

Examples of Inductive and Coinductive Definitions in HOL

Stefan Berghofer
Tobias Nipkow
Lawrence C Paulson
Markus Wenzel

November 22, 2007

Abstract

This is a collection of small examples to demonstrate Isabelle/HOL's (co)inductive definitions package. Large examples appear on many other sessions, such as Lambda, IMP, and Auth.

Contents

1	Common patterns of induction	5
1.1	Variations on statement structure	5
1.1.1	Local facts and parameters	5
1.1.2	Local definitions	5
1.1.3	Simple simultaneous goals	6
1.1.4	Compound simultaneous goals	7
1.2	Multiple rules	8
1.3	Inductive predicates	10
2	The Mutilated Chess Board Problem	11
3	Defining an Initial Algebra by Quotienting a Free Algebra	14
3.1	Defining the Free Algebra	14
3.2	Some Functions on the Free Algebra	15
3.2.1	The Set of Nonces	15
3.2.2	The Left Projection	16
3.2.3	The Right Projection	16
3.2.4	The Discriminator for Constructors	16
3.3	The Initial Algebra: A Quotiented Message Type	17
3.3.1	Characteristic Equations for the Abstract Constructors	18
3.4	The Abstract Function to Return the Set of Nonces	18
3.5	The Abstract Function to Return the Left Part	19

3.6	The Abstract Function to Return the Right Part	20
3.7	Injectivity Properties of Some Constructors	20
3.8	The Abstract Discriminator	22
4	Quotienting a Free Algebra Involving Nested Recursion	23
4.1	Defining the Free Algebra	23
4.2	Some Functions on the Free Algebra	24
4.2.1	The Set of Variables	24
4.2.2	Functions for Freeness	25
4.3	The Initial Algebra: A Quotiented Message Type	26
4.4	Every list of abstract expressions can be expressed in terms of a list of concrete expressions	27
4.4.1	Characteristic Equations for the Abstract Constructors	27
4.5	The Abstract Function to Return the Set of Variables	28
4.6	Injectivity Properties of Some Constructors	29
4.7	Injectivity of <i>FnCall</i>	30
4.8	The Abstract Discriminator	31
5	Terms over a given alphabet	32
6	Arithmetic and boolean expressions	33
7	Infinitely branching trees	34
7.1	The Brouwer ordinals, as in ZF/Induct/Brouwer.thy.	35
7.2	A WF Ordering for The Brouwer ordinals (Michael Compton)	35
8	Ordinals	36
9	Sigma algebras	38
10	Combinatory Logic example: the Church-Rosser Theorem	38
10.1	Definitions	39
10.2	Reflexive/Transitive closure preserves Church-Rosser property	40
10.3	Non-contraction results	40
10.4	Results about Parallel Contraction	41
10.5	Basic properties of parallel contraction	41
11	Meta-theory of propositional logic	42
11.1	The datatype of propositions	43
11.2	The proof system	43
11.3	The semantics	43
11.3.1	Semantics of propositional logic.	43
11.3.2	Logical consequence	43
11.4	Proof theory of propositional logic	44
11.4.1	Weakening, left and right	44

11.4.2	The deduction theorem	44
11.4.3	The cut rule	44
11.4.4	Soundness of the rules wrt truth-table semantics . . .	44
11.5	Completeness	45
11.5.1	Towards the completeness proof	45
11.6	Completeness – lemmas for reducing the set of assumptions .	46
11.6.1	Completeness theorem	46
12	Definition of type <i>l</i>list by a greatest fixed point	70
12.0.2	Sample function definitions. Item-based ones start with <i>L</i>	72
12.0.3	Simplification	72
12.1	Type checking by coinduction	73
12.2	<i>LList-corec</i> satisfies the desired recursion equation	73
12.2.1	The directions of the equality are proved separately .	74
12.3	<i>l</i> list equality as a <i>gfp</i> ; the bisimulation principle	75
12.3.1	Coinduction, using <i>LListD-Fun</i>	75
12.3.2	To show two <i>LLists</i> are equal, exhibit a bisimulation! [also admits true equality] Replace <i>A</i> by some partic- ular set, like $\{x. \text{True}\}$???	77
12.4	Finality of <i>l</i> list(<i>A</i>): Uniqueness of functions defined by core- cursion	77
12.4.1	Obsolete proof of <i>LList-corec-unique</i> : complete induc- tion, not coinduction	77
12.5	<i>Lconst</i> : defined directly by <i>lfp</i>	78
12.6	Isomorphisms	79
12.6.1	Distinctness of constructors	79
12.6.2	<i>l</i> list constructors	79
12.6.3	Injectiveness of <i>CONS</i> and <i>LCons</i>	79
12.7	Reasoning about <i>l</i> list(<i>A</i>)	80
12.8	The functional <i>Lmap</i>	80
12.8.1	Two easy results about <i>Lmap</i>	81
12.9	<i>Lappend</i> – its two arguments cause some complications! . . .	81
12.9.1	Alternative type-checking proofs for <i>Lappend</i>	82
12.10	Lazy lists as the type ' <i>a l</i> list – strongly typed versions of above	82
12.10.1	<i>l</i> list-case: case analysis for ' <i>a l</i> list	82
12.10.2	<i>l</i> list-corec: corecursion for ' <i>a l</i> list	82
12.11	Proofs about type ' <i>a l</i> list functions	83
12.12	Deriving <i>l</i> list-equalityI – <i>l</i> list equality is a bisimulation	83
12.12.1	To show two <i>l</i> lists are equal, exhibit a bisimulation! [also admits true equality]	84
12.12.2	Rules to prove the 2nd premise of <i>l</i> list-equalityI	84
12.13	The functional <i>lmap</i>	85
12.13.1	Two easy results about <i>lmap</i>	85

12.14	iterates – <i>l</i> list-fun-equalityI cannot be used!	85
12.15	A rather complex proof about iterates – cf Andy Pitts	86
12.15.1	Two lemmas about <i>natrec n x (%m. g)</i> , which is essentially $(g^{\wedge}n)(x)$	86
12.16	<i>lappend</i> – its two arguments cause some complications!	86
12.16.1	Two proofs that <i>lmap</i> distributes over <i>lappend</i>	87
13	The ”filter” functional for coinductive lists –defined by a combination of induction and coinduction	88
13.1	<i>findRel</i> : basic laws	88
13.2	Properties of <i>Domain (findRel p)</i>	88
13.3	<i>find</i> : basic equations	89
13.4	<i>lfilter</i> : basic equations	89
13.5	<i>lfilter</i> : simple facts by coinduction	90
13.6	Numerous lemmas required to prove <i>lfilter-conj</i>	91
13.7	Numerous lemmas required to prove $lfilter\ p\ (lmap\ f\ l) = lmap\ f\ (lfilter\ (\%x.\ p(f\ x))\ l)$	92
14	Mutual Induction via Iterated Inductive Definitions	93
14.1	Commands	93
14.2	Expressions	95
14.3	Equivalence of IF <i>e</i> THEN <i>c</i> ;;(WHILE <i>e</i> DO <i>c</i>) ELSE SKIP and WHILE <i>e</i> DO <i>c</i>	97
14.4	Equivalence of (IF <i>e</i> THEN <i>c</i> 1 ELSE <i>c</i> 2);; <i>c</i> and IF <i>e</i> THEN (<i>c</i> 1;; <i>c</i>) ELSE (<i>c</i> 2;; <i>c</i>)	97
14.5	Equivalence of VALOF <i>c</i> 1 RESULTIS (VALOF <i>c</i> 2 RESULTIS <i>e</i>) and VALOF <i>c</i> 1;; <i>c</i> 2 RESULTIS <i>e</i>	98
14.6	Equivalence of VALOF SKIP RESULTIS <i>e</i> and <i>e</i>	98
14.7	Equivalence of VALOF <i>x</i> := <i>e</i> RESULTIS <i>x</i> and <i>e</i>	99

1 Common patterns of induction

```
theory Common-Patterns
imports Main
begin
```

The subsequent Isar proof schemes illustrate common proof patterns supported by the generic *induct* method.

To demonstrate variations on statement (goal) structure we refer to the induction rule of Peano natural numbers: $\llbracket P\ 0; \bigwedge n. P\ n \implies P\ (Suc\ n) \rrbracket \implies P\ n$, which is the simplest case of datatype induction. We shall also see more complex (mutual) datatype inductions involving several rules. Working with inductive predicates is similar, but involves explicit facts about membership, instead of implicit syntactic typing.

1.1 Variations on statement structure

1.1.1 Local facts and parameters

Augmenting a problem by additional facts and locally fixed variables is a bread-and-butter method in many applications. This is where unwieldy object-level \forall and \implies used to occur in the past. The *induct* method works with primary means of the proof language instead.

```
lemma
  fixes  $n :: nat$ 
    and  $x :: 'a$ 
  assumes  $A\ n\ x$ 
  shows  $P\ n\ x$  using  $\langle A\ n\ x \rangle$ 
proof (induct n arbitrary: x)
  case 0
  note  $prem = \langle A\ 0\ x \rangle$ 
  show  $P\ 0\ x$  sorry
next
  case ( $Suc\ n$ )
  note  $hyp = \langle \bigwedge x. A\ n\ x \implies P\ n\ x \rangle$ 
    and  $prem = \langle A\ (Suc\ n)\ x \rangle$ 
  show  $P\ (Suc\ n)\ x$  sorry
qed
```

1.1.2 Local definitions

Here the idea is to turn sub-expressions of the problem into a defined induction variable. This is often accompanied with fixing of auxiliary parameters in the original expression, otherwise the induction step would refer invariably to particular entities. This combination essentially expresses a partially abstracted representation of inductive expressions.

```
lemma
```

```

fixes  $a :: 'a \Rightarrow nat$ 
assumes  $A (a\ x)$ 
shows  $P (a\ x)$  using  $\langle A (a\ x) \rangle$ 
proof (induct  $n \equiv a\ x$  arbitrary:  $x$ )
  case 0
    note  $prem = \langle A (a\ x) \rangle$ 
    and  $defn = \langle 0 = a\ x \rangle$ 
    show  $P (a\ x)$  sorry
  next
    case (Suc  $n$ )
    note  $hyp = \langle \bigwedge x. A (a\ x) \Longrightarrow n = a\ x \Longrightarrow P (a\ x) \rangle$ 
    and  $prem = \langle A (a\ x) \rangle$ 
    and  $defn = \langle Suc\ n = a\ x \rangle$ 
    show  $P (a\ x)$  sorry
qed

```

Observe how the local definition $n = a\ x$ recurs in the inductive cases as $0 = a\ x$ and $Suc\ n = a\ x$, according to underlying induction rule.

1.1.3 Simple simultaneous goals

The most basic simultaneous induction operates on several goals one-by-one, where each case refers to induction hypotheses that are duplicated according to the number of conclusions.

```

lemma
  fixes  $n :: nat$ 
  shows  $P\ n$  and  $Q\ n$ 
proof (induct  $n$ )
  case 0 case 1
    show  $P\ 0$  sorry
  next
    case 0 case 2
    show  $Q\ 0$  sorry
  next
    case (Suc  $n$ ) case 1
    note  $hyps = \langle P\ n \rangle \langle Q\ n \rangle$ 
    show  $P (Suc\ n)$  sorry
  next
    case (Suc  $n$ ) case 2
    note  $hyps = \langle P\ n \rangle \langle Q\ n \rangle$ 
    show  $Q (Suc\ n)$  sorry
qed

```

The split into subcases may be deferred as follows – this is particularly relevant for goal statements with local premises.

```

lemma
  fixes  $n :: nat$ 
  shows  $A\ n \Longrightarrow P\ n$ 

```

```

    and  $B\ n \implies Q\ n$ 
proof (induct n)
  case 0
  {
    case 1
    note  $\langle A\ 0 \rangle$ 
    show  $P\ 0$  sorry
  next
    case 2
    note  $\langle B\ 0 \rangle$ 
    show  $Q\ 0$  sorry
  }
next
  case (Suc n)
  note  $\langle A\ n \implies P\ n \rangle$ 
  and  $\langle B\ n \implies Q\ n \rangle$ 
  {
    case 1
    note  $\langle A\ (Suc\ n) \rangle$ 
    show  $P\ (Suc\ n)$  sorry
  next
    case 2
    note  $\langle B\ (Suc\ n) \rangle$ 
    show  $Q\ (Suc\ n)$  sorry
  }
qed

```

1.1.4 Compound simultaneous goals

The following pattern illustrates the slightly more complex situation of simultaneous goals with individual local assumptions. In compound simultaneous statements like this, local assumptions need to be included into each goal, using \implies of the Pure framework. In contrast, local parameters do not require separate \wedge prefixes here, but may be moved into the common context of the whole statement.

```

lemma
  fixes  $n :: nat$ 
  and  $x :: 'a$ 
  and  $y :: 'b$ 
  shows  $A\ n\ x \implies P\ n\ x$ 
  and  $B\ n\ y \implies Q\ n\ y$ 
proof (induct n arbitrary: x y)
  case 0
  {
    case 1
    note  $prem = \langle A\ 0\ x \rangle$ 
    show  $P\ 0\ x$  sorry
  }

```

```

{
  case 2
  note prem = ⟨B 0 y⟩
  show Q 0 y sorry
}
next
case (Suc n)
note hyps = ⟨ $\bigwedge x. A\ n\ x \implies P\ n\ x$ ⟩  $\langle \bigwedge y. B\ n\ y \implies Q\ n\ y \rangle$ 
then have some-intermediate-result sorry
{
  case 1
  note prem = ⟨A (Suc n) x⟩
  show P (Suc n) x sorry
}
{
  case 2
  note prem = ⟨B (Suc n) y⟩
  show Q (Suc n) y sorry
}
qed

```

Here *induct* provides again nested cases with numbered sub-cases, which allows to share common parts of the body context. In typical applications, there could be a long intermediate proof of general consequences of the induction hypotheses, before finishing each conclusion separately.

