (auto intro: Finite-Diff funrestrict-type simp add: funrestrict)
  apply (frule-tac  $A = A$  and  $M = M$  and  $a = a$  in setsum-decr3)
  apply (simp (no-asm-simp) add: multiset-def multiset-fun-iff)
  apply blast
  apply (simp (no-asm-simp) add: mset-of-def)
  apply (drule-tac  $b = \text{if } u \text{ then } v \text{ else } w$  for  $u \ v \ w$  in sym, simp-all)
  apply (subgoal-tac  $\{x \in A - \{a\} . 0 < \text{funrestrict } (M, A - \{x\}) 'x\} = A - \{a\}$ )
  apply (auto intro!: setsum-cong simp add: zdiff-eq-iff zadd-commute multiset-def
    multiset-fun-iff mset-of-def)

```

done

**lemma** *multiset-induct2*:

```

  [[multiset(M); P(0);
    (∧ M a. [[multiset(M); a ∉ mset-of(M); P(M)] ⇒ P(cons(⟨a, 1⟩, M))];
    (∧ M b. [[multiset(M); b ∈ mset-of(M); P(M)] ⇒ P(M(b := M'b #+ 1))]]
    ⇒ P(M)
  apply (subgoal-tac ∃ n ∈ nat. setsum (λx. $# mcount (M, x), {x ∈ mset-of (M)
    . 0 < M ' x}) = $# n)
  apply (rule-tac [2] not-zneg-int-of)
  apply (simp-all (no-asm-simp) add: znegative-iff-zless-0 not-zless-iff-zle)
  apply (rule-tac [2] g-zpos-imp-setsum-zpos)
  prefer 2 apply (blast intro: multiset-set-of-Finite Collect-subset [THEN sub-
    set-Finite])
  prefer 2 apply (simp add: multiset-def multiset-fun-iff, clarify)
  apply (rule multiset-induct-aux [rule-format], auto)
  done

```

**lemma** *munion-single-case1*:

```

  [[multiset(M); a ∉ mset-of(M)] ⇒ M +# {#a#} = cons(⟨a, 1⟩, M)
  apply (simp add: multiset-def msingle-def)
  apply (auto simp add: munion-def)
  apply (unfold mset-of-def, simp)
  apply (rule fun-extension, rule lam-type, simp-all)
  apply (auto simp add: multiset-fun-iff fun-extend-apply)
  apply (drule-tac c = a and b = 1 in fun-extend3)
  apply (auto simp add: cons-eq Un-commute [of - {a}])
  done

```

**lemma** *munion-single-case2*:

```

  [[multiset(M); a ∈ mset-of(M)] ⇒ M +# {#a#} = M(a := M'a #+ 1)
  apply (simp add: multiset-def)
  apply (auto simp add: munion-def multiset-fun-iff msingle-def)
  apply (unfold mset-of-def, simp)
  apply (subgoal-tac A ∪ {a} = A)
  apply (rule fun-extension)
  apply (auto dest: domain-type intro: lam-type update-type)
  done

```

**lemma** *multiset-induct*:

```

  assumes M: multiset(M)
    and P0: P(0)
    and step: (∧ M a. [[multiset(M); P(M)] ⇒ P(M +# {#a#})]
  shows P(M)
  apply (rule multiset-induct2 [OF M])
  apply (simp-all add: P0)
  apply (frule-tac [2] a = b in munion-single-case2 [symmetric])

```

**apply** (*frule-tac*  $a = a$  **in** *munion-single-case1* [*symmetric*])  
**apply** (*auto intro: step*)  
**done**

**lemma** *MCollect-multiset* [*simp*]:  
 $\text{multiset}(M) \implies \text{multiset}(\{\# x \in M. P(x)\# \})$   
**apply** (*simp add: MCollect-def multiset-def mset-of-def, clarify*)  
**apply** (*rule-tac*  $x = \{x \in A. P(x)\}$  **in** *exI*)  
**apply** (*auto dest: CollectD1* [*THEN* [2] *apply-type*]  
 $\text{intro: Collect-subset}$  [*THEN subset-Finite*] *funrestrict-type*)  
**done**

**lemma** *mset-of-MCollect* [*simp*]:  
 $\text{multiset}(M) \implies \text{mset-of}(\{\# x \in M. P(x)\# \}) \subseteq \text{mset-of}(M)$   
**by** (*auto simp add: mset-of-def MCollect-def multiset-def funrestrict-def*)

**lemma** *MCollect-mem-iff* [*iff*]:  
 $x \in \text{mset-of}(\{\# x \in M. P(x)\# \}) \longleftrightarrow x \in \text{mset-of}(M) \wedge P(x)$   
**by** (*simp add: MCollect-def mset-of-def*)

**lemma** *mcount-MCollect* [*simp*]:  
 $\text{mcount}(\{\# x \in M. P(x)\# \}, a) = (\text{if } P(a) \text{ then } \text{mcount}(M, a) \text{ else } 0)$   
**by** (*simp add: mcount-def MCollect-def mset-of-def*)

**lemma** *multiset-partition*:  $\text{multiset}(M) \implies M = \{\# x \in M. P(x)\# \} + \# \{\# x \in M. \neg P(x)\# \}$   
**by** (*simp add: multiset-equality*)

**lemma** *natify-elem-is-self* [*simp*]:  
 $\llbracket \text{multiset}(M); a \in \text{mset-of}(M) \rrbracket \implies \text{natify}(M'a) = M'a$   
**by** (*auto simp add: multiset-def mset-of-def multiset-fun-iff*)

**lemma** *munion-eq-conv-diff*:  $\llbracket \text{multiset}(M); \text{multiset}(N) \rrbracket$   
 $\implies (M + \# \{\# a\# \} = N + \# \{\# b\# \}) \longleftrightarrow (M = N \wedge a = b \mid$   
 $M = N - \# \{\# a\# \} + \# \{\# b\# \} \wedge N = M - \# \{\# b\# \} + \# \{\# a\# \})$   
**apply** (*simp del: mcount-single add: multiset-equality*)  
**apply** (*rule iffI, erule-tac* [2] *disjE, erule-tac* [3] *conjE*)  
**apply** (*case-tac*  $a=b$ , *auto*)  
**apply** (*drule-tac*  $x = a$  **in** *spec*)  
**apply** (*drule-tac* [2]  $x = b$  **in** *spec*)  
**apply** (*drule-tac* [3]  $x = aa$  **in** *spec*)  
**apply** (*drule-tac* [4]  $x = a$  **in** *spec, auto*)  
**apply** (*subgoal-tac* [!] *mcount* ( $N, a$ ) :*nat*)  
**apply** (*erule-tac* [3] *natE, erule natE, auto*)  
**done**

**lemma** *melem-diff-single*:  
 $\text{multiset}(M) \implies$   
 $k \in \text{mset-of}(M - \# \{ \#a\# \}) \longleftrightarrow (k=a \wedge 1 < \text{mcount}(M,a)) \mid (k \neq a \wedge k \in \text{mset-of}(M))$   
**apply** (*simp add: multiset-def*)  
**apply** (*simp add: normalize-def mset-of-def msingle-def mdiff-def mcount-def*)  
**apply** (*auto dest: domain-type intro: zero-less-diff [THEN iffD1]*)  
*simp add: multiset-fun-iff apply-iff*)  
**apply** (*force intro!: lam-type*)  
**apply** (*force intro!: lam-type*)  
**apply** (*force intro!: lam-type*)  
**done**

**lemma** *munion-eq-conv-exist*:  
 $\llbracket M \in \text{Mult}(A); N \in \text{Mult}(A) \rrbracket$   
 $\implies (M + \# \{ \#a\# \} = N + \# \{ \#b\# \}) \longleftrightarrow$   
 $(M=N \wedge a=b \mid (\exists K \in \text{Mult}(A). M = K + \# \{ \#b\# \} \wedge N = K + \# \{ \#a\# \}))$   
**by** (*auto simp add: Mult-iff-multiset melem-diff-single munion-eq-conv-diff*)

## 8.2 Multiset Orderings

**lemma** *multirel1-type*:  $\text{multirel1}(A, r) \subseteq \text{Mult}(A) * \text{Mult}(A)$   
**by** (*auto simp add: multirel1-def*)

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

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

Monotonicity of *multirel1*

**lemma** *multirel1-mono1*:  $A \subseteq B \implies \text{multirel1}(A, r) \subseteq \text{multirel1}(B, r)$   
**apply** (*auto simp add: multirel1-def*)  
**apply** (*auto simp add: Un-subset-iff Mult-iff-multiset*)  
**apply** (*rule-tac x = a in bexI*)  
**apply** (*rule-tac x = M0 in bexI, simp*)  
**apply** (*rule-tac x = K in bexI*)  
**apply** (*auto simp add: Mult-iff-multiset*)  
**done**

**lemma** *multirel1-mono2*:  $r \subseteq s \implies \text{multirel1}(A, r) \subseteq \text{multirel1}(A, s)$   
**apply** (*simp add: multirel1-def, auto*)  
**apply** (*rule-tac x = a in bexI*)  
**apply** (*rule-tac x = M0 in bexI*)

```

apply (simp-all add: Mult-iff-multiset)
apply (rule-tac  $x = K$  in exI)
apply (simp-all add: Mult-iff-multiset, auto)
done

```

```

lemma multirel1-mono:
   $\llbracket A \subseteq B; r \subseteq s \rrbracket \implies \text{multirel1}(A, r) \subseteq \text{multirel1}(B, s)$ 
apply (rule subset-trans)
apply (rule multirel1-mono1)
apply (rule-tac [2] multirel1-mono2, auto)
done

```

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

```

lemma not-less-0 [iff]:  $\langle M, 0 \rangle \notin \text{multirel1}(A, r)$ 
by (auto simp add: multirel1-def Mult-iff-multiset)

```