1.2 Multiple rules

Multiple induction rules emerge from mutual definitions of datatypes, inductive predicates, functions etc. The *induct* method accepts replicated arguments (with *and* separator), corresponding to each projection of the induction principle.

The goal statement essentially follows the same arrangement, although it might be subdivided into simultaneous sub-problems as before!

```

datatype foo = Foo1 nat | Foo2 bar
and bar = Bar1 bool | Bar2 bazar
and bazar = Bazar foo

```

The pack of induction rules for this datatype is:

```

[[ $\bigwedge nat. P1\ (Foo1\ nat); \bigwedge bar. P2\ bar \implies P1\ (Foo2\ bar); \bigwedge bool. P2\ (Bar1\ bool);$ 
 $\bigwedge bazar. P3\ bazar \implies P2\ (Bar2\ bazar); \bigwedge foo. P1\ foo \implies P3\ (Bazar\ foo)]]$ 
 $\implies P1\ foo$ 
[[ $\bigwedge nat. P1\ (Foo1\ nat); \bigwedge bar. P2\ bar \implies P1\ (Foo2\ bar); \bigwedge bool. P2\ (Bar1\ bool);$ 
 $\bigwedge bazar. P3\ bazar \implies P2\ (Bar2\ bazar); \bigwedge foo. P1\ foo \implies P3\ (Bazar\ foo)]]$ 
 $\implies P2\ bar$ 
[[ $\bigwedge nat. P1\ (Foo1\ nat); \bigwedge bar. P2\ bar \implies P1\ (Foo2\ bar); \bigwedge bool. P2\ (Bar1\ bool);$ 
 $\bigwedge bazar. P3\ bazar \implies P2\ (Bar2\ bazar); \bigwedge foo. P1\ foo \implies P3\ (Bazar\ foo)]]$ 

```


$\implies P\mathcal{J} \text{ bazar}$

This corresponds to the following basic proof pattern:

```
lemma
  fixes foo :: foo
    and bar :: bar
    and bazar :: bazar
  shows P foo
    and Q bar
    and R bazar
proof (induct foo and bar and bazar)
  case (Foo1 n)
  show P (Foo1 n) sorry
next
  case (Foo2 bar)
  note  $\langle Q \text{ bar} \rangle$ 
  show P (Foo2 bar) sorry
next
  case (Bar1 b)
  show Q (Bar1 b) sorry
next
  case (Bar2 bazar)
  note  $\langle R \text{ bazar} \rangle$ 
  show Q (Bar2 bazar) sorry
next
  case (Bazar foo)
  note  $\langle P \text{ foo} \rangle$ 
  show R (Bazar foo) sorry
qed
```

This can be combined with the previous techniques for compound statements, e.g. like this.

```
lemma
  fixes x :: 'a and y :: 'b and z :: 'c
    and foo :: foo
    and bar :: bar
    and bazar :: bazar
  shows
     $A \text{ x foo} \implies P \text{ x foo}$ 
  and
     $B1 \text{ y bar} \implies Q1 \text{ y bar}$ 
     $B2 \text{ y bar} \implies Q2 \text{ y bar}$ 
  and
     $C1 \text{ z bazar} \implies R1 \text{ z bazar}$ 
     $C2 \text{ z bazar} \implies R2 \text{ z bazar}$ 
     $C3 \text{ z bazar} \implies R3 \text{ z bazar}$ 
proof (induct foo and bar and bazar arbitrary: x and y and z)
oops
```

1.3 Inductive predicates

The most basic form of induction involving predicates (or sets) essentially eliminates a given membership fact.

```
inductive Even :: nat  $\Rightarrow$  bool where  
  zero: Even 0  
| double: Even n  $\Longrightarrow$  Even (2 * n)
```

```
lemma  
  assumes Even n  
  shows P n  
  using assms  
proof induct  
  case zero  
  show P 0 sorry  
next  
  case (double n)  
  note  $\langle \textit{Even } n \rangle$  and  $\langle \textit{P } n \rangle$   
  show P (2 * n) sorry  
qed
```

Alternatively, an initial rule statement may be proven as follows, performing “in-situ” elimination with explicit rule specification.

```
lemma Even n  $\Longrightarrow$  P n  
proof (induct rule: Even.induct)  
  oops
```

Simultaneous goals do not introduce anything new.

```
lemma  
  assumes Even n  
  shows P1 n and P2 n  
  using assms  
proof induct  
  case zero  
  {  
    case 1  
    show P1 0 sorry  
  next  
    case 2  
    show P2 0 sorry  
  }  
next  
  case (double n)  
  note  $\langle \textit{Even } n \rangle$  and  $\langle \textit{P1 } n \rangle$  and  $\langle \textit{P2 } n \rangle$   
  {  
    case 1  
    show P1 (2 * n) sorry  
  next
```

```

    case 2
    show P2 (2 * n) sorry
  }
qed

```

Working with mutual rules requires special care in composing the statement as a two-level conjunction, using lists of propositions separated by *and*. For example:

```

inductive Evn :: nat => bool and Odd :: nat => bool
where
  zero: Evn 0
| succ-Evn: Evn n ==> Odd (Suc n)
| succ-Odd: Odd n ==> Evn (Suc n)

```

```

lemma
  Evn n ==> P1 n
  Evn n ==> P2 n
  Evn n ==> P3 n
and
  Odd n ==> Q1 n
  Odd n ==> Q2 n
proof (induct rule: Evn-Odd.inducts)
  case zero
  { case 1 show P1 0 sorry }
  { case 2 show P2 0 sorry }
  { case 3 show P3 0 sorry }
next
  case (succ-Evn n)
  note <Evn n> and <P1 n> <P2 n> <P3 n>
  { case 1 show Q1 (Suc n) sorry }
  { case 2 show Q2 (Suc n) sorry }
next
  case (succ-Odd n)
  note <Odd n> and <Q1 n> <Q2 n>
  { case 1 show P1 (Suc n) sorry }
  { case 2 show P2 (Suc n) sorry }
  { case 3 show P3 (Suc n) sorry }
qed

end

```

2 The Mutilated Chess Board Problem

```

theory Mutil imports Main begin

```

The Mutilated Chess Board Problem, formalized inductively.

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

inductive-set

$tiling :: 'a \text{ set set} \Rightarrow 'a \text{ set set}$
for $A :: 'a \text{ set set}$
where
 $empty \ [simp, intro]: \{\} \in tiling \ A$
 $| \ Un \ [simp, intro]: \ [\ a \in A; t \in tiling \ A; a \cap t = \{\} \] \Rightarrow a \cup t \in tiling \ A$

inductive-set

$domino :: (nat \times nat) \text{ set set}$
where
 $horiz \ [simp]: \{(i, j), (i, Suc \ j)\} \in domino$
 $| \ vertl \ [simp]: \{(i, j), (Suc \ i, j)\} \in domino$

Sets of squares of the given colour

definition

$coloured :: nat \Rightarrow (nat \times nat) \text{ set}$ **where**
 $coloured \ b = \{(i, j). (i + j) \bmod 2 = b\}$

abbreviation

$whites :: (nat \times nat) \text{ set}$ **where**
 $whites == coloured \ 0$

abbreviation

$blacks :: (nat \times nat) \text{ set}$ **where**
 $blacks == coloured \ (Suc \ 0)$

The union of two disjoint tilings is a tiling

lemma *tiling-UnI* [intro]:

$[\ [t \in tiling \ A; u \in tiling \ A; t \cap u = \{\} \] \Rightarrow t \cup u \in tiling \ A$

apply (*induct set: tiling*)

apply (*auto simp add: Un-assoc*)

done

Chess boards

lemma *Sigma-Suc1* [simp]:

$lessThan \ (Suc \ n) \times B = (\{n\} \times B) \cup ((lessThan \ n) \times B)$

by *auto*

lemma *Sigma-Suc2* [simp]:

$A \times lessThan \ (Suc \ n) = (A \times \{n\}) \cup (A \times (lessThan \ n))$

by *auto*

lemma *sing-Times-lemma*: $(\{i\} \times \{n\}) \cup (\{i\} \times \{m\}) = \{(i, m), (i, n)\}$

by *auto*

lemma *dominoes-tile-row* [intro!]: $\{i\} \times lessThan \ (2 * n) \in tiling \ domino$

```

apply (induct n)
apply (simp-all add: Un-assoc [symmetric])
apply (rule tiling.Un)
apply (auto simp add: sing-Times-lemma)
done

```

```

lemma dominoes-tile-matrix: (lessThan m) × lessThan (2 * n) ∈ tiling domino
by (induct m) auto

```

coloured and Dominoes

```

lemma coloured-insert [simp]:
  coloured b ∩ (insert (i, j) t) =
    (if (i + j) mod 2 = b then insert (i, j) (coloured b ∩ t)
     else coloured b ∩ t)
by (auto simp add: coloured-def)

```

```

lemma domino-singletons:
  d ∈ domino ==>
    (∃ i j. whites ∩ d = {(i, j)}) ∧
    (∃ m n. blacks ∩ d = {(m, n)})
apply (erule domino.cases)
apply (auto simp add: mod-Suc)
done

```

```

lemma domino-finite [simp]: d ∈ domino ==> finite d
by (erule domino.cases, auto)

```

Tilings of dominoes

```

lemma tiling-domino-finite [simp]: t ∈ tiling domino ==> finite t
by (induct set: tiling) auto

```

declare

```

Int-Un-distrib [simp]
Diff-Int-distrib [simp]

```

```

lemma tiling-domino-0-1:
  t ∈ tiling domino ==> card(whites ∩ t) = card(blacks ∩ t)
apply (induct set: tiling)
apply (drule-tac [2] domino-singletons)
apply auto
apply (subgoal-tac ∀ p C. C ∩ a = {p} --> p ∉ t)
  — this lemma tells us that both “inserts” are non-trivial
apply (simp (no-asm-simp))
apply blast
done

```

Final argument is surprisingly complex

```

theorem gen-mutil-not-tiling:

```

```

     $t \in \text{tiling domino} \implies$ 
     $(i + j) \bmod 2 = 0 \implies (m + n) \bmod 2 = 0 \implies$ 
     $\{(i, j), (m, n)\} \subseteq t$ 
     $\implies (t - \{(i, j)\} - \{(m, n)\}) \notin \text{tiling domino}$ 
apply (rule notI)
apply (subgoal-tac
    card (whites  $\cap$  (t - {(i, j)} - {(m, n)})) <
    card (blacks  $\cap$  (t - {(i, j)} - {(m, n)})))
apply (force simp only: tiling-domino-0-1)
apply (simp add: tiling-domino-0-1 [symmetric])
apply (simp add: coloured-def card-Diff2-less)
done

```

Apply the general theorem to the well-known case

```

theorem mutil-not-tiling:
    t = lessThan (2 * Suc m)  $\times$  lessThan (2 * Suc n)
     $\implies t - \{(0, 0)\} - \{(Suc (2 * m), Suc (2 * n))\} \notin \text{tiling domino}$ 
apply (rule gen-mutil-not-tiling)
apply (blast intro!: dominoes-tile-matrix)
apply auto
done

end

```

3 Defining an Initial Algebra by Quotienting a Free Algebra

theory QuoDataType **imports** Main **begin**

3.1 Defining the Free Algebra

Messages with encryption and decryption as free constructors.

```

datatype
    freemsg = NONCE nat
            | MPAIR freemsg freemsg
            | CRYPT nat freemsg
            | DECRYPT nat freemsg

```

The equivalence relation, which makes encryption and decryption inverses provided the keys are the same.