```

lemma less-munion:  $\llbracket \langle N, M0 +\# \{\#a\# \} \rangle \in \text{multirel1}(A, r); M0 \in \text{Mult}(A) \rrbracket$ 
 $\implies$ 
  ( $\exists M. \langle M, M0 \rangle \in \text{multirel1}(A, r) \wedge N = M +\# \{\#a\# \}$ ) |
  ( $\exists K. K \in \text{Mult}(A) \wedge (\forall b \in \text{mset-of}(K). \langle b, a \rangle \in r) \wedge N = M0 +\# K$ )
apply (frule multirel1-type [THEN subsetD])
apply (simp add: multirel1-iff)
apply (auto simp add: munion-eq-conv-exist)
apply (rule-tac  $x = Ka +\# K$  in exI, auto, simp add: Mult-iff-multiset)
apply (simp (no-asm-simp) add: munion-left-cancel munion-assoc)
apply (auto simp add: munion-commute)
done

```

```

lemma multirel1-base:  $\llbracket M \in \text{Mult}(A); a \in A \rrbracket \implies \langle M, M +\# \{\#a\# \} \rangle \in \text{multirel1}(A, r)$ 
apply (auto simp add: multirel1-iff)
apply (simp add: Mult-iff-multiset)
apply (rule-tac  $x = a$  in exI, clarify)
apply (rule-tac  $x = M$  in exI, simp)
apply (rule-tac  $x = 0$  in exI, auto)
done

```

```

lemma acc-0:  $\text{acc}(0) = 0$ 
by (auto intro!: equalityI dest: acc.dom-subset [THEN subsetD])

```

```

lemma lemma1:  $\llbracket \forall b \in A. \langle b, a \rangle \in r \longrightarrow$ 
  ( $\forall M \in \text{acc}(\text{multirel1}(A, r)). M +\# \{\#b\# \} : \text{acc}(\text{multirel1}(A, r));$ 
   $M0 \in \text{acc}(\text{multirel1}(A, r)); a \in A;$ 
   $\forall M. \langle M, M0 \rangle \in \text{multirel1}(A, r) \longrightarrow M +\# \{\#a\# \} \in \text{acc}(\text{multirel1}(A, r)) \rrbracket$ 
 $\implies M0 +\# \{\#a\# \} \in \text{acc}(\text{multirel1}(A, r))$ 
apply (subgoal-tac  $M0 \in \text{Mult}(A)$  )
prefer 2
apply (erule acc.cases)

```



```

apply (erule fieldE)
apply (auto dest: multirel1-type [THEN subsetD])
apply (rule accI)
apply (rename-tac N)
apply (drule less-munion, blast)
apply (auto simp add: Mult-iff-multiset)
apply (erule-tac  $P = \forall x \in \text{mset-of } (K) . \langle x, a \rangle \in r$  in rev-mp)
apply (erule-tac  $P = \text{mset-of } (K) \subseteq A$  in rev-mp)
apply (erule-tac  $M = K$  in multiset-induct)

apply (simp (no-asm-simp))

apply (simp add: Ball-def Un-subset-iff, clarify)
apply (drule-tac  $x = aa$  in spec, simp)
apply (subgoal-tac  $aa \in A$ )
prefer 2 apply blast
apply (drule-tac  $x = M0 +\# M$  and  $P =$ 
 $\lambda x. x \in \text{acc}(\text{multirel1}(A, r)) \longrightarrow Q(x)$  for  $Q$  in spec)
apply (simp add: munion-assoc [symmetric])

apply (auto intro!: multirel1-base [THEN fieldI2] simp add: Mult-iff-multiset)
done

lemma lemma2:  $\llbracket \forall b \in A. \langle b, a \rangle \in r$ 
 $\longrightarrow (\forall M \in \text{acc}(\text{multirel1}(A, r)). M +\# \{\#b\} : \text{acc}(\text{multirel1}(A, r)));$ 
 $M \in \text{acc}(\text{multirel1}(A, r)); a \in A \rrbracket \Longrightarrow M +\# \{\#a\} \in \text{acc}(\text{multirel1}(A,$ 
 $r))$ 
apply (erule acc-induct)
apply (blast intro: lemma1)
done

lemma lemma3:  $\llbracket \text{wf}[A](r); a \in A \rrbracket$ 
 $\Longrightarrow \forall M \in \text{acc}(\text{multirel1}(A, r)). M +\# \{\#a\} \in \text{acc}(\text{multirel1}(A, r))$ 
apply (erule-tac  $a = a$  in wf-on-induct, blast)
apply (blast intro: lemma2)
done

lemma lemma4:  $\text{multiset}(M) \Longrightarrow \text{mset-of}(M) \subseteq A \longrightarrow$ 
 $\text{wf}[A](r) \longrightarrow M \in \text{field}(\text{multirel1}(A, r)) \longrightarrow M \in \text{acc}(\text{multirel1}(A, r))$ 
apply (erule multiset-induct)

apply clarify
apply (rule accI, force)
apply (simp add: multirel1-def)

apply clarify
apply simp
apply (subgoal-tac  $\text{mset-of } (M) \subseteq A$ )

```

```

prefer 2 apply blast
apply clarify
apply (drule-tac a = a in lemma3, blast)
apply (subgoal-tac M ∈ field (multirel1 (A,r)))
apply blast
apply (rule multirel1-base [THEN fieldI1])
apply (auto simp add: Mult-iff-multiset)
done

```

```

lemma all-accessible:  $\llbracket wf[A](r); M \in Mult(A); A \neq 0 \rrbracket \implies M \in acc(multirel1(A, r))$ 
apply (erule not-emptyE)
apply (rule lemma4 [THEN mp, THEN mp, THEN mp])
apply (rule-tac [4] multirel1-base [THEN fieldI1])
apply (auto simp add: Mult-iff-multiset)
done

```

```

lemma wf-on-multirel1:  $wf[A](r) \implies wf[A-||>nat-\{0\}](multirel1(A, r))$ 
apply (case-tac A=0)
apply (simp (no-asm-simp))
apply (rule wf-imp-wf-on)
apply (rule wf-on-field-imp-wf)
apply (simp (no-asm-simp) add: wf-on-0)
apply (rule-tac A = acc (multirel1 (A,r)) in wf-on-subset-A)
apply (rule wf-on-acc)
apply (blast intro: all-accessible)
done

```

```

lemma wf-multirel1:  $wf(r) \implies wf(multirel1(field(r), r))$ 
apply (simp (no-asm-use) add: wf-iff-wf-on-field)
apply (drule wf-on-multirel1)
apply (rule-tac A = field (r) -||> nat - {0} in wf-on-subset-A)
apply (simp (no-asm-simp))
apply (rule field-rel-subset)
apply (rule multirel1-type)
done

```

```

lemma multirel-type:  $multirel(A, r) \subseteq Mult(A)*Mult(A)$ 
apply (simp add: multirel-def)
apply (rule trancl-type [THEN subset-trans])
apply (auto dest: multirel1-type [THEN subsetD])
done

```

```

lemma multirel-mono:
 $\llbracket A \subseteq B; r \subseteq s \rrbracket \implies multirel(A, r) \subseteq multirel(B, s)$ 
apply (simp add: multirel-def)

```

**apply** (*rule trancl-mono*)  
**apply** (*rule multirel1-mono, auto*)  
**done**

**lemma** *add-diff-eq*:  $k \in \text{nat} \implies 0 < k \longrightarrow n \# + k \# - 1 = n \# + (k \# - 1)$   
**by** (*erule nat-induct, auto*)

**lemma** *mdiff-union-single-conv*:  $\llbracket a \in \text{mset-of}(J); \text{multiset}(I); \text{multiset}(J) \rrbracket$   
 $\implies I \# + J \# - \{\#a\} = I \# + (J \# - \{\#a\})$   
**apply** (*simp (no-asm-simp) add: multiset-equality*)  
**apply** (*case-tac a  $\notin$  mset-of (I)*)  
**apply** (*auto simp add: mcount-def mset-of-def multiset-def multiset-fun-iff*)  
**apply** (*auto dest: domain-type simp add: add-diff-eq*)  
**done**

**lemma** *diff-add-commute*:  $\llbracket n \leq m; m \in \text{nat}; n \in \text{nat}; k \in \text{nat} \rrbracket \implies m \# - n \# +$   
 $k = m \# + k \# - n$   
**by** (*auto simp add: le-iff less-iff-succ-add*)

**lemma** *multirel-implies-one-step*:  
 $\langle M, N \rangle \in \text{multirel}(A, r) \implies$   
 $\text{trans}[A](r) \longrightarrow$   
 $(\exists I J K.$   
 $I \in \text{Mult}(A) \wedge J \in \text{Mult}(A) \wedge K \in \text{Mult}(A) \wedge$   
 $N = I \# + J \wedge M = I \# + K \wedge J \neq 0 \wedge$   
 $(\forall k \in \text{mset-of}(K). \exists j \in \text{mset-of}(J). \langle k, j \rangle \in r))$   
**apply** (*simp add: multirel-def Ball-def Bex-def*)  
**apply** (*erule converse-trancl-induct*)  
**apply** (*simp-all add: multirel1-iff Mult-iff-multiset*)

**apply** *clarify*  
**apply** (*rule-tac x = M0 in exI, force*)

**apply** *clarify*  
**apply** *hypsubst-thin*  
**apply** (*case-tac a  $\in$  mset-of (Ka)*)  
**apply** (*rule-tac x = I in exI, simp (no-asm-simp)*)  
**apply** (*rule-tac x = J in exI, simp (no-asm-simp)*)  
**apply** (*rule-tac x = (Ka  $\# - \{\#a\}$ )  $\# + K$  in exI, simp (no-asm-simp)*)  
**apply** (*simp-all add: Un-subset-iff*)  
**apply** (*simp (no-asm-simp) add: munion-assoc [symmetric]*)  
**apply** (*drule-tac t =  $\lambda M. M \# - \{\#a\}$  in subst-context*)  
**apply** (*simp add: mdiff-union-single-conv melem-diff-single, clarify*)  
**apply** (*erule disjE, simp*)