The first two rules are the desired equations. The next four rules make the equations applicable to subterms. The last two rules are symmetry and transitivity.

```

inductive-set
    msgrel :: (freemsg * freemsg) set

```

```

and msg-rel :: [freemsg, freemsg] => bool (infixl ~ 50)
where
  X ~ Y == (X,Y) ∈ msgrel
| CD:   CRYPT K (DECRYPT K X) ~ X
| DC:   DECRYPT K (CRYPT K X) ~ X
| NONCE: NONCE N ~ NONCE N
| MPAIR: [X ~ X'; Y ~ Y'] ==> MPAIR X Y ~ MPAIR X' Y'
| CRYPT: X ~ X' ==> CRYPT K X ~ CRYPT K X'
| DECRYPT: X ~ X' ==> DECRYPT K X ~ DECRYPT K X'
| SYM:   X ~ Y ==> Y ~ X
| TRANS: [X ~ Y; Y ~ Z] ==> X ~ Z

```

Proving that it is an equivalence relation

```

lemma msgrel-refl: X ~ X
by (induct X) (blast intro: msgrel.intros)+

```

```

theorem equiv-msgrel: equiv UNIV msgrel
proof -
  have reflexive msgrel by (simp add: refl-def msgrel-refl)
  moreover have sym msgrel by (simp add: sym-def, blast intro: msgrel.SYM)
  moreover have trans msgrel by (simp add: trans-def, blast intro: msgrel.TRANS)
  ultimately show ?thesis by (simp add: equiv-def)
qed

```

3.2 Some Functions on the Free Algebra

3.2.1 The Set of Nonces

A function to return the set of nonces present in a message. It will be lifted to the initial algebra, to serve as an example of that process.

```

consts
  freenonces :: freemsg => nat set

```

```

primrec
  freenonces (NONCE N) = {N}
  freenonces (MPAIR X Y) = freenonces X ∪ freenonces Y
  freenonces (CRYPT K X) = freenonces X
  freenonces (DECRYPT K X) = freenonces X

```

This theorem lets us prove that the nonces function respects the equivalence relation. It also helps us prove that Nonce (the abstract constructor) is injective

```

theorem msgrel-imp-eq-freenonces: U ~ V ==> freenonces U = freenonces V
by (induct set: msgrel) auto

```

3.2.2 The Left Projection

A function to return the left part of the top pair in a message. It will be lifted to the initial algebra, to serve as an example of that process.

```
consts freeleft :: freemsg  $\Rightarrow$  freemsg
primrec
  freeleft (NONCE N) = NONCE N
  freeleft (MPAIR X Y) = X
  freeleft (CRYPT K X) = freeleft X
  freeleft (DECRYPT K X) = freeleft X
```

This theorem lets us prove that the left function respects the equivalence relation. It also helps us prove that MPair (the abstract constructor) is injective

```
theorem msgrel-imp-equiv-freeleft:
   $U \sim V \implies \text{freeleft } U \sim \text{freeleft } V$ 
by (induct set: msgrel) (auto intro: msgrel.intros)
```

3.2.3 The Right Projection

A function to return the right part of the top pair in a message.

```
consts freeright :: freemsg  $\Rightarrow$  freemsg
primrec
  freeright (NONCE N) = NONCE N
  freeright (MPAIR X Y) = Y
  freeright (CRYPT K X) = freeright X
  freeright (DECRYPT K X) = freeright X
```

This theorem lets us prove that the right function respects the equivalence relation. It also helps us prove that MPair (the abstract constructor) is injective

```
theorem msgrel-imp-equiv-freeright:
   $U \sim V \implies \text{freeright } U \sim \text{freeright } V$ 
by (induct set: msgrel) (auto intro: msgrel.intros)
```

3.2.4 The Discriminator for Constructors

A function to distinguish nonces, mpairs and encryptions

```
consts freediscrim :: freemsg  $\Rightarrow$  int
primrec
  freediscrim (NONCE N) = 0
  freediscrim (MPAIR X Y) = 1
  freediscrim (CRYPT K X) = freediscrim X + 2
  freediscrim (DECRYPT K X) = freediscrim X - 2
```

This theorem helps us prove $\text{Nonce } N \neq \text{MPair } X Y$

```
theorem msgrel-imp-eq-freediscrim:
```


$U \sim V \implies \text{freediscrim } U = \text{freediscrim } V$
by (induct set: msgrel) auto

3.3 The Initial Algebra: A Quotiented Message Type

typedef (Msg) msg = UNIV // msgrel
by (auto simp add: quotient-def)

The abstract message constructors

definition

$\text{Nonce} :: \text{nat} \Rightarrow \text{msg}$ **where**
 $\text{Nonce } N = \text{Abs-Msg}(\text{msgrel} \{ \text{NONCE } N \})$

definition

$\text{MPair} :: [\text{msg}, \text{msg}] \Rightarrow \text{msg}$ **where**
 $\text{MPair } X \ Y =$
 $\text{Abs-Msg} (\bigcup U \in \text{Rep-Msg } X. \bigcup V \in \text{Rep-Msg } Y. \text{msgrel} \{ \text{MPAIR } U \ V \})$

definition

$\text{Crypt} :: [\text{nat}, \text{msg}] \Rightarrow \text{msg}$ **where**
 $\text{Crypt } K \ X =$
 $\text{Abs-Msg} (\bigcup U \in \text{Rep-Msg } X. \text{msgrel} \{ \text{CRYPT } K \ U \})$

definition

$\text{Decrypt} :: [\text{nat}, \text{msg}] \Rightarrow \text{msg}$ **where**
 $\text{Decrypt } K \ X =$
 $\text{Abs-Msg} (\bigcup U \in \text{Rep-Msg } X. \text{msgrel} \{ \text{DECRYPT } K \ U \})$

Reduces equality of equivalence classes to the msgrel relation: $(\text{msgrel } \{ \{x\} = \text{msgrel } \{ \{y\} \} = (x \sim y)$

lemmas equiv-msgrel-iff = eq-equiv-class-iff [OF equiv-msgrel UNIV-I UNIV-I]

declare equiv-msgrel-iff [simp]

All equivalence classes belong to set of representatives

lemma [simp]: $\text{msgrel} \{ U \} \in \text{Msg}$
by (auto simp add: Msg-def quotient-def intro: msgrel-refl)

lemma inj-on-Abs-Msg: inj-on Abs-Msg Msg
apply (rule inj-on-inverseI)
apply (erule Abs-Msg-inverse)
done

Reduces equality on abstractions to equality on representatives

declare inj-on-Abs-Msg [THEN inj-on-iff, simp]

declare Abs-Msg-inverse [simp]

3.3.1 Characteristic Equations for the Abstract Constructors

lemma *MPair*: $MPair\ (Abs\text{-}Msg(msgrel\{\{U\}\}))\ (Abs\text{-}Msg(msgrel\{\{V\}\})) = Abs\text{-}Msg\ (msgrel\{\{MPAIR\ U\ V\}\})$

proof –

have $(\lambda U\ V. msgrel\ \{\{MPAIR\ U\ V\}\})\ respects2\ msgrel$

by $(simp\ add: congruent2\text{-}def\ msgrel.MPAIR)$

thus $?thesis$

by $(simp\ add: MPair\text{-}def\ UN\text{-}equiv\text{-}class2\ [OF\ equiv\text{-}msgrel\ equiv\text{-}msgrel])$

qed

lemma *Crypt*: $Crypt\ K\ (Abs\text{-}Msg(msgrel\{\{U\}\})) = Abs\text{-}Msg\ (msgrel\{\{CRYPT\ K\ U\}\})$

proof –

have $(\lambda U. msgrel\ \{\{CRYPT\ K\ U\}\})\ respects\ msgrel$

by $(simp\ add: congruent\text{-}def\ msgrel.CRYPT)$

thus $?thesis$

by $(simp\ add: Crypt\text{-}def\ UN\text{-}equiv\text{-}class\ [OF\ equiv\text{-}msgrel])$

qed

lemma *Decrypt*:

$Decrypt\ K\ (Abs\text{-}Msg(msgrel\{\{U\}\})) = Abs\text{-}Msg\ (msgrel\{\{DECRYPT\ K\ U\}\})$

proof –

have $(\lambda U. msgrel\ \{\{DECRYPT\ K\ U\}\})\ respects\ msgrel$

by $(simp\ add: congruent\text{-}def\ msgrel.DECRYPT)$

thus $?thesis$

by $(simp\ add: Decrypt\text{-}def\ UN\text{-}equiv\text{-}class\ [OF\ equiv\text{-}msgrel])$

qed

Case analysis on the representation of a msg as an equivalence class.

lemma *eq-Abs-Msg* $[case\text{-}names\ Abs\text{-}Msg, cases\ type: msg]:$

$(!!U. z = Abs\text{-}Msg(msgrel\{\{U\}\}) ==> P) ==> P$

apply $(rule\ Rep\text{-}Msg\ [of\ z, unfolded\ Msg\text{-}def, THEN\ quotientE])$

apply $(drule\ arg\text{-}cong\ [where\ f = Abs\text{-}Msg])$

apply $(auto\ simp\ add: Rep\text{-}Msg\text{-}inverse\ intro: msgrel\text{-}refl)$

done

Establishing these two equations is the point of the whole exercise

theorem *CD-eq* $[simp]: Crypt\ K\ (Decrypt\ K\ X) = X$

by $(cases\ X, simp\ add: Crypt\ Decrypt\ CD)$

theorem *DC-eq* $[simp]: Decrypt\ K\ (Crypt\ K\ X) = X$

by $(cases\ X, simp\ add: Crypt\ Decrypt\ DC)$

3.4 The Abstract Function to Return the Set of Nonces

definition

$nonces :: msg \Rightarrow nat\ set$ **where**

$nonces\ X = (\bigcup U \in Rep\text{-}Msg\ X. freenonces\ U)$

lemma *nonces-congruent: freenonces respects msgrel*
by (*simp add: congruent-def msgrel-imp-eq-freenonces*)

Now prove the four equations for *nonces*

lemma *nonces-Nonce [simp]: nonces (Nonce N) = {N}*
by (*simp add: nonces-def Nonce-def*
UN-equiv-class [OF equiv-msgrel nonces-congruent])

lemma *nonces-MPair [simp]: nonces (MPair X Y) = nonces X \cup nonces Y*
apply (*cases X, cases Y*)
apply (*simp add: nonces-def MPair*
UN-equiv-class [OF equiv-msgrel nonces-congruent])
done

lemma *nonces-Crypt [simp]: nonces (Crypt K X) = nonces X*
apply (*cases X*)
apply (*simp add: nonces-def Crypt*
UN-equiv-class [OF equiv-msgrel nonces-congruent])
done

lemma *nonces-Decrypt [simp]: nonces (Decrypt K X) = nonces X*
apply (*cases X*)
apply (*simp add: nonces-def Decrypt*
UN-equiv-class [OF equiv-msgrel nonces-congruent])
done

3.5 The Abstract Function to Return the Left Part

definition

left :: *msg* \Rightarrow *msg* **where**
left X = *Abs-Msg* ($\bigcup U \in \text{Rep-Msg } X. \text{msgrel} \{ \text{freeleft } U \}$)

lemma *left-congruent: ($\lambda U. \text{msgrel} \{ \text{freeleft } U \}$) respects msgrel*
by (*simp add: congruent-def msgrel-imp-eqv-freeleft*)

Now prove the four equations for *left*

lemma *left-Nonce [simp]: left (Nonce N) = Nonce N*
by (*simp add: left-def Nonce-def*
UN-equiv-class [OF equiv-msgrel left-congruent])

lemma *left-MPair [simp]: left (MPair X Y) = X*
apply (*cases X, cases Y*)
apply (*simp add: left-def MPair*
UN-equiv-class [OF equiv-msgrel left-congruent])
done

lemma *left-Crypt [simp]: left (Crypt K X) = left X*
apply (*cases X*)
apply (*simp add: left-def Crypt*)

```

UN-equiv-class [OF equiv-msgrel left-congruent])
done

lemma left-Decrypt [simp]: left (Decrypt K X) = left X
apply (cases X)
apply (simp add: left-def Decrypt
UN-equiv-class [OF equiv-msgrel left-congruent])
done

```

3.6 The Abstract Function to Return the Right Part

definition

```

right :: msg ⇒ msg where
right X = Abs-Msg (⋃ U ∈ Rep-Msg X. msgrel “{freeright U}”)

```

lemma right-congruent: $(\lambda U. \text{msgrel “}\{ \text{freeright } U \})$ respects msgrel
by (simp add: congruent-def msgrel-imp-eqv-freeright)