```

apply (erule disjE, simp)
apply (drule-tac  $x = a$  and  $P = \lambda x. x : \# Ka \longrightarrow Q(x)$  for  $Q$  in spec)
apply clarify
apply (rule-tac  $x = xa$  in exI)
apply (simp (no-asm-simp))
apply (blast dest: trans-onD)

apply (subgoal-tac  $a : \# I$ )
apply (rule-tac  $x = I - \# \{ \# a \# \}$  in exI, simp (no-asm-simp))
apply (rule-tac  $x = J + \# \{ \# a \# \}$  in exI)
apply (simp (no-asm-simp) add: Un-subset-iff)
apply (rule-tac  $x = Ka + \# K$  in exI)
apply (simp (no-asm-simp) add: Un-subset-iff)
apply (rule conjI)
apply (simp (no-asm-simp) add: multiset-equality mcount-elem [THEN succ-pred-eq-self])
apply (rule conjI)
apply (drule-tac  $t = \lambda M. M - \# \{ \# a \# \}$  in subst-context)
apply (simp add: mdiff-union-inverse2)
apply (simp-all (no-asm-simp) add: multiset-equality)
apply (rule diff-add-commute [symmetric])
apply (auto intro: mcount-elem)
apply (subgoal-tac  $a \in \text{mset-of } (I + \# Ka)$  )
apply (drule-tac [2] sym, auto)
done

```

**lemma** *melem-imp-eq-diff-union* [*simp*]:  $\llbracket a \in \text{mset-of}(M); \text{multiset}(M) \rrbracket \implies M - \# \{ \# a \# \} + \# \{ \# a \# \} = M$   
**by** (*simp* *add*: *multiset-equality mcount-elem [THEN succ-pred-eq-self]*)

**lemma** *msize-eq-succ-imp-eq-union*:  
 $\llbracket \text{msize}(M) = \# \text{succ}(n); M \in \text{Mult}(A); n \in \text{nat} \rrbracket$   
 $\implies \exists a N. M = N + \# \{ \# a \# \} \wedge N \in \text{Mult}(A) \wedge a \in A$   
**apply** (*drule* *msize-eq-succ-imp-elem*, *auto*)  
**apply** (*rule-tac*  $x = a$  **in** *exI*)  
**apply** (*rule-tac*  $x = M - \# \{ \# a \# \}$  **in** *exI*)  
**apply** (*frule* *Mult-into-multiset*)  
**apply** (*simp* (*no-asm-simp*))  
**apply** (*auto* *simp* *add*: *Mult-iff-multiset*)  
**done**

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

```

apply (erule nat-induct, clarify)
apply (drule-tac  $M = J$  in msize-eq-0-iff, auto)

apply (subgoal-tac msize ( $J = \# \text{ succ } (x)$ ))
  prefer 2 apply simp
apply (frule-tac  $A = A$  in msize-eq-succ-imp-eq-union)
apply (simp-all add: Mult-iff-multiset, clarify)
apply (rename-tac  $J'$ , simp)
apply (case-tac  $J' = 0$ )
apply (simp add: multirel-def)
apply (rule r-into-trancl, clarify)
apply (simp add: multirel1-iff Mult-iff-multiset, force)

apply (drule sym, rotate-tac -1, simp)
apply (erule-tac  $V = \# x = \text{msize } (J')$  in thin-rl)
apply (frule-tac  $M = K$  and  $P = \lambda x. \langle x, a \rangle \in r$  in multiset-partition)
apply (erule-tac  $P = \forall k \in \text{mset-of } (K) . P(k)$  for  $P$  in rev-mp)
apply (erule ssubst)
apply (simp add: Ball-def, auto)
apply (subgoal-tac  $< (I + \# \{ \# x \in K. \langle x, a \rangle \in r \# \}) + \# \{ \# x \in K. \langle x, a \rangle \notin r \# \}, (I + \# \{ \# x \in K. \langle x, a \rangle \in r \# \}) + \# J' > \in \text{multirel}(A, r)$  )
  prefer 2
  apply (drule-tac  $x = I + \# \{ \# x \in K. \langle x, a \rangle \in r \# \}$  in spec)
  apply (rotate-tac -1)
  apply (drule-tac  $x = J'$  in spec)
  apply (rotate-tac -1)
  apply (drule-tac  $x = \{ \# x \in K. \langle x, a \rangle \notin r \# \}$  in spec, simp) apply blast
apply (simp add: munion-assoc [symmetric] multirel-def)
apply (rule-tac  $b = I + \# \{ \# x \in K. \langle x, a \rangle \in r \# \} + \# J'$  in trancl-trans, blast)
apply (rule r-into-trancl)
apply (simp add: multirel1-iff Mult-iff-multiset)
apply (rule-tac  $x = a$  in exI)
apply (simp (no-asm-simp))
apply (rule-tac  $x = I + \# J'$  in exI)
apply (auto simp add: munion-ac Un-subset-iff)
done

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

apply (subgoal-tac multiset ( $J$ ))
  prefer 2 apply (simp add: Mult-iff-multiset)
apply (frule-tac  $M = J$  in msize-int-of-nat)
apply (auto intro: one-step-implies-multirel-lemma)
done

```

**lemma** *multirel-irrefl-lemma*:

$Finite(A) \implies part-ord(A, r) \longrightarrow (\forall x \in A. \exists y \in A. \langle x, y \rangle \in r) \longrightarrow A=0$   
**apply** (*erule* *Finite-induct*)  
**apply** (*auto* *dest*: *subset-consI* [*THEN* [*2*] *part-ord-subset*])  
**apply** (*auto* *simp* *add*: *part-ord-def* *irrefl-def*)  
**apply** (*drule-tac*  $x = xa$  **in** *bspec*)  
**apply** (*drule-tac* [*2*]  $a = xa$  **and**  $b = x$  **in** *trans-onD*, *auto*)  
**done**

**lemma** *irrefl-on-multirel*:

$part-ord(A, r) \implies irrefl(Mult(A), multirel(A, r))$   
**apply** (*simp* *add*: *irrefl-def*)  
**apply** (*subgoal-tac* *trans*[*A*](*r*) )  
**prefer** 2 **apply** (*simp* *add*: *part-ord-def*, *clarify*)  
**apply** (*drule* *multirel-implies-one-step*, *clarify*)  
**apply** (*simp* *add*: *Mult-iff-multiset*, *clarify*)  
**apply** (*subgoal-tac* *Finite* (*mset-of* (*K*)))  
**apply** (*frule-tac*  $r = r$  **in** *multirel-irrefl-lemma*)  
**apply** (*frule-tac*  $B = mset-of$  (*K*) **in** *part-ord-subset*)  
**apply** *simp-all*  
**apply** (*auto* *simp* *add*: *multiset-def* *mset-of-def*)  
**done**

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

**apply** (*simp* *add*: *multirel-def* *trans-on-def*)  
**apply** (*blast* *intro*: *trancl-trans*)  
**done**

**lemma** *multirel-trans*:

$\llbracket \langle M, N \rangle \in multirel(A, r); \langle N, K \rangle \in multirel(A, r) \rrbracket \implies \langle M, K \rangle \in multirel(A, r)$   
**apply** (*simp* *add*: *multirel-def*)  
**apply** (*blast* *intro*: *trancl-trans*)  
**done**

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

**apply** (*simp* *add*: *multirel-def*)  
**apply** (*rule* *trans-trancl*)  
**done**

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

**apply** (*simp* (*no-asm*) *add*: *part-ord-def*)  
**apply** (*blast* *intro*: *irrefl-on-multirel* *trans-on-multirel*)  
**done**

**lemma** *munion-multirel1-mono*:

```


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

apply (frule multirel1-type [THEN subsetD])
apply (auto simp add: multirel1-iff Mult-iff-multiset)
apply (rule-tac  $x = a$  in exI)
apply (simp (no-asm-simp))
apply (rule-tac  $x = K +\# M0$  in exI)
apply (simp (no-asm-simp) add: Un-subset-iff)
apply (rule-tac  $x = Ka$  in exI)
apply (simp (no-asm-simp) add: munion-assoc)
done

```

**lemma** munion-multirel-mono2:

```


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

apply (frule multirel-type [THEN subsetD])
apply (simp (no-asm-use) add: multirel-def)
apply clarify
apply (drule-tac  $\psi = \langle M, N \rangle \in \text{multirel1}(A, r) \hat{+}$  in asm-rl)
apply (erule rev-mp)
apply (erule rev-mp)
apply (erule rev-mp)
apply (erule trancl-induct, clarify)
apply (blast intro: munion-multirel1-mono r-into-trancl, clarify)
apply (subgoal-tac  $y \in \text{Mult}(A)$  )
prefer 2
apply (blast dest: multirel-type [unfolded multirel-def, THEN subsetD])
apply (subgoal-tac  $\langle K +\# y, K +\# z \rangle \in \text{multirel1}(A, r)$  )
prefer 2 apply (blast intro: munion-multirel1-mono)
apply (blast intro: r-into-trancl trancl-trans)
done

```

**lemma** munion-multirel-mono1:

```


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

apply (frule multirel-type [THEN subsetD])
apply (rule-tac  $P = \lambda x. \langle x, u \rangle \in \text{multirel}(A, r)$  for  $u$  in munion-commute [THEN subst])
apply (subst munion-commute [of N])
apply (rule munion-multirel-mono2)
apply (auto simp add: Mult-iff-multiset)
done

```

**lemma** munion-multirel-mono:

```


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

apply (subgoal-tac  $M \in \text{Mult}(A) \wedge N \in \text{Mult}(A) \wedge K \in \text{Mult}(A) \wedge L \in \text{Mult}(A)$  )
prefer 2 apply (blast dest: multirel-type [THEN subsetD])

```

```

apply (blast intro: munion-multirel-mono1 multirel-trans munion-multirel-mono2)
done

```

## 8.4 Ordinal Multisets

```

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

```

```

lemmas multirel-Memrel-mono = multirel-mono [OF field-Memrel-mono Mem-
rel-mono]

```