Now prove the four equations for *right*

```

lemma right-Nonce [simp]: right (Nonce N) = Nonce N
by (simp add: right-def Nonce-def
UN-equiv-class [OF equiv-msgrel right-congruent])

```

```

lemma right-MPair [simp]: right (MPair X Y) = Y
apply (cases X, cases Y)
apply (simp add: right-def MPair
UN-equiv-class [OF equiv-msgrel right-congruent])
done

```

```

lemma right-Crypt [simp]: right (Crypt K X) = right X
apply (cases X)
apply (simp add: right-def Crypt
UN-equiv-class [OF equiv-msgrel right-congruent])
done

```

```

lemma right-Decrypt [simp]: right (Decrypt K X) = right X
apply (cases X)
apply (simp add: right-def Decrypt
UN-equiv-class [OF equiv-msgrel right-congruent])
done

```

3.7 Injectivity Properties of Some Constructors

lemma NONCE-imp-eq: $\text{NONCE } m \sim \text{NONCE } n \implies m = n$
by (drule msgrel-imp-eq-freenonces, simp)

Can also be proved using the function *nonces*

```

lemma Nonce-Nonce-eq [iff]: (Nonce m = Nonce n) = (m = n)
by (auto simp add: Nonce-def msgrel-refl dest: NONCE-imp-eq)

```

lemma *MPAIR-imp-eqv-left*: $MPAIR\ X\ Y \sim MPAIR\ X'\ Y' \implies X \sim X'$
by (*drule msgrel-imp-eqv-freeleft, simp*)

lemma *MPair-imp-eq-left*:
assumes *eq*: $MPair\ X\ Y = MPair\ X'\ Y'$ **shows** $X = X'$
proof –
from *eq*
have $left\ (MPair\ X\ Y) = left\ (MPair\ X'\ Y')$ **by** *simp*
thus *?thesis* **by** *simp*
qed

lemma *MPAIR-imp-eqv-right*: $MPAIR\ X\ Y \sim MPAIR\ X'\ Y' \implies Y \sim Y'$
by (*drule msgrel-imp-eqv-freeright, simp*)

lemma *MPair-imp-eq-right*: $MPair\ X\ Y = MPair\ X'\ Y' \implies Y = Y'$
apply (*cases X, cases X', cases Y, cases Y'*)
apply (*simp add: MPair*)
apply (*erule MPAIR-imp-eqv-right*)
done

theorem *MPair-MPair-eq [iff]*: $(MPair\ X\ Y = MPair\ X'\ Y') = (X=X' \ \&\ Y=Y')$
by (*blast dest: MPair-imp-eq-left MPair-imp-eq-right*)

lemma *NONCE-neqv-MPAIR*: $NONCE\ m \sim MPAIR\ X\ Y \implies False$
by (*drule msgrel-imp-eq-freediscrim, simp*)

theorem *Nonce-neq-MPair [iff]*: $Nonce\ N \neq MPair\ X\ Y$
apply (*cases X, cases Y*)
apply (*simp add: Nonce-def MPair*)
apply (*blast dest: NONCE-neqv-MPAIR*)
done

Example suggested by a referee

theorem *Crypt-Nonce-neq-Nonce*: $Crypt\ K\ (Nonce\ M) \neq Nonce\ N$
by (*auto simp add: Nonce-def Crypt dest: msgrel-imp-eq-freediscrim*)

...and many similar results

theorem *Crypt2-Nonce-neq-Nonce*: $Crypt\ K\ (Crypt\ K'\ (Nonce\ M)) \neq Nonce\ N$
by (*auto simp add: Nonce-def Crypt dest: msgrel-imp-eq-freediscrim*)

theorem *Crypt-Crypt-eq [iff]*: $(Crypt\ K\ X = Crypt\ K\ X') = (X=X')$
proof
assume $Crypt\ K\ X = Crypt\ K\ X'$
hence $Decrypt\ K\ (Crypt\ K\ X) = Decrypt\ K\ (Crypt\ K\ X')$ **by** *simp*
thus $X = X'$ **by** *simp*
next
assume $X = X'$

```

    thus Crypt K X = Crypt K X' by simp
qed

theorem Decrypt-Decrypt-eq [iff]: (Decrypt K X = Decrypt K X') = (X=X')
proof
  assume Decrypt K X = Decrypt K X'
  hence Crypt K (Decrypt K X) = Crypt K (Decrypt K X') by simp
  thus X = X' by simp
next
  assume X = X'
  thus Decrypt K X = Decrypt K X' by simp
qed

lemma msg-induct [case-names Nonce MPair Crypt Decrypt, cases type: msg]:
  assumes N:  $\bigwedge N. P \text{ (Nonce } N)$ 
    and M:  $\bigwedge X Y. \llbracket P \text{ } X; P \text{ } Y \rrbracket \implies P \text{ (MPair } X \text{ } Y)$ 
    and C:  $\bigwedge K X. P \text{ } X \implies P \text{ (Crypt } K \text{ } X)$ 
    and D:  $\bigwedge K X. P \text{ } X \implies P \text{ (Decrypt } K \text{ } X)$ 
  shows P msg
proof (cases msg)
  case (Abs-Msg U)
  have P (Abs-Msg (msgrel “ {U}))
  proof (induct U)
    case (NONCE N)
    with N show ?case by (simp add: Nonce-def)
  next
    case (MPAIR X Y)
    with M [of Abs-Msg (msgrel “ {X}) Abs-Msg (msgrel “ {Y})]
    show ?case by (simp add: MPair)
  next
    case (CRYPT K X)
    with C [of Abs-Msg (msgrel “ {X})]
    show ?case by (simp add: Crypt)
  next
    case (DECRYPT K X)
    with D [of Abs-Msg (msgrel “ {X})]
    show ?case by (simp add: Decrypt)
  qed
  with Abs-Msg show ?thesis by (simp only:)
qed

```

3.8 The Abstract Discriminator

However, as *Crypt-Nonce-neq-Nonce* above illustrates, we don't need this function in order to prove discrimination theorems.

definition

```

discrim :: msg  $\Rightarrow$  int where
discrim X = contents ( $\bigcup U \in \text{Rep-Msg } X. \{\text{freediscrim } U\}$ )

```

lemma *discrim-congruent*: $(\lambda U. \{freediscrim\ U\})$ respects msgrel
by (*simp add: congruent-def msgrel-imp-eq-freediscrim*)

Now prove the four equations for *discrim*

lemma *discrim-Nonce* [*simp*]: *discrim* (*Nonce* *N*) = 0
by (*simp add: discrim-def Nonce-def*
UN-equiv-class [OF equiv-msgrel discrim-congruent])

lemma *discrim-MPair* [*simp*]: *discrim* (*MPair* *X* *Y*) = 1
apply (*cases* *X*, *cases* *Y*)
apply (*simp add: discrim-def MPair*
UN-equiv-class [OF equiv-msgrel discrim-congruent])
done

lemma *discrim-Crypt* [*simp*]: *discrim* (*Crypt* *K* *X*) = *discrim* *X* + 2
apply (*cases* *X*)
apply (*simp add: discrim-def Crypt*
UN-equiv-class [OF equiv-msgrel discrim-congruent])
done

lemma *discrim-Decrypt* [*simp*]: *discrim* (*Decrypt* *K* *X*) = *discrim* *X* - 2
apply (*cases* *X*)
apply (*simp add: discrim-def Decrypt*
UN-equiv-class [OF equiv-msgrel discrim-congruent])
done

end

4 Quotienting a Free Algebra Involving Nested Recursion

theory *QuoNestedDataType* **imports** *Main* **begin**

4.1 Defining the Free Algebra

Messages with encryption and decryption as free constructors.

datatype
freeExp = *VAR* *nat*
| *PLUS* *freeExp* *freeExp*
| *FNCALL* *nat* *freeExp* *list*

The equivalence relation, which makes PLUS associative.

The first rule is the desired equation. The next three rules make the equations applicable to subterms. The last two rules are symmetry and transi-

tivity.

inductive-set

```

  exprel :: (freeExp * freeExp) set
  and exp-rel :: [freeExp, freeExp] => bool (infixl ~ 50)
  where
    X ~ Y == (X, Y) ∈ exprel
  | ASSOC: PLUS X (PLUS Y Z) ~ PLUS (PLUS X Y) Z
  | VAR: VAR N ~ VAR N
  | PLUS:  $\llbracket X \sim X'; Y \sim Y' \rrbracket \implies PLUS\ X\ Y \sim PLUS\ X'\ Y'$ 
  | FNCALL:  $(Xs, Xs') \in listrel\ exprel \implies FNCALL\ F\ Xs \sim FNCALL\ F\ Xs'$ 
  | SYM:  $X \sim Y \implies Y \sim X$ 
  | TRANS:  $\llbracket X \sim Y; Y \sim Z \rrbracket \implies X \sim Z$ 
  monos listrel-mono

```

Proving that it is an equivalence relation

```

lemma exprel-refl: X ~ X
  and list-exprel-refl: (Xs, Xs) ∈ listrel(exprel)
  by (induct X and Xs) (blast intro: exprel.intros listrel.intros)+

```

theorem equiv-exprel: equiv UNIV exprel

proof –

```

  have reflexive exprel by (simp add: refl-def exprel-refl)
  moreover have sym exprel by (simp add: sym-def, blast intro: exprel.SYM)
  moreover have trans exprel by (simp add: trans-def, blast intro: exprel.TRANS)
  ultimately show ?thesis by (simp add: equiv-def)

```

qed

theorem equiv-list-exprel: equiv UNIV (listrel exprel)

using equiv-listrel [OF equiv-exprel] by simp

lemma FNCALL-Nil: FNCALL F [] ~ FNCALL F []

apply (rule exprel.intros)

apply (rule listrel.intros)

done

lemma FNCALL-Cons:

```

 $\llbracket X \sim X'; (Xs, Xs') \in listrel(exprel) \rrbracket$ 
 $\implies FNCALL\ F\ (X \# Xs) \sim FNCALL\ F\ (X' \# Xs')$ 

```

by (blast intro: exprel.intros listrel.intros)

4.2 Some Functions on the Free Algebra

4.2.1 The Set of Variables

A function to return the set of variables present in a message. It will be lifted to the initial algebra, to serve as an example of that process. Note that the "free" refers to the free datatype rather than to the concept of a free variable.

consts

$freevars :: freeExp \Rightarrow nat\ set$
 $freevars-list :: freeExp\ list \Rightarrow nat\ set$

primrec

$freevars\ (VAR\ N) = \{N\}$
 $freevars\ (PLUS\ X\ Y) = freevars\ X \cup freevars\ Y$
 $freevars\ (FNCALL\ F\ Xs) = freevars-list\ Xs$

$freevars-list\ [] = \{\}$
 $freevars-list\ (X\ \# \ Xs) = freevars\ X \cup freevars-list\ Xs$

This theorem lets us prove that the vars function respects the equivalence relation. It also helps us prove that Variable (the abstract constructor) is injective

theorem $exprel-imp-eq-freevars: U \sim V \implies freevars\ U = freevars\ V$
apply ($induct\ set: exprel$)
apply ($erule-tac\ [4]\ listrel.induct$)
apply ($simp-all\ add: Un-assoc$)
done

4.2.2 Functions for Freeness

A discriminator function to distinguish vars, sums and function calls

consts $freediscrim :: freeExp \Rightarrow int$

primrec

$freediscrim\ (VAR\ N) = 0$
 $freediscrim\ (PLUS\ X\ Y) = 1$
 $freediscrim\ (FNCALL\ F\ Xs) = 2$

theorem $exprel-imp-eq-freediscrim:$

$U \sim V \implies freediscrim\ U = freediscrim\ V$

by ($induct\ set: exprel$) *auto*

This function, which returns the function name, is used to prove part of the injectivity property for FnCall.

consts $freefun :: freeExp \Rightarrow nat$

primrec

$freefun\ (VAR\ N) = 0$
 $freefun\ (PLUS\ X\ Y) = 0$
 $freefun\ (FNCALL\ F\ Xs) = F$

theorem $exprel-imp-eq-freefun:$

$U \sim V \implies freefun\ U = freefun\ V$

by ($induct\ set: exprel$) ($simp-all\ add: listrel.intros$)

This function, which returns the list of function arguments, is used to prove part of the injectivity property for FnCall.

```

consts freeargs      :: freeExp  $\Rightarrow$  freeExp list
primrec
  freeargs (VAR N) = []
  freeargs (PLUS X Y) = []
  freeargs (FNCALL F Xs) = Xs

theorem exprel-imp-eqv-freeargs:
   $U \sim V \implies (\text{freeargs } U, \text{freeargs } V) \in \text{listrel exprel}$ 
apply (induct set: exprel)
apply (erule-tac [4] listrel.induct)
apply (simp-all add: listrel.intros)
apply (blast intro: symD [OF equiv.sym [OF equiv-list-exprel]])
apply (blast intro: transD [OF equiv.trans [OF equiv-list-exprel]])
done

```

4.3 The Initial Algebra: A Quotiented Message Type

```

typedef (Exp) exp = UNIV // exprel
by (auto simp add: quotient-def)

```

The abstract message constructors

```

definition
  Var :: nat  $\Rightarrow$  exp where
  Var N = Abs-Exp(exprel "{VAR N}")

```

```

definition
  Plus :: [exp, exp]  $\Rightarrow$  exp where
  Plus X Y =
    Abs-Exp ( $\bigcup U \in \text{Rep-Exp } X. \bigcup V \in \text{Rep-Exp } Y. \text{exprel "{PLUS U V}"}$ )

```

```

definition
  FnCall :: [nat, exp list]  $\Rightarrow$  exp where
  FnCall F Xs =
    Abs-Exp ( $\bigcup Us \in \text{listset } (\text{map Rep-Exp } Xs). \text{exprel "{FNCALL F Us}"}$ )

```

Reduces equality of equivalence classes to the *exprel* relation: (*exprel* "{x}" = *exprel* "{y}" = ($x \sim y$)

```

lemmas equiv-exprel-iff = eq-equiv-class-iff [OF equiv-exprel UNIV-I UNIV-I]

```

```

declare equiv-exprel-iff [simp]

```

All equivalence classes belong to set of representatives

```

lemma [simp]: exprel "{U}"  $\in$  Exp
by (auto simp add: Exp-def quotient-def intro: exprel-refl)

```

```

lemma inj-on-Abs-Exp: inj-on Abs-Exp Exp
apply (rule inj-on-inverseI)
apply (erule Abs-Exp-inverse)
done

```

Reduces equality on abstractions to equality on representatives

declare *inj-on-Abs-Exp* [*THEN inj-on-iff*, *simp*]

declare *Abs-Exp-inverse* [*simp*]

Case analysis on the representation of a exp as an equivalence class.

lemma *eq-Abs-Exp* [*case-names Abs-Exp*, *cases type: exp*]:
 ($!!U. z = \text{Abs-Exp}(\text{exprel}\{\{U\}\}) \implies P \implies P$)
apply (*rule Rep-Exp* [*of z*, *unfolded Exp-def*, *THEN quotientE*])
apply (*drule arg-cong* [**where** $f = \text{Abs-Exp}$])
apply (*auto simp add: Rep-Exp-inverse intro: exprel-refl*)
done

4.4 Every list of abstract expressions can be expressed in terms of a list of concrete expressions

definition

Abs-ExpList :: *freeExp list* => *exp list* **where**
Abs-ExpList *Xs* = *map* ($\%U. \text{Abs-Exp}(\text{exprel}\{\{U\}\})$) *Xs*

lemma *Abs-ExpList-Nil* [*simp*]: *Abs-ExpList* [] == []
by (*simp add: Abs-ExpList-def*)

lemma *Abs-ExpList-Cons* [*simp*]:
 $\text{Abs-ExpList } (X \# Xs) == \text{Abs-Exp } (\text{exprel}\{\{X\}\}) \# \text{Abs-ExpList } Xs$
by (*simp add: Abs-ExpList-def*)

lemma *ExpList-rep*: $\exists Us. z = \text{Abs-ExpList } Us$
apply (*induct z*)
apply (*rule-tac* [2] $z=a$ **in** *eq-Abs-Exp*)
apply (*auto simp add: Abs-ExpList-def Cons-eq-map-conv intro: exprel-refl*)
done

lemma *eq-Abs-ExpList* [*case-names Abs-ExpList*]:
 ($!!Us. z = \text{Abs-ExpList } Us \implies P \implies P$)
by (*rule exE* [*OF ExpList-rep*], *blast*)

4.4.1 Characteristic Equations for the Abstract Constructors

lemma *Plus*: $\text{Plus } (\text{Abs-Exp}(\text{exprel}\{\{U\}\})) (\text{Abs-Exp}(\text{exprel}\{\{V\}\})) =$
 $\text{Abs-Exp } (\text{exprel}\{\{PLUS \ U \ V\}\})$

proof –

have ($\lambda U \ V. \text{exprel } \{ \{ PLUS \ U \ V \} \}$) *respects2 exprel*

by (*simp add: congruent2-def exprel.PLUS*)

thus *?thesis*

by (*simp add: Plus-def UN-equiv-class2* [*OF equiv-exprel equiv-exprel*])

qed

It is not clear what to do with FnCall: it's argument is an abstraction of

an *exp list*. Is it just Nil or Cons? What seems to work best is to regard an *exp list* as a *listrel exprel* equivalence class

This theorem is easily proved but never used. There's no obvious way even to state the analogous result, *FnCall-Cons*.

lemma *FnCall-Nil*: $\text{FnCall } F \ [] = \text{Abs-Exp } (\text{exprel} \{ \text{FNCALL } F \ [] \})$
by (*simp add: FnCall-def*)

lemma *FnCall-respects*:
 $(\lambda Us. \text{exprel} \{ \text{FNCALL } F \ Us \}) \text{ respects } (\text{listrel exprel})$
by (*simp add: congruent-def exprel.FNCALL*)

lemma *FnCall-sing*:
 $\text{FnCall } F \ [\text{Abs-Exp}(\text{exprel} \{ U \})] = \text{Abs-Exp } (\text{exprel} \{ \text{FNCALL } F \ [U] \})$
proof –
have $(\lambda U. \text{exprel} \{ \text{FNCALL } F \ [U] \}) \text{ respects exprel}$
by (*simp add: congruent-def FNCALL-Cons listrel.intros*)
thus ?thesis
by (*simp add: FnCall-def UN-equiv-class [OF equiv-exprel]*)
qed

lemma *listset-Rep-Exp-Abs-Exp*:
 $\text{listset } (\text{map Rep-Exp } (\text{Abs-ExpList } Us)) = \text{listrel exprel} \{ \{ Us \} \}$
by (*induct Us (simp-all add: listrel-Cons Abs-ExpList-def)*)

lemma *FnCall*:
 $\text{FnCall } F \ (\text{Abs-ExpList } Us) = \text{Abs-Exp } (\text{exprel} \{ \text{FNCALL } F \ Us \})$
proof –
have $(\lambda Us. \text{exprel} \{ \text{FNCALL } F \ Us \}) \text{ respects } (\text{listrel exprel})$
by (*simp add: congruent-def exprel.FNCALL*)
thus ?thesis
by (*simp add: FnCall-def UN-equiv-class [OF equiv-list-exprel] listset-Rep-Exp-Abs-Exp*)
qed

Establishing this equation is the point of the whole exercise

theorem *Plus-assoc*: $\text{Plus } X \ (\text{Plus } Y \ Z) = \text{Plus } (\text{Plus } X \ Y) \ Z$
by (*cases X, cases Y, cases Z, simp add: Plus exprel.ASSOC*)

4.5 The Abstract Function to Return the Set of Variables

definition

$\text{vars} :: \text{exp} \Rightarrow \text{nat set}$ **where**
 $\text{vars } X = (\bigcup U \in \text{Rep-Exp } X. \text{freevars } U)$

lemma *vars-respects*: *freevars respects exprel*
by (*simp add: congruent-def exprel-imp-eq-freevars*)

The extension of the function *vars* to lists

```

consts vars-list :: exp list  $\Rightarrow$  nat set
primrec
  vars-list [] = {}
  vars-list(E#Es) = vars E  $\cup$  vars-list Es

```

Now prove the three equations for *vars*

```

lemma vars-Variable [simp]: vars (Var N) = {N}
by (simp add: vars-def Var-def
  UN-equiv-class [OF equiv-exprel vars-respects])

```

```

lemma vars-Plus [simp]: vars (Plus X Y) = vars X  $\cup$  vars Y
apply (cases X, cases Y)
apply (simp add: vars-def Plus
  UN-equiv-class [OF equiv-exprel vars-respects])
done

```

```

lemma vars-FnCall [simp]: vars (FnCall F Xs) = vars-list Xs
apply (cases Xs rule: eq-Abs-ExpList)
apply (simp add: FnCall)
apply (induct-tac Us)
apply (simp-all add: vars-def UN-equiv-class [OF equiv-exprel vars-respects])
done

```

```

lemma vars-FnCall-Nil: vars (FnCall F Nil) = {}
by simp

```

```

lemma vars-FnCall-Cons: vars (FnCall F (X#Xs)) = vars X  $\cup$  vars-list Xs
by simp

```

4.6 Injectivity Properties of Some Constructors

```

lemma VAR-imp-eq: VAR m  $\sim$  VAR n  $\implies$  m = n
by (drule exprel-imp-eq-freevars, simp)

```

Can also be proved using the function *vars*

```

lemma Var-Var-eq [iff]: (Var m = Var n) = (m = n)
by (auto simp add: Var-def exprel-refl dest: VAR-imp-eq)

```

```

lemma VAR-neqv-PLUS: VAR m  $\sim$  PLUS X Y  $\implies$  False
by (drule exprel-imp-eq-freediscrim, simp)

```

```

theorem Var-neq-Plus [iff]: Var N  $\neq$  Plus X Y
apply (cases X, cases Y)
apply (simp add: Var-def Plus)
apply (blast dest: VAR-neqv-PLUS)
done

```

```

theorem Var-neq-FnCall [iff]: Var N  $\neq$  FnCall F Xs
apply (cases Xs rule: eq-Abs-ExpList)

```

```

apply (auto simp add: FnCall Var-def)
apply (drule exprel-imp-eq-freediscrim, simp)
done

```

4.7 Injectivity of *FnCall*

definition

```

fun :: exp  $\Rightarrow$  nat where
fun X = contents ( $\bigcup U \in \text{Rep-Exp } X. \{\text{freefun } U\}$ )

```

lemma *fun-respects*: ($\%U. \{\text{freefun } U\}$) *respects exprel*
by (simp add: congruent-def exprel-imp-eq-freefun)

lemma *fun-FnCall* [simp]: *fun* (*FnCall* *F* *Xs*) = *F*
apply (cases *Xs* rule: eq-Abs-ExpList)
apply (simp add: FnCall fun-def UN-equiv-class [OF equiv-exprel fun-respects])
done

definition

```

args :: exp  $\Rightarrow$  exp list where
args X = contents ( $\bigcup U \in \text{Rep-Exp } X. \{\text{Abs-ExpList } (\text{freeargs } U)\}$ )

```

This result can probably be generalized to arbitrary equivalence relations, but with little benefit here.

lemma *Abs-ExpList-eq*:

```

(y, z)  $\in$  listrel exprel  $\implies$  Abs-ExpList (y) = Abs-ExpList (z)
by (induct set: listrel) simp-all

```

lemma *args-respects*: ($\%U. \{\text{Abs-ExpList } (\text{freeargs } U)\}$) *respects exprel*
by (simp add: congruent-def Abs-ExpList-eq exprel-imp-eqv-freeargs)

lemma *args-FnCall* [simp]: *args* (*FnCall* *F* *Xs*) = *Xs*
apply (cases *Xs* rule: eq-Abs-ExpList)
apply (simp add: FnCall args-def UN-equiv-class [OF equiv-exprel args-respects])
done

lemma *FnCall-FnCall-eq* [iff]:

```

(FnCall F Xs = FnCall F' Xs') = (F=F' & Xs=Xs')

```

proof

```

assume FnCall F Xs = FnCall F' Xs'
hence fun (FnCall F Xs) = fun (FnCall F' Xs')
and args (FnCall F Xs) = args (FnCall F' Xs') by auto
thus F=F' & Xs=Xs' by simp

```

next

```

assume F=F' & Xs=Xs' thus FnCall F Xs = FnCall F' Xs' by simp

```

qed

4.8 The Abstract Discriminator

However, as *FnCall-Var-neq-Var* illustrates, we don't need this function in order to prove discrimination theorems.

definition

discrim :: *exp* \Rightarrow *int* **where**
discrim *X* = *contents* ($\bigcup U \in \text{Rep-Exp } X. \{\text{freediscrim } U\}$)

lemma *discrim-respects*: ($\lambda U. \{\text{freediscrim } U\}$) *respects exprel*
by (*simp add: congruent-def exprel-imp-eq-freediscrim*)