```

lemma omultiset-is-multiset [simp]: omultiset(M)  $\implies$  multiset(M)
apply (simp add: omultiset-def)
apply (auto simp add: Mult-iff-multiset)
done

```

```

lemma munion-omultiset [simp]:  $\llbracket$ omultiset(M); omultiset(N) $\rrbracket \implies$  omultiset(M
+ $\#$  N)
apply (simp add: omultiset-def, clarify)
apply (rule-tac x = i  $\cup$  ia in exI)
apply (simp add: Mult-iff-multiset Ord-Un Un-subset-iff)
apply (blast intro: field-Memrel-mono)
done

```

```

lemma mdiff-omultiset [simp]: omultiset(M)  $\implies$  omultiset(M - $\#$  N)
apply (simp add: omultiset-def, clarify)
apply (simp add: Mult-iff-multiset)
apply (rule-tac x = i in exI)
apply (simp (no-asm-simp))
done

```

```

lemma irrefl-Memrel: Ord(i)  $\implies$  irrefl(field(Memrel(i)), Memrel(i))
apply (rule irreflI, clarify)
apply (subgoal-tac Ord (x) )
prefer 2 apply (blast intro: Ord-in-Ord)
apply (drule-tac i = x in ltI [THEN lt-irrefl], auto)
done

```

```

lemma trans-iff-trans-on: trans(r)  $\longleftrightarrow$  trans[field(r)](r)
by (simp add: trans-on-def trans-def, auto)

```

```

lemma part-ord-Memrel: Ord(i)  $\implies$  part-ord(field(Memrel(i)), Memrel(i))
apply (simp add: part-ord-def)
apply (simp (no-asm) add: trans-iff-trans-on [THEN iff-sym])
apply (blast intro: trans-Memrel irrefl-Memrel)
done

```



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

**lemma** *mless-not-refl*:  $\neg(M <\# M)$   
**apply** (*simp add: mless-def, clarify*)  
**apply** (*frule multirel-type* [THEN *subsetD*])  
**apply** (*drule part-ord-mless*)  
**apply** (*simp add: part-ord-def irrefl-def*)  
**done**

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

**lemma** *mless-trans*:  $\llbracket K <\# M; M <\# N \rrbracket \implies K <\# N$   
**apply** (*simp add: mless-def, clarify*)  
**apply** (*rule-tac x = i  $\cup$  ia in exI*)  
**apply** (*blast dest: multirel-Memrel-mono* [OF *Un-upper1 Un-upper1*, THEN *subsetD*]  
*multirel-Memrel-mono* [OF *Un-upper2 Un-upper2*, THEN *subsetD*]  
*intro: multirel-trans Ord-Un*)  
**done**

**lemma** *mless-not-sym*:  $M <\# N \implies \neg N <\# M$   
**apply** *clarify*  
**apply** (*rule mless-not-refl* [THEN *notE*])  
**apply** (*erule mless-trans, assumption*)  
**done**

**lemma** *mless-asy*:  $\llbracket M <\# N; \neg P \rrbracket \implies N <\# M \implies P$   
**by** (*blast dest: mless-not-sym*)

**lemma** *mle-refl* [*simp*]:  $\text{omultiset}(M) \implies M <\# = M$   
**by** (*simp add: mle-def*)

**lemma** *mle-antisym*:  
 $\llbracket M <\# = N; N <\# = M \rrbracket \implies M = N$   
**apply** (*simp add: mle-def*)  
**apply** (*blast dest: mless-not-sym*)  
**done**

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

```

apply (simp add: mle-def)
apply (blast intro: mless-trans)
done

```

```

lemma mless-le-iff:  $M <\# N \longleftrightarrow (M <\#= N \wedge M \neq N)$ 
by (simp add: mle-def, auto)

```

```

lemma munion-less-mono2:  $\llbracket M <\# N; \text{omultiset}(K) \rrbracket \implies K +\# M <\# K +\# N$ 
apply (simp add: mless-def omultiset-def, clarify)
apply (rule-tac  $x = i \cup ia$  in  $exI$ )
apply (simp add: Mult-iff-multiset Ord-Un Un-subset-iff)
apply (rule munion-multirel-mono2)
apply (blast intro: multirel-Memrel-mono [THEN subsetD])
apply (simp add: Mult-iff-multiset)
apply (blast intro: field-Memrel-mono [THEN subsetD])
done

```

```

lemma munion-less-mono1:  $\llbracket M <\# N; \text{omultiset}(K) \rrbracket \implies M +\# K <\# N +\# K$ 
by (force dest: munion-less-mono2 simp add: munion-commute)

```

```

lemma mless-imp-omultiset:  $M <\# N \implies \text{omultiset}(M) \wedge \text{omultiset}(N)$ 
by (auto simp add: mless-def omultiset-def dest: multirel-type [THEN subsetD])

```

```

lemma munion-less-mono:  $\llbracket M <\# K; N <\# L \rrbracket \implies M +\# N <\# K +\# L$ 
apply (frule-tac  $M = M$  in mless-imp-omultiset)
apply (frule-tac  $M = N$  in mless-imp-omultiset)
apply (blast intro: munion-less-mono1 munion-less-mono2 mless-trans)
done

```

```

lemma mle-imp-omultiset:  $M <\#= N \implies \text{omultiset}(M) \wedge \text{omultiset}(N)$ 
by (auto simp add: mle-def mless-imp-omultiset)

```

```

lemma mle-mono:  $\llbracket M <\#= K; N <\#= L \rrbracket \implies M +\# N <\#= K +\# L$ 
apply (frule-tac  $M = M$  in mle-imp-omultiset)
apply (frule-tac  $M = N$  in mle-imp-omultiset)
apply (auto simp add: mle-def intro: munion-less-mono1 munion-less-mono2 munion-less-mono)
done

```

```

lemma omultiset-0 [iff]:  $\text{omultiset}(0)$ 
by (auto simp add: omultiset-def Mult-iff-multiset)

```

```

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

```

```

apply (simp add: mle-def mless-def)
apply (subgoal-tac  $\exists i. \text{Ord } (i) \wedge M \in \text{Mult}(\text{field}(\text{Memrel}(i)))$ )
  prefer 2 apply (simp add: omultiset-def)
apply (case-tac  $M=0, \text{simp-all}, \text{clarify}$ )
apply (subgoal-tac  $<0 +\# 0, 0 +\# M> \in \text{multirel}(\text{field } (\text{Memrel}(i)), \text{Memrel}(i))$ )
apply (rule-tac [2] one-step-implies-multirel)
apply (auto simp add: Mult-iff-multiset)
done

lemma munion-upper1:  $\llbracket \text{omultiset}(M); \text{omultiset}(N) \rrbracket \implies M <\# = M +\# N$ 
apply (subgoal-tac  $M +\# 0 <\# = M +\# N$ )
apply (rule-tac [2] mle-mono, auto)
done

end

```

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

**theory** *Rmap* **imports** *ZF* **begin**

**consts**

*rmap* ::  $i \Rightarrow i$

**inductive**

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

**intros**

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

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

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

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

**unfolding** *rmap.defs*

**apply** (*rule lfp-mono*)

**apply** (*rule rmap.bnd-mono*) +

**apply** (*assumption* | *rule Sigma-mono list-mono domain-mono range-mono basic-monos*) +

**done**

**inductive-cases**

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

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

**declare** *rmap.intros* [*intro*]

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

**apply** (*rule rmap.dom-subset* [*THEN subset-trans*])

```

apply (assumption |
  rule domain-rel-subset range-rel-subset Sigma-mono list-mono) +
done

lemma rmap-total:  $A \subseteq \text{domain}(r) \implies \text{list}(A) \subseteq \text{domain}(\text{rmap}(r))$ 
apply (rule subsetI)
apply (erule list.induct)
apply blast+
done

lemma rmap-functional:  $\text{function}(r) \implies \text{function}(\text{rmap}(r))$ 
unfolding function-def
apply (rule impI [THEN allI, THEN allI])
apply (erule rmap.induct)
apply blast+
done

If  $f$  is a function then  $\text{rmap}(f)$  behaves as expected.

lemma rmap-fun-type:  $f \in A \multimap B \implies \text{rmap}(f): \text{list}(A) \multimap \text{list}(B)$ 
by (simp add: Pi-iff rmap-rel-type rmap-functional rmap-total)

lemma rmap-Nil:  $\text{rmap}(f) \text{ `Nil} = \text{Nil}$ 
by (unfold apply-def) blast

lemma rmap-Cons:  $\llbracket f \in A \multimap B; x \in A; xs: \text{list}(A) \rrbracket$ 
 $\implies \text{rmap}(f) \text{ `Cons}(x, xs) = \text{Cons}(f \text{ `} x, \text{rmap}(f) \text{ `} xs)$ 
by (blast intro: apply-equality apply-Pair rmap-fun-type rmap.intros)

end

```

## 10 Meta-theory of propositional logic

**theory** *PropLog* **imports** *ZF* **begin**

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

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

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

### 10.1 The datatype of propositions

**consts**

*propn* :: *i*

**datatype** *propn* =

*FIs*

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

## 10.2 The proof system

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

### abbreviation

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

### inductive

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

#### intros

$H$ :  $\llbracket p \in H; \ p \in \text{propn} \rrbracket \Longrightarrow H \mid - p$   
 $K$ :  $\llbracket p \in \text{propn}; \ q \in \text{propn} \rrbracket \Longrightarrow H \mid - p \Rightarrow q \Rightarrow p$   
 $S$ :  $\llbracket p \in \text{propn}; \ q \in \text{propn}; \ r \in \text{propn} \rrbracket$   
 $\Longrightarrow H \mid - (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$   
 $DN$ :  $p \in \text{propn} \Longrightarrow H \mid - ((p \Rightarrow \text{Fls}) \Rightarrow \text{Fls}) \Rightarrow p$   
 $MP$ :  $\llbracket H \mid - p \Rightarrow q; \ H \mid - p; \ p \in \text{propn}; \ q \in \text{propn} \rrbracket \Longrightarrow H \mid - q$

**type-intros** *propn.intros*

**declare** *propn.intros* [*simp*]

## 10.3 The semantics

### 10.3.1 Semantics of propositional logic.

#### consts

*is-true-fun* ::  $[i, i] \Rightarrow i$

#### primrec

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

#### definition

*is-true* ::  $[i, i] \Rightarrow o$     **where**  
*is-true*( $p$ ,  $t$ )  $\equiv$  *is-true-fun*( $p$ ,  $t$ ) = 1  
— this definition is required since predicates can't be recursive

**lemma** *is-true-Fls* [*simp*]: *is-true*(*Fls*,  $t$ )  $\longleftrightarrow$  *False*  
**by** (*simp add: is-true-def*)

**lemma** *is-true-Var* [*simp*]: *is-true*( $\#v$ ,  $t$ )  $\longleftrightarrow v \in t$   
**by** (*simp add: is-true-def*)

**lemma** *is-true-Imp* [*simp*]: *is-true*( $p \Rightarrow q$ ,  $t$ )  $\longleftrightarrow$  (*is-true*( $p$ ,  $t$ )  $\longrightarrow$  *is-true*( $q$ ,  $t$ ))  
**by** (*simp add: is-true-def*)

### 10.3.2 Logical consequence

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

**definition**

$logcon :: [i,i] \Rightarrow o \quad (\text{infixl } \langle | \Rightarrow \rangle 50) \quad \text{where}$   
 $H \models p \equiv \forall t. (\forall q \in H. is\_true(q,t)) \longrightarrow is\_true(p,t)$

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

**consts**

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

**primrec**

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

## 10.4 Proof theory of propositional logic

**lemma** *thms-mono*:  $G \subseteq H \Longrightarrow thms(G) \subseteq thms(H)$

**unfolding** *thms.defs*

**apply** (*rule* *lfp-mono*)

**apply** (*rule* *thms.bnd-mono*) +

**apply** (*assumption* | *rule* *univ-mono* *basic-monos*) +

**done**

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

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

**lemma** *thms-MP*:  $\llbracket H \vdash p \Rightarrow q; H \vdash p \rrbracket \Longrightarrow H \vdash q$

— Stronger Modus Ponens rule: no typechecking!

**apply** (*rule* *thms.MP*)

**apply** (*erule* *asm-rl* *thms-in-pl* *thms-in-pl* [*THEN* *ImpE*]) +

**done**

**lemma** *thms-I*:  $p \in propn \Longrightarrow H \vdash p \Rightarrow p$

— Rule is called *I* for Identity Combinator, not for Introduction.

**apply** (*rule* *thms.S* [*THEN* *thms-MP*, *THEN* *thms-MP*])

**apply** (*rule-tac* [5] *thms.K*)

**apply** (*rule-tac* [4] *thms.K*)

**apply** *simp-all*

**done**

### 10.4.1 Weakening, left and right

**lemma** *weaken-left*:  $\llbracket G \subseteq H; G \vdash p \rrbracket \Longrightarrow H \vdash p$

— Order of premises is convenient with *THEN*

**by** (*erule* *thms-mono* [*THEN* *subsetD*])

**lemma** *weaken-left-cons*:  $H \vdash p \Longrightarrow cons(a,H) \vdash p$

**by** (*erule subset-consI* [*THEN weaken-left*])  
**lemmas** *weaken-left-Un1* = *Un-upper1* [*THEN weaken-left*]  
**lemmas** *weaken-left-Un2* = *Un-upper2* [*THEN weaken-left*]  
**lemma** *weaken-right*:  $\llbracket H \mid - q; p \in \text{propn} \rrbracket \implies H \mid - p \Rightarrow q$   
**by** (*simp-all add: thms.K* [*THEN thms-MP*] *thms-in-pl*)

#### 10.4.2 The deduction theorem

**theorem** *deduction*:  $\llbracket \text{cons}(p, H) \mid - q; p \in \text{propn} \rrbracket \implies H \mid - p \Rightarrow q$   
**apply** (*erule thms.induct*)  
**apply** (*blast intro: thms-I thms.H* [*THEN weaken-right*])  
**apply** (*blast intro: thms.K* [*THEN weaken-right*])  
**apply** (*blast intro: thms.S* [*THEN weaken-right*])  
**apply** (*blast intro: thms.DN* [*THEN weaken-right*])  
**apply** (*blast intro: thms.S* [*THEN thms-MP*] [*THEN thms-MP*])  
**done**

#### 10.4.3 The cut rule

**lemma** *cut*:  $\llbracket H \mid - p; \text{cons}(p, H) \mid - q \rrbracket \implies H \mid - q$   
**apply** (*rule deduction* [*THEN thms-MP*])  
**apply** (*simp-all add: thms-in-pl*)  
**done**  
**lemma** *thms-FlsE*:  $\llbracket H \mid - \text{Fls}; p \in \text{propn} \rrbracket \implies H \mid - p$   
**apply** (*rule thms.DN* [*THEN thms-MP*])  
**apply** (*rule-tac* [2] *weaken-right*)  
**apply** (*simp-all add: propn.intros*)  
**done**  
**lemma** *thms-notE*:  $\llbracket H \mid - p \Rightarrow \text{Fls}; H \mid - p; q \in \text{propn} \rrbracket \implies H \mid - q$   
**by** (*erule thms-MP* [*THEN thms-FlsE*])

#### 10.4.4 Soundness of the rules wrt truth-table semantics

**theorem** *soundness*:  $H \mid - p \implies H \models p$   
**unfolding** *logcon-def*  
**apply** (*induct set: thms*)  
**apply** *auto*  
**done**

### 10.5 Completeness

#### 10.5.1 Towards the completeness proof

**lemma** *Fls-Imp*:  $\llbracket H \mid - p \Rightarrow \text{Fls}; q \in \text{propn} \rrbracket \implies H \mid - p \Rightarrow q$   
**apply** (*frule thms-in-pl*)  
**apply** (*rule deduction*)

```

apply (rule weaken-left-cons [THEN thms-notE])
apply (blast intro: thms.H elim: ImpE)+
done

lemma Imp-Fls:  $\llbracket H \mid - p; H \mid - q \Rightarrow Fls \rrbracket \Longrightarrow H \mid - (p \Rightarrow q) \Rightarrow Fls$ 
apply (frule thms-in-pl)
apply (frule thms-in-pl [of concl:  $q \Rightarrow Fls$ ])
apply (rule deduction)
apply (erule weaken-left-cons [THEN thms-MP])
apply (rule consI1 [THEN thms.H, THEN thms-MP])
apply (blast intro: weaken-left-cons elim: ImpE)+
done

lemma hyps-thms-if:
   $p \in \text{propn} \Longrightarrow \text{hyps}(p,t) \mid - (\text{if is-true}(p,t) \text{ then } p \text{ else } p \Rightarrow Fls)$ 
  — Typical example of strengthening the induction statement.
apply simp
apply (induct-tac p)
apply (simp-all add: thms-I thms.H)
apply (safe elim!: Fls-Imp [THEN weaken-left-Un1] Fls-Imp [THEN weaken-left-Un2])
apply (blast intro: weaken-left-Un1 weaken-left-Un2 weaken-right Imp-Fls)+
done

lemma logcon-thms-p:  $\llbracket p \in \text{propn}; 0 \mid = p \rrbracket \Longrightarrow \text{hyps}(p,t) \mid - p$ 
  — Key lemma for completeness; yields a set of assumptions satisfying  $p$ 
apply (drule hyps-thms-if)
apply (simp add: logcon-def)
done

For proving certain theorems in our new propositional logic.

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

The excluded middle in the form of an elimination rule.

lemma thms-excluded-middle:
   $\llbracket p \in \text{propn}; q \in \text{propn} \rrbracket \Longrightarrow H \mid - (p \Rightarrow q) \Rightarrow ((p \Rightarrow Fls) \Rightarrow q) \Rightarrow q$ 
apply (rule deduction [THEN deduction])
apply (rule thms.DN [THEN thms-MP])
apply (best intro!: propn-SIs intro: propn-Is)+
done

lemma thms-excluded-middle-rule:
   $\llbracket \text{cons}(p,H) \mid - q; \text{cons}(p \Rightarrow Fls,H) \mid - q; p \in \text{propn} \rrbracket \Longrightarrow H \mid - q$ 
  — Hard to prove directly because it requires cuts
apply (rule thms-excluded-middle [THEN thms-MP, THEN thms-MP])
apply (blast intro!: propn-SIs intro: propn-Is)+
done

```



### 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*:

$p \in \text{propn} \implies \text{hyps}(p, t - \{v\}) \subseteq \text{cons}(\#v \Rightarrow \text{Fls}, \text{hyps}(p, t) - \{\#v\})$   
**by** (*induct set: propn*) *auto*

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*:

$p \in \text{propn} \implies \text{hyps}(p, \text{cons}(v, t)) \subseteq \text{cons}(\#v, \text{hyps}(p, t) - \{\#v \Rightarrow \text{Fls}\})$   
**by** (*induct set: propn*) *auto*

Two lemmas for use with *weaken-left*

**lemma** *cons-Diff-same*:  $B - C \subseteq \text{cons}(a, B - \text{cons}(a, C))$   
**by** *blast*

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

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*:  $p \in \text{propn} \implies \text{hyps}(p, t) \in \text{Fin}(\bigcup v \in \text{nat}. \{\#v, \#v \Rightarrow \text{Fls}\})$   
**by** (*induct set: propn*) *auto*

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

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

**lemma** *completeness-0-lemma* [*rule-format*]:

$\llbracket p \in \text{propn}; 0 \models p \rrbracket \implies \forall t. \text{hyps}(p, t) - \text{hyps}(p, t0) \vdash p$   
**apply** (*frule hyps-finite*)  
**apply** (*erule Fin-induct*)  
**apply** (*simp add: logcon-thms-p Diff-0*)

inductive step

**apply** *safe*

Case  $\text{hyps}(p, t) - \text{cons}(\#v, Y) \vdash p$

**apply** (*rule thms-excluded-middle-rule*)  
**apply** (*erule-tac* [3] *propn.intros*)  
**apply** (*blast intro: cons-Diff-same* [*THEN* *weaken-left*])  
**apply** (*blast intro: cons-Diff-subset2* [*THEN* *weaken-left*]  
*hyps-Diff* [*THEN* *Diff-weaken-left*])

Case  $\text{hyps}(p, t) - \text{cons}(\#v \Rightarrow \text{Fls}, Y) \mid - p$