Now prove the four equations for *discrim*

lemma *discrim-Var* [*simp*]: *discrim* (*Var* *N*) = 0
by (*simp add: discrim-def Var-def*
UN-equiv-class [OF equiv-exprel discrim-respects])

lemma *discrim-Plus* [*simp*]: *discrim* (*Plus* *X* *Y*) = 1
apply (*cases X, cases Y*)
apply (*simp add: discrim-def Plus*
UN-equiv-class [OF equiv-exprel discrim-respects])

done

lemma *discrim-FnCall* [*simp*]: *discrim* (*FnCall* *F* *Xs*) = 2
apply (*rule-tac z=Xs in eq-Abs-ExpList*)
apply (*simp add: discrim-def FnCall*
UN-equiv-class [OF equiv-exprel discrim-respects])

done

The structural induction rule for the abstract type

theorem *exp-inducts*:

assumes *V*: $\bigwedge \text{nat}. P1 \text{ (Var nat)}$
and *P*: $\bigwedge \text{exp1 exp2}. \llbracket P1 \text{ exp1}; P1 \text{ exp2} \rrbracket \Longrightarrow P1 \text{ (Plus exp1 exp2)}$
and *F*: $\bigwedge \text{nat list}. P2 \text{ list} \Longrightarrow P1 \text{ (FnCall nat list)}$
and *Nil*: $P2 []$
and *Cons*: $\bigwedge \text{exp list}. \llbracket P1 \text{ exp}; P2 \text{ list} \rrbracket \Longrightarrow P2 \text{ (exp \# list)}$
shows *P1 exp and P2 list*

proof –

obtain *U* **where** *exp*: *exp* = (*Abs-Exp* (*exprel* “ {*U*})) **by** (*cases exp*)
obtain *Us* **where** *list*: *list* = *Abs-ExpList* *Us* **by** (*rule eq-Abs-ExpList*)
have *P1* (*Abs-Exp* (*exprel* “ {*U*})) **and** *P2* (*Abs-ExpList* *Us*)
proof (*induct U and Us*)
case (*VAR nat*)
with *V* **show** ?*case* **by** (*simp add: Var-def*)
next
case (*PLUS X Y*)
with *P* [*of Abs-Exp* (*exprel* “ {*X*})) *Abs-Exp* (*exprel* “ {*Y*})]
show ?*case* **by** (*simp add: Plus*)
next
case (*FNCALL nat list*)

```

    with F [of Abs-ExpList list]
    show ?case by (simp add: FnCall)
next
  case Nil-freeExp
  with Nil show ?case by simp
next
  case Cons-freeExp
  with Cons show ?case by simp
qed
with exp and list show P1 exp and P2 list by (simp-all only:)
qed
end

```

5 Terms over a given alphabet

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

```

datatype ('a, 'b) term =
  Var 'a
| App 'b ('a, 'b) term list

```

Substitution function on terms

```

consts
  subst-term :: ('a => ('a, 'b) term) => ('a, 'b) term => ('a, 'b) term
  subst-term-list ::
    ('a => ('a, 'b) term) => ('a, 'b) term list => ('a, 'b) term list

```

```

primrec
  subst-term f (Var a) = f a
  subst-term f (App b ts) = App b (subst-term-list f ts)

  subst-term-list f [] = []
  subst-term-list f (t # ts) =
    subst-term f t # subst-term-list f ts

```

A simple theorem about composition of substitutions

```

lemma subst-comp:
  subst-term (subst-term f1 o f2) t =
    subst-term f1 (subst-term f2 t)
and subst-term-list (subst-term f1 o f2) ts =
  subst-term-list f1 (subst-term-list f2 ts)
by (induct t and ts) simp-all

```

Alternative induction rule

lemma


```

assumes var: !!v. P (Var v)
  and app: !!f ts. list-all P ts ==> P (App f ts)
shows term-induct2: P t
  and list-all P ts
apply (induct t and ts)
  apply (rule var)
  apply (rule app)
  apply assumption
  apply simp-all
done

end

```

6 Arithmetic and boolean expressions

theory *ABexp* **imports** *Main* **begin**

```

datatype 'a aexp =
  IF 'a bexp 'a aexp 'a aexp
| Sum 'a aexp 'a aexp
| Diff 'a aexp 'a aexp
| Var 'a
| Num nat
and 'a bexp =
  Less 'a aexp 'a aexp
| And 'a bexp 'a bexp
| Neg 'a bexp

```

Evaluation of arithmetic and boolean expressions

consts

```

evala :: ('a ==> nat) ==> 'a aexp ==> nat
evalb :: ('a ==> nat) ==> 'a bexp ==> bool

```

primrec

```

evala env (IF b a1 a2) = (if evalb env b then evala env a1 else evala env a2)
evala env (Sum a1 a2) = evala env a1 + evala env a2
evala env (Diff a1 a2) = evala env a1 - evala env a2
evala env (Var v) = env v
evala env (Num n) = n

```

```

evalb env (Less a1 a2) = (evala env a1 < evala env a2)
evalb env (And b1 b2) = (evalb env b1 ∧ evalb env b2)
evalb env (Neg b) = (¬ evalb env b)

```

Substitution on arithmetic and boolean expressions

consts

$subst :: ('a \Rightarrow 'b \ aexp) \Rightarrow 'a \ aexp \Rightarrow 'b \ aexp$
 $substb :: ('a \Rightarrow 'b \ aexp) \Rightarrow 'a \ bexp \Rightarrow 'b \ bexp$

primrec

$subst \ f \ (IF \ b \ a1 \ a2) = IF \ (substb \ f \ b) \ (subst \ f \ a1) \ (subst \ f \ a2)$
 $subst \ f \ (Sum \ a1 \ a2) = Sum \ (subst \ f \ a1) \ (subst \ f \ a2)$
 $subst \ f \ (Diff \ a1 \ a2) = Diff \ (subst \ f \ a1) \ (subst \ f \ a2)$
 $subst \ f \ (Var \ v) = f \ v$
 $subst \ f \ (Num \ n) = Num \ n$

$substb \ f \ (Less \ a1 \ a2) = Less \ (subst \ f \ a1) \ (subst \ f \ a2)$
 $substb \ f \ (And \ b1 \ b2) = And \ (substb \ f \ b1) \ (substb \ f \ b2)$
 $substb \ f \ (Neg \ b) = Neg \ (substb \ f \ b)$

lemma *subst1-aexp*:

$evala \ env \ (subst \ (Var \ (v := a')) \ a) = evala \ (env \ (v := evala \ env \ a')) \ a$

and *subst1-bexp*:

$evalb \ env \ (substb \ (Var \ (v := a')) \ b) = evalb \ (env \ (v := evala \ env \ a')) \ b$
 — one variable

by (*induct a and b simp-all*)

lemma *subst-all-aexp*:

$evala \ env \ (subst \ s \ a) = evala \ (\lambda x. \ evala \ env \ (s \ x)) \ a$

and *subst-all-bexp*:

$evalb \ env \ (substb \ s \ b) = evalb \ (\lambda x. \ evala \ env \ (s \ x)) \ b$

by (*induct a and b auto*)

end

7 Infinitely branching trees

theory *Tree* **imports** *Main* **begin**

datatype $'a \ tree =$

$Atom \ 'a$
 $| \ Branch \ nat \ \Rightarrow \ 'a \ tree$

consts

$map-tree :: ('a \ \Rightarrow \ 'b) \ \Rightarrow \ 'a \ tree \ \Rightarrow \ 'b \ tree$

primrec

$map-tree \ f \ (Atom \ a) = Atom \ (f \ a)$
 $map-tree \ f \ (Branch \ ts) = Branch \ (\lambda x. \ map-tree \ f \ (ts \ x))$

lemma *tree-map-compose*: $map-tree \ g \ (map-tree \ f \ t) = map-tree \ (g \circ f) \ t$

by (*induct t simp-all*)

consts

$exists-tree :: ('a \ \Rightarrow \ bool) \ \Rightarrow \ 'a \ tree \ \Rightarrow \ bool$

primrec

exists-tree P (*Atom* a) = P a
exists-tree P (*Branch* ts) = $(\exists x. \text{exists-tree } P (ts\ x))$

lemma *exists-map*:

$(!!x. P\ x ==> Q\ (f\ x)) ==>$
exists-tree $P\ ts ==> \text{exists-tree } Q\ (\text{map-tree } f\ ts)$
by (*induct* ts) *auto*

7.1 The Brouwer ordinals, as in ZF/Induct/Brouwer.thy.

datatype *brouwer* = *Zero* | *Succ* *brouwer* | *Lim* *nat* => *brouwer*

Addition of ordinals

consts

add :: [*brouwer*,*brouwer*] => *brouwer*

primrec

add i *Zero* = i
add i (*Succ* j) = *Succ* (*add* i j)
add i (*Lim* f) = *Lim* ($\%n. \text{add } i\ (f\ n)$)

lemma *add-assoc*: *add* (*add* i j) k = *add* i (*add* j k)

by (*induct* k) *auto*

Multiplication of ordinals

consts

mult :: [*brouwer*,*brouwer*] => *brouwer*

primrec

mult i *Zero* = *Zero*
mult i (*Succ* j) = *add* (*mult* i j) i
mult i (*Lim* f) = *Lim* ($\%n. \text{mult } i\ (f\ n)$)

lemma *add-mult-distrib*: *mult* i (*add* j k) = *add* (*mult* i j) (*mult* i k)

by (*induct* k) (*auto simp add: add-assoc*)

lemma *mult-assoc*: *mult* (*mult* i j) k = *mult* i (*mult* j k)

by (*induct* k) (*auto simp add: add-mult-distrib*)

We could probably instantiate some axiomatic type classes and use the standard infix operators.

7.2 A WF Ordering for The Brouwer ordinals (Michael Comp-ton)

To define *recdef* style functions we need an ordering on the Brouwer ordinals. Start with a predecessor relation and form its transitive closure.

definition

brouwer-pred :: (*brouwer* * *brouwer*) *set* **where**

$brouwer\text{-}pred = (\bigcup i. \{(m,n). n = Succ\ m \vee (EX\ f. n = Lim\ f \ \&\ m = f\ i)\})$

definition

$brouwer\text{-}order :: (brouwer * brouwer) \text{ set } \mathbf{where}$
 $brouwer\text{-}order = brouwer\text{-}pred^+ +$

lemma $wf\text{-}brouwer\text{-}pred$: $wf\ brouwer\text{-}pred$

by($unfold\ wf\text{-}def\ brouwer\text{-}pred\text{-}def, \text{clarify}, induct\text{-}tac\ x, blast+$)

lemma $wf\text{-}brouwer\text{-}order$: $wf\ brouwer\text{-}order$

by($unfold\ brouwer\text{-}order\text{-}def, rule\ wf\text{-}transcl[OF\ wf\text{-}brouwer\text{-}pred]$)

lemma $[simp]$: $(j, Succ\ j) : brouwer\text{-}order$

by($auto\ simp\ add: brouwer\text{-}order\text{-}def\ brouwer\text{-}pred\text{-}def$)

lemma $[simp]$: $(f\ n, Lim\ f) : brouwer\text{-}order$

by($auto\ simp\ add: brouwer\text{-}order\text{-}def\ brouwer\text{-}pred\text{-}def$)

Example of a recdef

consts

$add2 :: (brouwer * brouwer) \Rightarrow brouwer$

recdef $add2\ inv\text{-}image\ brouwer\text{-}order\ (\lambda\ (x,y). y)$

$add2\ (i, Zero) = i$

$add2\ (i, (Succ\ j)) = Succ\ (add2\ (i, j))$

$add2\ (i, (Lim\ f)) = Lim\ (\lambda\ n. add2\ (i, (f\ n)))$

(**hints** $recdef\text{-}wf: wf\text{-}brouwer\text{-}order$)

lemma $add2\text{-}assoc$: $add2\ (add2\ (i, j), k) = add2\ (i, add2\ (j, k))$

by ($induct\ k\ auto$)

end

8 Ordinals

theory *Ordinals* **imports** *Main* **begin**

Some basic definitions of ordinal numbers. Draws an Agda development (in Martin-Löf type theory) by Peter Hancock (see <http://www.dcs.ed.ac.uk/home/pgh/chat.html>).

datatype *ordinal* =

Zero

| *Succ ordinal*

| *Limit nat => ordinal*

consts

$pred :: ordinal \Rightarrow nat \Rightarrow ordinal\ option$

primrec

```

    pred Zero n = None
    pred (Succ a) n = Some a
    pred (Limit f) n = Some (f n)

consts
    iter :: ('a => 'a) => nat => ('a => 'a)
primrec
    iter f 0 = id
    iter f (Suc n) = f ∘ (iter f n)

definition
    OpLim :: (nat => (ordinal => ordinal)) => (ordinal => ordinal) where
    OpLim F a = Limit (λn. F n a)

definition
    OpItw :: (ordinal => ordinal) => (ordinal => ordinal)    (⋒) where
    ⋒f = OpLim (iter f)

consts
    cantor :: ordinal => ordinal => ordinal
primrec
    cantor a Zero = Succ a
    cantor a (Succ b) = ⋒(λx. cantor x b) a
    cantor a (Limit f) = Limit (λn. cantor a (f n))

consts
    Nbla :: (ordinal => ordinal) => (ordinal => ordinal)    (∇)
primrec
    ∇f Zero = f Zero
    ∇f (Succ a) = f (Succ (∇f a))
    ∇f (Limit h) = Limit (λn. ∇f (h n))

definition
    deriv :: (ordinal => ordinal) => (ordinal => ordinal) where
    deriv f = ∇(⋒f)

consts
    veblen :: ordinal => ordinal => ordinal
primrec
    veblen Zero = ∇(OpLim (iter (cantor Zero)))
    veblen (Succ a) = ∇(OpLim (iter (veblen a)))
    veblen (Limit f) = ∇(OpLim (λn. veblen (f n)))

definition veb a = veblen a Zero
definition ε0 = veb Zero
definition Γ0 = Limit (λn. iter veb n Zero)

end

```

9 Sigma algebras

theory *Sigma-Algebra* **imports** *Main* **begin**

This is just a tiny example demonstrating the use of inductive definitions in classical mathematics. We define the least σ -algebra over a given set of sets.

inductive-set

$\sigma\text{-algebra} :: 'a \text{ set set} \Rightarrow 'a \text{ set set}$

for $A :: 'a \text{ set set}$

where

basic: $a \in A \Rightarrow a \in \sigma\text{-algebra } A$

| *UNIV*: $UNIV \in \sigma\text{-algebra } A$

| *complement*: $a \in \sigma\text{-algebra } A \Rightarrow -a \in \sigma\text{-algebra } A$

| *Union*: $(!!i::nat. a\ i \in \sigma\text{-algebra } A) \Rightarrow (\bigcup i. a\ i) \in \sigma\text{-algebra } A$

The following basic facts are consequences of the closure properties of any σ -algebra, merely using the introduction rules, but no induction nor cases.

theorem *sigma-algebra-empty*: $\{\} \in \sigma\text{-algebra } A$

proof –

have $UNIV \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.UNIV}$)

hence $-UNIV \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.complement}$)

also have $-UNIV = \{\}$ **by** *simp*

finally show *?thesis* .

qed

theorem *sigma-algebra-Inter*:

$(!!i::nat. a\ i \in \sigma\text{-algebra } A) \Rightarrow (\bigcap i. a\ i) \in \sigma\text{-algebra } A$

proof –

assume $!!i::nat. a\ i \in \sigma\text{-algebra } A$

hence $!!i::nat. -(a\ i) \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.complement}$)

hence $(\bigcup i. -(a\ i)) \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.Union}$)

hence $-(\bigcup i. -(a\ i)) \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.complement}$)

also have $-(\bigcup i. -(a\ i)) = (\bigcap i. a\ i)$ **by** *simp*

finally show *?thesis* .

qed

end

10 Combinatory Logic example: the Church-Rosser Theorem

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

Curiously, combinators do not include free variables.

Example taken from [?].

HOL system proofs may be found in the HOL distribution at .../contrib/rule-induction/cl.ml

10.1 Definitions

Datatype definition of combinators S and K .

```
datatype comb = K
      | S
      | Ap comb comb (infixl ## 90)
```

```
notation (xsymbols)
  Ap (infixl • 90)
```

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

```
inductive-set
  contract :: (comb*comb) set
  and contract-rel1 :: [comb,comb] => bool (infixl  $-1->$  50)
  where
     $x -1-> y == (x,y) \in \text{contract}$ 
    | K:  $K \## x \## y -1-> x$ 
    | S:  $S \## x \## y \## z -1-> (x \## z) \## (y \## z)$ 
    | Ap1:  $x -1-> y ==> x \## z -1-> y \## z$ 
    | Ap2:  $x -1-> y ==> z \## x -1-> z \## y$ 
```

```
abbreviation
  contract-rel :: [comb,comb] => bool (infixl  $---->$  50) where
     $x ----> y == (x,y) \in \text{contract}^*$ 
```

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

```
inductive-set
  parcontract :: (comb*comb) set
  and parcontract-rel1 :: [comb,comb] => bool (infixl  $=1=>$  50)
  where
     $x =1=> y == (x,y) \in \text{parcontract}$ 
    | refl:  $x =1=> x$ 
    | K:  $K \## x \## y =1=> x$ 
    | S:  $S \## x \## y \## z =1=> (x \## z) \## (y \## z)$ 
    | Ap:  $[| x =1=> y; z =1=> w |] ==> x \## z =1=> y \## w$ 
```

```
abbreviation
  parcontract-rel :: [comb,comb] => bool (infixl  $===>$  50) where
     $x ===> y == (x,y) \in \text{parcontract}^*$ 
```

Misc definitions.

```
definition
  I :: comb where
     $I = S \## K \## K$ 
```

```
definition
```

$\text{diamond} :: ('a * 'a)\text{set} \Rightarrow \text{bool}$ **where**
 — confluence; Lambda/Commutation treats this more abstractly
 $\text{diamond}(r) = (\forall x y. (x,y) \in r \dashrightarrow$
 $\quad (\forall y'. (x,y') \in r \dashrightarrow$
 $\quad (\exists z. (y,z) \in r \ \& \ (y',z) \in r)))$

10.2 Reflexive/Transitive closure preserves Church-Rosser property

So does the Transitive closure, with a similar proof

Strip lemma. The induction hypothesis covers all but the last diamond of the strip.

lemma *diamond-strip-lemmaE* [rule-format]:
 $[[\text{diamond}(r); (x,y) \in r^*]] \Rightarrow$
 $\quad \forall y'. (x,y') \in r \dashrightarrow (\exists z. (y',z) \in r^* \ \& \ (y,z) \in r)$
apply (*unfold diamond-def*)
apply (*erule rtrancl-induct*)
apply (*meson rtrancl-refl*)
apply (*meson rtrancl-trans r-into-rtrancl*)
done

lemma *diamond-rtrancl*: $\text{diamond}(r) \Rightarrow \text{diamond}(r^*)$
apply (*simp (no-asm-simp) add: diamond-def*)
apply (*rule impI [THEN allI, THEN allI]*)
apply (*erule rtrancl-induct, blast*)
apply (*meson rtrancl-trans r-into-rtrancl diamond-strip-lemmaE*)
done

10.3 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases

$K\text{-contractE}$ [elim!]: $K -1-> r$
and $S\text{-contractE}$ [elim!]: $S -1-> r$
and $Ap\text{-contractE}$ [elim!]: $p\#\#q -1-> r$

declare *contract.K* [intro!] *contract.S* [intro!]
declare *contract.Ap1* [intro] *contract.Ap2* [intro]

lemma *I-contract-E* [elim!]: $I -1-> z \Rightarrow P$
by (*unfold I-def, blast*)

lemma *K1-contractD* [elim!]: $K\#\#x -1-> z \Rightarrow (\exists x'. z = K\#\#x' \ \& \ x -1-> x')$
by *blast*

lemma *Ap-reduce1* [intro]: $x \dashrightarrow y \Rightarrow x\#\#z \dashrightarrow y\#\#z$


```

apply (erule rtrancl-induct)
apply (blast intro: rtrancl-trans)+
done

```

```

lemma Ap-reduce2 [intro]:  $x \dashrightarrow y \implies z \# \# x \dashrightarrow z \# \# y$ 
apply (erule rtrancl-induct)
apply (blast intro: rtrancl-trans)+
done

```

```

lemma KIII-contract1:  $K \# \# I \# \# (I \# \# I) - 1 -> I$ 
by (rule contract.K)

```

```

lemma KIII-contract2:  $K \# \# I \# \# (I \# \# I) - 1 -> K \# \# I \# \# ((K \# \# I) \# \# (K \# \# I))$ 
by (unfold I-def, blast)

```

```

lemma KIII-contract3:  $K \# \# I \# \# ((K \# \# I) \# \# (K \# \# I)) - 1 -> I$ 
by blast

```

```

lemma not-diamond-contract:  $\sim \text{diamond}(\text{contract})$ 
apply (unfold diamond-def)
apply (best intro: KIII-contract1 KIII-contract2 KIII-contract3)
done

```

10.4 Results about Parallel Contraction

Derive a case for each combinator constructor.

inductive-cases

```

    K-parcontractE [elim!]:  $K = 1 \Rightarrow r$ 
    and S-parcontractE [elim!]:  $S = 1 \Rightarrow r$ 
    and Ap-parcontractE [elim!]:  $p \# \# q = 1 \Rightarrow r$ 

```

```

declare parcontract.intros [intro]

```

10.5 Basic properties of parallel contraction

```

lemma K1-parcontractD [dest!]:  $K \# \# x = 1 \Rightarrow z \implies (\exists x'. z = K \# \# x' \ \& \ x = 1 \Rightarrow x')$ 
by blast

```

```

lemma S1-parcontractD [dest!]:  $S \# \# x = 1 \Rightarrow z \implies (\exists x'. z = S \# \# x' \ \& \ x = 1 \Rightarrow x')$ 
by blast

```

```

lemma S2-parcontractD [dest!]:
     $S \# \# x \# \# y = 1 \Rightarrow z \implies (\exists x' y'. z = S \# \# x' \# \# y' \ \& \ x = 1 \Rightarrow x' \ \& \ y = 1 \Rightarrow y')$ 
by blast

```

The rules above are not essential but make proofs much faster

Church-Rosser property for parallel contraction

```
lemma diamond-parcontract: diamond parcontract
apply (unfold diamond-def)
apply (rule impI [THEN allI, THEN allI])
apply (erule parcontract.induct, fast+)
done
```

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

```
lemma contract-subset-parcontract: contract <= parcontract
apply (rule subsetI)
apply (simp only: split-tupled-all)
apply (erule contract.induct, blast+)
done
```

Reductions: simply throw together reflexivity, transitivity and the one-step reductions

```
declare r-into-rtrancl [intro] rtrancl-trans [intro]
```

```
lemma reduce-I: I##x <-> x
by (unfold I-def, blast)
```

```
lemma parcontract-subset-reduce: parcontract <= contract^*
apply (rule subsetI)
apply (simp only: split-tupled-all)
apply (erule parcontract.induct, blast+)
done
```

```
lemma reduce-eq-parreduce: contract^* = parcontract^*
by (rule equalityI contract-subset-parcontract [THEN rtrancl-mono]
      parcontract-subset-reduce [THEN rtrancl-subset-rtrancl])+
```

```
lemma diamond-reduce: diamond(contract^*)
by (simp add: reduce-eq-parreduce diamond-rtrancl diamond-parcontract)
```

end

11 Meta-theory of propositional logic

```
theory PropLog imports Main begin
```

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

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

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

11.1 The datatype of propositions

```
datatype 'a pl =
  false |
  var 'a (#- [1000]) |
  imp 'a pl 'a pl (infixr -> 90)
```

11.2 The proof system

```
inductive
  thms :: ['a pl set, 'a pl] => bool (infixl |- 50)
for H :: 'a pl set
where
  H [intro]: p ∈ H ==> H |- p
  | K:      H |- p -> q -> p
  | S:      H |- (p -> q -> r) -> (p -> q) -> p -> r
  | DN:     H |- ((p -> false) -> false) -> p
  | MP:     [| H |- p -> q; H |- p |] ==> H |- q
```

11.3 The semantics

11.3.1 Semantics of propositional logic.

```
consts
  eval :: ['a set, 'a pl] => bool    (-[[-]] [100,0] 100)

primrec
  tt[[false]] = False
  tt[[#v]]     = (v ∈ tt)
  eval-imp: tt[[p -> q]] = (tt[[p]] --> tt[[q]])
```

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

```
consts
  hyps :: ['a pl, 'a set] => 'a pl set
```

```
primrec
  hyps false tt = {}
  hyps (#v) tt = {if v ∈ tt then #v else #v -> false}
  hyps (p -> q) tt = hyps p tt Un hyps q tt
```

11.3.2 Logical consequence

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

```
definition
  sat :: ['a pl set, 'a pl] => bool (infixl |= 50) where
    H |= p = (∀ tt. (∀ q ∈ H. tt[[q]]) --> tt[[p]])
```

11.4 Proof theory of propositional logic

lemma *thms-mono*: $G \leq H \implies \text{thms}(G) \leq \text{thms}(H)$
apply (*rule predicate1I*)
apply (*erule thms.induct*)
apply (*auto intro: thms.intros*)
done

lemma *thms-I*: $H \mid\!-\! p \multimap p$
— Called *I* for Identity Combinator, not for Introduction.
by (*best intro: thms.K thms.S thms.MP*)