```

apply (rule thms-excluded-middle-rule)
  apply (erule-tac [3] propn.intros)
  apply (blast intro: cons-Diff-subset2 [THEN weaken-left]
    hyps-cons [THEN Diff-weaken-left])
  apply (blast intro: cons-Diff-same [THEN weaken-left])
done

```

### 10.5.3 Completeness theorem

**lemma** *completeness-0*:  $\llbracket p \in \text{propn}; \ 0 \models p \rrbracket \Longrightarrow 0 \mid - p$   
 — The base case for completeness

```

apply (rule Diff-cancel [THEN subst])
apply (blast intro: completeness-0-lemma)
done

```

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

```

by (simp add: logcon-def)

```

**lemma** *completeness*:

```

   $H \in \text{Fin}(\text{propn}) \Longrightarrow p \in \text{propn} \Longrightarrow H \models p \Longrightarrow H \mid - p$ 
apply (induct arbitrary: p set: Fin)
apply (safe intro!: completeness-0)
apply (rule weaken-left-cons [THEN thms-MP])
apply (blast intro!: logcon-Imp propn.intros)
apply (blast intro: propn-Is)
done

```

**theorem** *thms-iff*:  $H \in \text{Fin}(\text{propn}) \Longrightarrow H \mid - p \longleftrightarrow H \models p \wedge p \in \text{propn}$   
**by** (*blast intro: soundness completeness thms-in-pl*)

**end**

## 11 Lists of $n$ elements

**theory** *ListN* **imports** *ZF* **begin**

Inductive definition of lists of  $n$  elements; see [3].

```

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

```

```

lemma list-into-listn:  $l \in \text{list}(A) \implies \langle \text{length}(l), l \rangle \in \text{listn}(A)$ 
  by (induct set: list) (simp-all add: listn.intros)

lemma listn-iff:  $\langle n, l \rangle \in \text{listn}(A) \iff l \in \text{list}(A) \wedge \text{length}(l) = n$ 
  apply (rule iffI)
  apply (erule listn.induct)
  apply auto
  apply (blast intro: list-into-listn)
  done

lemma listn-image-eq:  $\text{listn}(A) \restriction \{n\} = \{l \in \text{list}(A). \text{length}(l) = n\}$ 
  apply (rule equality-iffI)
  apply (simp add: listn-iff separation image-singleton-iff)
  done

lemma listn-mono:  $A \subseteq B \implies \text{listn}(A) \subseteq \text{listn}(B)$ 
  unfolding listn.defs
  apply (rule lfp-mono)
  apply (rule listn.bnd-mono)+
  apply (assumption | rule univ-mono Sigma-mono list-mono basic-monos)+
  done

lemma listn-append:
   $\llbracket \langle n, l \rangle \in \text{listn}(A); \langle n', l' \rangle \in \text{listn}(A) \rrbracket \implies \langle n\# + n', l @ l' \rangle \in \text{listn}(A)$ 
  apply (erule listn.induct)
  apply (frule listn.dom-subset [THEN subsetD])
  apply (simp-all add: listn.intros)
  done

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 ZF
begin

```

Curiously, combinators do not include free variables.  
 Example taken from [1].

## 12.1 Definitions

Datatype definition of combinators  $S$  and  $K$ .

```
consts comb :: i
datatype comb =
  K
| S
| app (p ∈ comb, q ∈ comb) (infixl  $\langle \cdot \rangle$  90)
```

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

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

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

**inductive**

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

**intros**

```
K:  $\llbracket p \in \text{comb}; q \in \text{comb} \rrbracket \Longrightarrow K \cdot p \cdot q \rightarrow^1 p$ 
S:  $\llbracket p \in \text{comb}; q \in \text{comb}; r \in \text{comb} \rrbracket \Longrightarrow S \cdot p \cdot q \cdot r \rightarrow^1 (p \cdot r) \cdot (q \cdot r)$ 
Ap1:  $\llbracket p \rightarrow^1 q; r \in \text{comb} \rrbracket \Longrightarrow p \cdot r \rightarrow^1 q \cdot r$ 
Ap2:  $\llbracket p \rightarrow^1 q; r \in \text{comb} \rrbracket \Longrightarrow r \cdot p \rightarrow^1 r \cdot q$ 
```

**type-intros** *comb.intros*

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

```
consts parcontract :: i
```

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

```
abbreviation parcontract-multi :: [i,i]  $\Rightarrow$  o (infixl  $\langle \Rightarrow \rangle$  50)
  where  $p \Rightarrow q \equiv \langle p, q \rangle \in \text{parcontract}^+$ 
```

**inductive**

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

**intros**

```
refl:  $\llbracket p \in \text{comb} \rrbracket \Longrightarrow p \Rightarrow^1 p$ 
K:  $\llbracket p \in \text{comb}; q \in \text{comb} \rrbracket \Longrightarrow K \cdot p \cdot q \Rightarrow^1 p$ 
S:  $\llbracket p \in \text{comb}; q \in \text{comb}; r \in \text{comb} \rrbracket \Longrightarrow S \cdot p \cdot q \cdot r \Rightarrow^1 (p \cdot r) \cdot (q \cdot r)$ 
Ap:  $\llbracket p \Rightarrow^1 q; r \Rightarrow^1 s \rrbracket \Longrightarrow p \cdot r \Rightarrow^1 q \cdot s$ 
```

**type-intros** *comb.intros*

Misc definitions.

```
definition I :: i
  where  $I \equiv S \cdot K \cdot K$ 
```

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

## 12.2 Transitive closure preserves the Church-Rosser property

**lemma** *diamond-strip-lemmaD* [*rule-format*]:  
 $\llbracket \text{diamond}(r); \langle x, y \rangle : r^+ \rrbracket \Longrightarrow$   
 $\forall y'. \langle x, y' \rangle : r \longrightarrow (\exists z. \langle y', z \rangle : r^+ \wedge \langle y, z \rangle : r)$   
**unfolding** *diamond-def*  
**apply** (*erule trancl-induct*)  
**apply** (*blast intro: r-into-trancl*)  
**apply** *clarify*  
**apply** (*drule spec* [*THEN mp*], *assumption*)  
**apply** (*blast intro: r-into-trancl trans-trancl* [*THEN transD*])  
**done**

**lemma** *diamond-trancl*:  $\text{diamond}(r) \Longrightarrow \text{diamond}(r^+)$   
**apply** (*simp* (*no-asm-simp*) *add: diamond-def*)  
**apply** (*rule impI* [*THEN allI*, *THEN allI*])  
**apply** (*erule trancl-induct*)  
**apply** *auto*  
**apply** (*best intro: r-into-trancl trans-trancl* [*THEN transD*])  
*dest: diamond-strip-lemmaD* +  
**done**

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

## 12.3 Results about Contraction

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

**lemmas** *contract-combE2* = *contract.dom-subset* [*THEN subsetD*, *THEN SigmaE2*]  
**and** *contract-combD1* = *contract.dom-subset* [*THEN subsetD*, *THEN SigmaD1*]  
**and** *contract-combD2* = *contract.dom-subset* [*THEN subsetD*, *THEN SigmaD2*]

**lemma** *field-contract-eq*:  $\text{field}(\text{contract}) = \text{comb}$   
**by** (*blast intro: contract.K elim!: contract-combE2*)

**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  $rtranc1$ -into- $rtranc12$ ]  
 $contract.S$  [THEN  $rtranc1$ -into- $rtranc12$ ]  
 $contract.Ap1$  [THEN  $rtranc1$ -into- $rtranc12$ ]  
 $contract.Ap2$  [THEN  $rtranc1$ -into- $rtranc12$ ]

**lemma**  $p \in comb \implies I \cdot p \rightarrow p$   
 — Example only: not used  
**unfolding**  $I$ -def **by** (blast intro: reduction-rls)

**lemma**  $comb-I$ :  $I \in comb$   
**unfolding**  $I$ -def **by** blast

## 12.4 Non-contraction results

Derive a case for each combinator constructor.

**inductive-cases**  $K$ -contractE [elim!]:  $K \rightarrow^1 r$   
**and**  $S$ -contractE [elim!]:  $S \rightarrow^1 r$   
**and**  $Ap$ -contractE [elim!]:  $p \cdot q \rightarrow^1 r$

**lemma**  $I$ -contract-E:  $I \rightarrow^1 r \implies P$   
**by** (auto simp add:  $I$ -def)

**lemma**  $K1$ -contractD:  $K \cdot p \rightarrow^1 r \implies (\exists q. r = K \cdot q \wedge p \rightarrow^1 q)$   
**by** auto

**lemma**  $Ap$ -reduce1:  $\llbracket p \rightarrow q; r \in comb \rrbracket \implies p \cdot r \rightarrow q \cdot r$   
**apply** (frule  $rtranc1$ -type [THEN subsetD, THEN SigmaD1])  
**apply** (drule field-contract-eq [THEN equalityD1, THEN subsetD])  
**apply** (erule  $rtranc1$ -induct)  
**apply** (blast intro: reduction-rls)  
**apply** (erule trans- $rtranc1$  [THEN transD])  
**apply** (blast intro: contract-combD2 reduction-rls)  
**done**

**lemma**  $Ap$ -reduce2:  $\llbracket p \rightarrow q; r \in comb \rrbracket \implies r \cdot p \rightarrow r \cdot q$   
**apply** (frule  $rtranc1$ -type [THEN subsetD, THEN SigmaD1])  
**apply** (drule field-contract-eq [THEN equalityD1, THEN subsetD])  
**apply** (erule  $rtranc1$ -induct)  
**apply** (blast intro: reduction-rls)  
**apply** (blast intro: trans- $rtranc1$  [THEN transD]  
 contract-combD2 reduction-rls)  
**done**

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

**lemma**  $KIII$ -contract1:  $K \cdot I \cdot (I \cdot I) \rightarrow^1 I$   
**by** (blast intro: comb-I)

**lemma**  $KIII$ -contract2:  $K \cdot I \cdot (I \cdot I) \rightarrow^1 K \cdot I \cdot ((K \cdot I) \cdot (K \cdot I))$   
**by** (unfold  $I$ -def) (blast intro: contract.intros)

**lemma** *KIII-contract3*:  $K \cdot I \cdot ((K \cdot I) \cdot (K \cdot I)) \rightarrow^1 I$   
**by** (*blast intro: comb-I*)

**lemma** *not-diamond-contract*:  $\neg \text{diamond}(\text{contract})$   
**unfolding** *diamond-def*  
**apply** (*blast intro: KIII-contract1 KIII-contract2 KIII-contract3*  
*elim!: I-contract-E*)  
**done**

## 12.5 Results about Parallel Contraction

For type checking: replaces  $a \Rightarrow^1 b$  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*:  $\text{field}(\text{parcontract}) = \text{comb}$   
**by** (*blast intro: parcontract.K elim!: parcontract-combE2*)

Derive a case for each combinator constructor.

**inductive-cases**  
*K-parcontractE* [*elim!*]:  $K \Rightarrow^1 r$   
**and** *S-parcontractE* [*elim!*]:  $S \Rightarrow^1 r$   
**and** *Ap-parcontractE* [*elim!*]:  $p \cdot q \Rightarrow^1 r$

**declare** *parcontract.intros* [*intro*]

## 12.6 Basic properties of parallel contraction

**lemma** *K1-parcontractD* [*dest!*]:  
 $K \cdot p \Rightarrow^1 r \implies (\exists p'. r = K \cdot p' \wedge p \Rightarrow^1 p')$   
**by** *auto*

**lemma** *S1-parcontractD* [*dest!*]:  
 $S \cdot p \Rightarrow^1 r \implies (\exists p'. r = S \cdot p' \wedge p \Rightarrow^1 p')$   
**by** *auto*

**lemma** *S2-parcontractD* [*dest!*]:  
 $S \cdot p \cdot q \Rightarrow^1 r \implies (\exists p' q'. r = S \cdot p' \cdot q' \wedge p \Rightarrow^1 p' \wedge q \Rightarrow^1 q')$   
**by** *auto*

**lemma** *diamond-parcontract*:  $\text{diamond}(\text{parcontract})$   
— Church-Rosser property for parallel contraction  
**unfolding** *diamond-def*  
**apply** (*rule impI* [*THEN allI, THEN allI*])

```

apply (erule parcontract.induct)
  apply (blast elim!: comb.free-elim intro: parcontract-combD2)+
done

```

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

```

lemma contract-imp-parcontract:  $p \rightarrow^1 q \implies p \Rightarrow^1 q$ 
  by (induct set: contract) auto

```

```

lemma reduce-imp-parreduce:  $p \rightarrow q \implies p \Rightarrow q$ 
  apply (frule rtranc1-type [THEN subsetD, THEN SigmaD1])
  apply (drule field-contract-eq [THEN equalityD1, THEN subsetD])
  apply (erule rtranc1-induct)
  apply (blast intro: r-into-tranc1)
  apply (blast intro: contract-imp-parcontract r-into-tranc1
    trans-tranc1 [THEN transD])
done

```

```

lemma parcontract-imp-reduce:  $p \Rightarrow^1 q \implies p \rightarrow q$ 
  apply (induct set: parcontract)
  apply (blast intro: reduction-rls)
  apply (blast intro: reduction-rls)
  apply (blast intro: reduction-rls)
  apply (blast intro: trans-rtranc1 [THEN transD])
  Ap-reduce1 Ap-reduce2 parcontract-combD1 parcontract-combD2)
done

```

```

lemma parreduce-imp-reduce:  $p \Rightarrow q \implies p \rightarrow q$ 
  apply (frule tranc1-type [THEN subsetD, THEN SigmaD1])
  apply (drule field-parcontract-eq [THEN equalityD1, THEN subsetD])
  apply (erule tranc1-induct, erule parcontract-imp-reduce)
  apply (erule trans-rtranc1 [THEN transD])
  apply (erule parcontract-imp-reduce)
done

```

```

lemma parreduce-iff-reduce:  $p \Rightarrow q \longleftrightarrow p \rightarrow q$ 
  by (blast intro: parreduce-imp-reduce reduce-imp-parreduce)

```

**end**

## 13 Primitive Recursive Functions: the inductive definition

**theory** Primrec **imports** ZF **begin**

Proof adopted from [4].

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



### 13.1 Basic definitions

**definition**

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

**definition**

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

**definition**

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

**definition**

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

**definition**

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

**consts**

$ACK :: i \Rightarrow i$

**primrec**

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

**abbreviation**

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

Useful special cases of evaluation.

**lemma**  $SC$ :  $\llbracket x \in \text{nat}; l \in \text{list}(\text{nat}) \rrbracket \Longrightarrow SC \text{ ' } (\text{Cons}(x, l)) = \text{succ}(x)$   
**by** (*simp add: SC-def*)

**lemma**  $CONSTANT$ :  $l \in \text{list}(\text{nat}) \Longrightarrow CONSTANT(k) \text{ ' } l = k$   
**by** (*simp add: CONSTANT-def*)

**lemma**  $PROJ$ -0:  $\llbracket x \in \text{nat}; l \in \text{list}(\text{nat}) \rrbracket \Longrightarrow PROJ(0) \text{ ' } (\text{Cons}(x, l)) = x$   
**by** (*simp add: PROJ-def*)

**lemma**  $COMP$ -1:  $l \in \text{list}(\text{nat}) \Longrightarrow COMP(g, [f]) \text{ ' } l = g \text{ ' } [f'l]$   
**by** (*simp add: COMP-def*)

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

**by** (*simp add: PREC-def*)

**lemma** *PREC-succ*:

$\llbracket x \in \text{nat}; l \in \text{list}(\text{nat}) \rrbracket$   
 $\implies \text{PREC}(f,g) \text{ ' } (\text{Cons}(\text{succ}(x),l)) =$   
 $g \text{ ' } \text{Cons}(\text{PREC}(f,g) \text{ ' } (\text{Cons}(x,l)), \text{Cons}(x,l))$   
**by** (*simp add: PREC-def*)

### 13.2 Inductive definition of the PR functions

**consts**

*prim-rec* :: *i*

**inductive**

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

**intros**

*SC*  $\in \text{prim-rec}$   
 $k \in \text{nat} \implies \text{CONSTANT}(k) \in \text{prim-rec}$   
 $i \in \text{nat} \implies \text{PROJ}(i) \in \text{prim-rec}$   
 $\llbracket g \in \text{prim-rec}; fs \in \text{list}(\text{prim-rec}) \rrbracket \implies \text{COMP}(g,fs) \in \text{prim-rec}$   
 $\llbracket f \in \text{prim-rec}; g \in \text{prim-rec} \rrbracket \implies \text{PREC}(f,g) \in \text{prim-rec}$

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

**by** (*erule subsetD [OF prim-rec.dom-subset]*)

**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*]:  $i \in \text{nat} \implies \text{ACK}(i) \in \text{prim-rec}$

**by** (*induct set: nat*) *simp-all*

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

**by** *auto*

### 13.3 Ackermann's function cases

**lemma** *ack-0*:  $j \in \text{nat} \implies \text{ack}(0,j) = \text{succ}(j)$

— *PROPERTY A 1*

**by** (*simp add: SC*)

**lemma** *ack-succ-0*:  $\text{ack}(\text{succ}(i), 0) = \text{ack}(i,1)$

```

— PROPERTY A 2
by (simp add: CONSTANT PREC-0)

lemma ack-succ-succ:
   $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(\text{succ}(i), \text{succ}(j)) = \text{ack}(i, \text{ack}(\text{succ}(i), j))$ 
— PROPERTY A 3
by (simp add: CONSTANT PREC-succ COMP-1 PROJ-0)

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
apply (induct i arbitrary: j set: nat)
  apply simp
  apply (induct-tac j)
    apply (erule-tac [2] succ-leI [THEN lt-trans1])
    apply (rule nat-0I [THEN nat-0-le, THEN lt-trans])
    apply auto
  done

lemma ack-lt-ack-succ2:  $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(i, j) < \text{ack}(i, \text{succ}(j))$ 
— PROPERTY A 5-, the single-step lemma
by (induct set: nat) (simp-all add: lt-ack2)

lemma ack-lt-mono2:  $\llbracket j < k; i \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, j) < \text{ack}(i, k)$ 
— PROPERTY A 5, monotonicity for <
apply (frule lt-nat-in-nat, assumption)
apply (erule succ-lt-induct)
  apply assumption
  apply (rule-tac [2] lt-trans)
  apply (auto intro: ack-lt-ack-succ2)
done

lemma ack-le-mono2:  $\llbracket j \leq k; i \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, j) \leq \text{ack}(i, k)$ 
— PROPERTY A 5', monotonicity for  $\leq$ 
apply (rule-tac  $f = \lambda j. \text{ack}(i, j)$  in Ord-lt-mono-imp-le-mono)
  apply (assumption | rule ack-lt-mono2 ack-type [THEN nat-into-Ord])
done

lemma ack2-le-ack1:
   $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(i, \text{succ}(j)) \leq \text{ack}(\text{succ}(i), j)$ 
— PROPERTY A 6
apply (induct-tac j)
  apply simp-all
  apply (rule ack-le-mono2)
    apply (rule lt-ack2 [THEN succ-leI, THEN le-trans])
    apply auto

```

done

**lemma** *ack-lt-ack-succ1*:  $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{ack}(i, j) < \text{ack}(\text{succ}(i), j)$   
 — PROPERTY A 7-, the single-step lemma  
**apply** (rule *ack-lt-mono2* [THEN *lt-trans2*])  
   **apply** (rule-tac [4] *ack2-le-ack1*)  
   **apply** *auto*  
 done

**lemma** *ack-lt-mono1*:  $\llbracket i < j; j \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, k) < \text{ack}(j, k)$   
 — PROPERTY A 7, monotonicity for <  
**apply** (frule *lt-nat-in-nat*, *assumption*)  
**apply** (erule *succ-lt-induct*)  
   **apply** *assumption*  
   **apply** (rule-tac [2] *lt-trans*)  
   **apply** (auto intro: *ack-lt-ack-succ1*)  
 done

**lemma** *ack-le-mono1*:  $\llbracket i \leq j; j \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{ack}(i, k) \leq \text{ack}(j, k)$   
 — PROPERTY A 7', monotonicity for ≤  
**apply** (rule-tac  $f = \lambda j. \text{ack } (j, k)$  in *Ord-lt-mono-imp-le-mono*)  
   **apply** (*assumption* | rule *ack-lt-mono1* *ack-type* [THEN *nat-into-Ord*])+  
 done

**lemma** *ack-1*:  $j \in \text{nat} \implies \text{ack}(1, j) = \text{succ}(\text{succ}(j))$   
 — PROPERTY A 8  
**by** (induct set: *nat*) *simp-all*

**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  
**by** (induct set: *nat*) (*simp-all add: ack-1*)

**lemma** *ack-nest-bound*:  
 $\llbracket i1 \in \text{nat}; i2 \in \text{nat}; j \in \text{nat} \rrbracket$   
 $\implies \text{ack}(i1, \text{ack}(i2, j)) < \text{ack}(\text{succ}(\text{succ}(i1 \# + i2)), j)$   
 — PROPERTY A 10  
**apply** (rule *lt-trans2* [OF - *ack2-le-ack1*])  
   **apply** *simp*  
   **apply** (rule *add-le-self* [THEN *ack-le-mono1*, THEN *lt-trans1*])  
   **apply** *auto*  
**apply** (force intro: *add-le-self2* [THEN *ack-lt-mono1*, THEN *ack-lt-mono2*])  
 done

**lemma** *ack-add-bound*:  
 $\llbracket i1 \in \text{nat}; i2 \in \text{nat}; j \in \text{nat} \rrbracket$   
 $\implies \text{ack}(i1, j) \# + \text{ack}(i2, j) < \text{ack}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(i1 \# + i2)))), j)$   
 — PROPERTY A 11  
**apply** (rule-tac  $j = \text{ack } (\text{succ } (1), \text{ack } (i1 \# + i2, j))$  in *lt-trans*)  
**apply** (*simp add: ack-2*)

```

apply (rule-tac [2] ack-nest-bound [THEN lt-trans2])
  apply (rule add-le-mono [THEN leI, THEN leI])
    apply (auto intro: add-le-self add-le-self2 ack-le-mono1)
done

```

**lemma** *ack-add-bound2*:

```

   $\llbracket i < \text{ack}(k,j); j \in \text{nat}; k \in \text{nat} \rrbracket$ 
   $\implies i \# + j < \text{ack}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(k))))), j)$ 
  — PROPERTY A 12.
  — Article uses existential quantifier but the ALF proof used  $k \# + \#4$ .
  — Quantified version must be nested  $\exists k'. \forall i,j \dots$ 
  apply (rule-tac  $j = \text{ack}(k,j) \# + \text{ack}(0,j)$  in lt-trans)
  apply (rule-tac [2] ack-add-bound [THEN lt-trans2])
    apply (rule add-lt-mono)
    apply auto
done

```

### 13.4 Main result

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

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

```

  unfolding SC-def
  apply (erule list.cases)
  apply (simp add: succ-iff)
  apply (simp add: ack-1 add-le-self)
done

```

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

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

```

  apply (induct-tac i)
  apply (simp add: nat-0-le)
  apply (erule lt-trans1 [OF succ-leI ack-lt-ack-succ1])
  apply auto
done

```

**lemma** *CONSTANT-case*:

```

   $\llbracket l \in \text{list}(\text{nat}); k \in \text{nat} \rrbracket \implies \text{CONSTANT}(k) \text{ ' } l < \text{ack}(k, \text{list-add}(l))$ 
  by (simp add: CONSTANT-def lt-ack1)

```

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

```

   $l \in \text{list}(\text{nat}) \implies \forall i \in \text{nat}. \text{PROJ}(i) \text{ ' } l < \text{ack}(0, \text{list-add}(l))$ 
  unfolding PROJ-def
  apply simp
  apply (erule list.induct)
  apply (simp add: nat-0-le)
  apply simp
  apply (rule ballI)

```

```

apply (erule-tac  $n = i$  in  $\text{nat}E$ )
apply (simp add: add-le-self)
apply simp
apply (erule bspec [THEN lt-trans2])
apply (rule-tac [2] add-le-self2 [THEN succ-leI])
apply auto
done

```

*COMP* case.

**lemma** *COMP-map-lemma*:

```

 $fs \in \text{list}(\{f \in \text{prim-rec}. \exists kf \in \text{nat}. \forall l \in \text{list}(\text{nat}). f'l < \text{ack}(kf, \text{list-add}(l))\})$ 
 $\implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}).$ 
 $\text{list-add}(\text{map}(\lambda f. f' l, fs)) < \text{ack}(k, \text{list-add}(l))$ 
apply (induct set: list)
apply (rule-tac  $x = 0$  in  $\text{bexI}$ )
apply (simp-all add: lt-ack1 nat-0-le)
apply clarify
apply (rule ballI [THEN bexI])
apply (rule add-lt-mono [THEN lt-trans])
apply (rule-tac [5] ack-add-bound)
apply blast
apply auto
done

```

**lemma** *COMP-case*:

```

 $\llbracket kg \in \text{nat};$ 
 $\forall l \in \text{list}(\text{nat}). g'l < \text{ack}(kg, \text{list-add}(l));$ 
 $fs \in \text{list}(\{f \in \text{prim-rec} .$ 
 $\quad \exists kf \in \text{nat}. \forall l \in \text{list}(\text{nat}).$ 
 $\quad \quad f'l < \text{ack}(kf, \text{list-add}(l))\}) \rrbracket$ 
 $\implies \exists k \in \text{nat}. \forall l \in \text{list}(\text{nat}). \text{COMP}(g, fs) l < \text{ack}(k, \text{list-add}(l))$ 
apply (simp add: COMP-def)
apply (frule list-CollectD)
apply (erule COMP-map-lemma [THEN bexE])
apply (rule ballI [THEN bexI])
apply (erule bspec [THEN lt-trans])
apply (rule-tac [2] lt-trans)
apply (rule-tac [3] ack-nest-bound)
apply (erule-tac [2] bspec [THEN ack-lt-mono2])
apply auto
done

```

*PREC* case.

**lemma** *PREC-case-lemma*:

```

 $\llbracket \forall l \in \text{list}(\text{nat}). f'l \# + \text{list-add}(l) < \text{ack}(kf, \text{list-add}(l));$ 
 $\quad \forall l \in \text{list}(\text{nat}). g'l \# + \text{list-add}(l) < \text{ack}(kg, \text{list-add}(l));$ 
 $f \in \text{prim-rec}; \quad kf \in \text{nat};$ 
 $g \in \text{prim-rec}; \quad kg \in \text{nat};$ 

```

```

  l ∈ list(nat)]
⇒ PREC(f,g) 'l #+ list-add(l) < ack(succ(kf#+kg), list-add(l))
  unfolding PREC-def
  apply (erule list.cases)
  apply (simp add: lt-trans [OF nat-le-refl lt-ack2])
  apply simp
  apply (erule ssubst) — get rid of the needless assumption
  apply (induct-tac a)
  apply simp-all

```

base case

```

  apply (rule lt-trans, erule bspec, assumption)
  apply (simp add: add-le-self [THEN ack-lt-mono1])

```

ind step

```

  apply (rule succ-leI [THEN lt-trans1])
  apply (rule-tac j = g ' ll #+ mm for ll mm in lt-trans1)
  apply (erule-tac [2] bspec)
  apply (rule nat-le-refl [THEN add-le-mono])
  apply typecheck
  apply (simp add: add-le-self2)

```

final part of the simplification

```

  apply simp
  apply (rule add-le-self2 [THEN ack-le-mono1, THEN lt-trans1])
  apply (erule-tac [4] ack-lt-mono2)
  apply auto
done

```

**lemma** *PREC-case*:

```

  [f ∈ prim-rec; kf ∈ nat;
   g ∈ prim-rec; kg ∈ nat;
   ∀ l ∈ list(nat). f 'l < ack(kf, list-add(l));
   ∀ l ∈ list(nat). g 'l < ack(kg, list-add(l))]
  ⇒ ∃ k ∈ nat. ∀ l ∈ list(nat). PREC(f,g) 'l < ack(k, list-add(l))
  apply (rule ballI [THEN bexI])
  apply (rule lt-trans1 [OF add-le-self PREC-case-lemma])
  apply typecheck
  apply (blast intro: ack-add-bound2 list-add-type)+
done

```

**lemma** *ack-bounds-prim-rec*:

```

  f ∈ prim-rec ⇒ ∃ k ∈ nat. ∀ l ∈ list(nat). f 'l < ack(k, list-add(l))
  apply (induct set: prim-rec)
  apply (auto intro: SC-case CONSTANT-case PROJ-case COMP-case PREC-case)
done

```

**theorem** *ack-not-prim-rec*:

```

  (λl ∈ list(nat). list-case(0, λx xs. ack(x,x), l)) ∉ prim-rec

```

```

apply (rule notI)
apply (drule ack-bounds-prim-rec)
apply force
done

end

```

## References

- [1] J. Camilleri and T. F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, Aug. 1992.
- [2] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall, fourth edition, 1997.
- [3] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.
- [4] N. Szasz. A machine checked proof that Ackermann’s function is not primitive recursive. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 317–338. Cambridge University Press, 1993.