11.4.1 Weakening, left and right

lemma *weaken-left*: $[G \subseteq H; G \mid\!-\! p] \implies H \mid\!-\! p$
— Order of premises is convenient with *THEN*
by (*erule thms-mono [THEN predicate1D]*)

lemmas *weaken-left-insert* = *subset-insertI [THEN weaken-left]*

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

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

lemma *weaken-right*: $H \mid\!-\! q \implies H \mid\!-\! p \multimap q$
by (*fast intro: thms.K thms.MP*)

11.4.2 The deduction theorem

theorem *deduction*: $\text{insert } p \text{ } H \mid\!-\! q \implies H \mid\!-\! p \multimap q$
apply (*induct set: thms*)
apply (*fast intro: thms-I thms.H thms.K thms.S thms.DN*
 thms.S [THEN thms.MP, THEN thms.MP] weaken-right)
done

11.4.3 The cut rule

lemmas *cut* = *deduction [THEN thms.MP]*

lemmas *thms-falseE* = *weaken-right [THEN thms.DN [THEN thms.MP]]*

lemmas *thms-notE* = *thms.MP [THEN thms-falseE, standard]*

11.4.4 Soundness of the rules wrt truth-table semantics

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

11.5 Completeness

11.5.1 Towards the completeness proof

```
lemma false-imp:  $H \vdash p \rightarrow \text{false} \implies H \vdash p \rightarrow q$   
  apply (rule deduction)  
  apply (erule weaken-left-insert [THEN thms-notE])  
  apply blast  
done
```

```
lemma imp-false:  
  [ $H \vdash p$ ;  $H \vdash q \rightarrow \text{false}$ ]  $\implies H \vdash (p \rightarrow q) \rightarrow \text{false}$   
  apply (rule deduction)  
  apply (blast intro: weaken-left-insert thms.MP thms.H)  
done
```

```
lemma hyps-thms-if:  $\text{hyps } p \text{ tt} \vdash (\text{if } \text{tt}[[p]] \text{ then } p \text{ else } p \rightarrow \text{false})$   
  — Typical example of strengthening the induction statement.  
  apply simp  
  apply (induct p)  
  apply (simp-all add: thms-I thms.H)  
  apply (blast intro: weaken-left-Un1 weaken-left-Un2 weaken-right  
    imp-false false-imp)  
done
```

```
lemma sat-thms-p:  $\{\} \models p \implies \text{hyps } p \text{ tt} \vdash p$   
  — Key lemma for completeness; yields a set of assumptions satisfying  $p$   
  apply (unfold sat-def)  
  apply (drule spec, erule mp [THEN if-P, THEN subst],  
    rule-tac [2] hyps-thms-if, simp)  
done
```

For proving certain theorems in our new propositional logic.

```
declare deduction [intro!]  
declare thms.H [THEN thms.MP, intro]
```

The excluded middle in the form of an elimination rule.

```
lemma thms-excluded-middle:  $H \vdash (p \rightarrow q) \rightarrow ((p \rightarrow \text{false}) \rightarrow q) \rightarrow q$   
  apply (rule deduction [THEN deduction])  
  apply (rule thms.DN [THEN thms.MP], best)  
done
```

```
lemma thms-excluded-middle-rule:  
  [ $\text{insert } p \text{ } H \vdash q$ ;  $\text{insert } (p \rightarrow \text{false}) \text{ } H \vdash q$ ]  $\implies H \vdash q$   
  — Hard to prove directly because it requires cuts  
  by (rule thms-excluded-middle [THEN thms.MP, THEN thms.MP], auto)
```

11.6 Completeness – lemmas for reducing the set of assumptions

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

lemma *hyps-Diff*: $\text{hyps } p \ (t - \{v\}) \leq \text{insert } (\#v \rightarrow \text{false}) \ ((\text{hyps } p \ t) - \{\#v\})$
by *(induct p) auto*

For the case $\text{hyps } p \ t - \text{insert } (\#v \rightarrow \text{Fls}) \ Y \mid - p$ we also have $\text{hyps } p \ t - \{\#v \rightarrow \text{Fls}\} \subseteq \text{hyps } p \ (\text{insert } v \ t)$.

lemma *hyps-insert*: $\text{hyps } p \ (\text{insert } v \ t) \leq \text{insert } (\#v) \ (\text{hyps } p \ t - \{\#v \rightarrow \text{false}\})$
by *(induct p) auto*

Two lemmas for use with *weaken-left*

lemma *insert-Diff-same*: $B - C \leq \text{insert } a \ (B - \text{insert } a \ C)$
by *fast*

lemma *insert-Diff-subset2*: $\text{insert } a \ (B - \{c\}) - D \leq \text{insert } a \ (B - \text{insert } c \ D)$
by *fast*

The set $\text{hyps } p \ t$ is finite, and elements have the form $\#v$ or $\#v \rightarrow \text{Fls}$.

lemma *hyps-finite*: $\text{finite}(\text{hyps } p \ t)$
by *(induct p) auto*

lemma *hyps-subset*: $\text{hyps } p \ t \leq (\text{UN } v. \{\#v, \#v \rightarrow \text{false}\})$
by *(induct p) auto*

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

11.6.1 Completeness theorem

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

lemma *completeness-0-lemma*:

$\{\} \mid = p \implies \forall t. \text{hyps } p \ t - \text{hyps } p \ t0 \mid - p$
apply (*rule* *hyps-subset* [*THEN* *hyps-finite* [*THEN* *finite-subset-induct*]])
apply (*simp* *add: sat-thms-p, safe*)

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

apply (*iprover* *intro: thms-excluded-middle-rule*
insert-Diff-same [*THEN* *weaken-left*]
insert-Diff-subset2 [*THEN* *weaken-left*]
hyps-Diff [*THEN* *Diff-weaken-left*])

Case $\text{hyps } p \ t - \text{insert } (\#v \rightarrow \text{false}, Y) \mid - p$

apply (*iprover* *intro: thms-excluded-middle-rule*
insert-Diff-same [*THEN* *weaken-left*])

```

      insert-Diff-subset2 [THEN weaken-left]
      hyps-insert [THEN Diff-weaken-left])
done

The base case for completeness

lemma completeness-0:  $\{\} \models p \implies \{\} \vdash p$ 
apply (rule Diff-cancel [THEN subst])
apply (erule completeness-0-lemma [THEN spec])
done

A semantic analogue of the Deduction Theorem

lemma sat-imp:  $\text{insert } p \ H \models q \implies H \models p \multimap q$ 
by (unfold sat-def, auto)

theorem completeness:  $\text{finite } H \implies H \models p \implies H \vdash p$ 
apply (induct arbitrary: p rule: finite-induct)
  apply (blast intro: completeness-0)
apply (iprover intro: sat-imp thms.H insertI1 weaken-left-insert [THEN thms.MP])
done

theorem syntax-iff-semantics:  $\text{finite } H \implies (H \vdash p) = (H \models p)$ 
by (blast intro: soundness completeness)

end

theory Sexp imports Main begin

types
  'a item = 'a Datatype.item
abbreviation Leaf == Datatype.Leaf
abbreviation Numb == Datatype.Numb

inductive-set
  sexp      :: 'a item set
  where
    LeafI: Leaf(a)  $\in$  sexp
    | NumbI: Numb(i)  $\in$  sexp
    | SconsI: [ $M \in$  sexp;  $N \in$  sexp]  $\implies$  Scons M N  $\in$  sexp

definition
  sexp-case :: [ $a \Rightarrow b$ ,  $\text{nat} \Rightarrow b$ , [ $'a \text{ item}$ ,  $'a \text{ item}$ ]  $\Rightarrow b$ ,
     $'a \text{ item}$ ]  $\Rightarrow b$  where
    sexp-case c d e M = (THE z. (EX x. M=Leaf(x) & z=c(x))
      | (EX k. M=Numb(k) & z=d(k))
      | (EX N1 N2. M = Scons N1 N2 & z=e N1 N2))

definition

```

pred-sexp :: ('a item * 'a item)set **where**
pred-sexp = ($\bigcup M \in \text{sexp}. \bigcup N \in \text{sexp}. \{(M, \text{Scons } M \ N), (N, \text{Scons } M \ N)\}$)

definition

sexp-rec :: ['a item, 'a=>'b, nat=>'b,
 ['a item, 'a item, 'b, 'b]=>'b] => 'b **where**
sexp-rec *M c d e* = wfrec *pred-sexp*
 (%g. *sexp-case* *c d* (%*N1 N2*. *e N1 N2 (g N1) (g N2)*))) *M*

lemma *sexp-case-Leaf* [*simp*]: *sexp-case c d e (Leaf a) = c(a)*
by (*simp add: sexp-case-def, blast*)

lemma *sexp-case-Numb* [*simp*]: *sexp-case c d e (Numb k) = d(k)*
by (*simp add: sexp-case-def, blast*)

lemma *sexp-case-Scons* [*simp*]: *sexp-case c d e (Scons M N) = e M N*
by (*simp add: sexp-case-def*)

lemma *sexp-In0I*: *M* ∈ *sexp* ==> *In0(M)* ∈ *sexp*
apply (*simp add: In0-def*)
apply (*erule sexp.NumbI [THEN sexp.SconsI]*)
done

lemma *sexp-In1I*: *M* ∈ *sexp* ==> *In1(M)* ∈ *sexp*
apply (*simp add: In1-def*)
apply (*erule sexp.NumbI [THEN sexp.SconsI]*)
done

declare *sexp.intros* [*intro, simp*]

lemma *range-Leaf-subset-sexp*: *range(Leaf) <= sexp*
by *blast*

lemma *Scons-D*: *Scons M N* ∈ *sexp* ==> *M* ∈ *sexp* & *N* ∈ *sexp*
by (*induct S == Scons M N set: sexp*) *auto*

lemma *pred-sexp-subset-Sigma*: *pred-sexp <= sexp <*> sexp*
by (*simp add: pred-sexp-def, blast*)

lemmas *trancI-pred-sexpD1* =
 pred-sexp-subset-Sigma
 [*THEN trancI-subset-Sigma, THEN subsetD, THEN SigmaD1*]
and *trancI-pred-sexpD2* =
 pred-sexp-subset-Sigma
 [*THEN trancI-subset-Sigma, THEN subsetD, THEN SigmaD2*]

lemma *pred-sexpI1*:
 [*M* ∈ *sexp*; *N* ∈ *sexp*] ==> (*M*, *Scons M N*) ∈ *pred-sexp*
by (*simp add: pred-sexp-def, blast*)

lemma *pred-sexpI2*:
 [*M* ∈ *sexp*; *N* ∈ *sexp*] ==> (*N*, *Scons M N*) ∈ *pred-sexp*
by (*simp add: pred-sexp-def, blast*)

lemmas *pred-sexp-t1* [*simp*] = *pred-sexpI1* [*THEN r-into-trancI*]
and *pred-sexp-t2* [*simp*] = *pred-sexpI2* [*THEN r-into-trancI*]

lemmas *pred-sexp-trans1* [*simp*] = *trans-trancI* [*THEN transD, OF - pred-sexp-t1*]
and *pred-sexp-trans2* [*simp*] = *trans-trancI* [*THEN transD, OF - pred-sexp-t2*]

declare *cut-apply* [*simp*]

lemma *pred-sexpE*:
 [*p* ∈ *pred-sexp*;
 !!*M N*. [*p* = (*M*, *Scons M N*); *M* ∈ *sexp*; *N* ∈ *sexp*] ==> *R*;
 !!*M N*. [*p* = (*N*, *Scons M N*); *M* ∈ *sexp*; *N* ∈ *sexp*] ==> *R*
] ==> *R*
by (*simp add: pred-sexp-def, blast*)

lemma *wf-pred-sexp*: *wf(pred-sexp)*
apply (*rule pred-sexp-subset-Sigma* [*THEN wfI*])
apply (*erule sexp.induct*)
apply (*blast elim!: pred-sexpE*) +
done

lemma *sexp-rec-unfold-lemma*:
 (%*M*. *sexp-rec M c d e*) ==
 wfrec pred-sexp (%*g*. *sexp-case c d* (%*N1 N2*. *e N1 N2 (g N1) (g N2)*)))
by (*simp add: sexp-rec-def*)

lemmas *sexp-rec-unfold* = *def-wfrec* [*OF sexp-rec-unfold-lemma wf-pred-sexp*]

```

lemma sexp-rec-Leaf: sexp-rec (Leaf a) c d h = c(a)
apply (subst sexp-rec-unfold)
apply (rule sexp-case-Leaf)
done

lemma sexp-rec-Numb: sexp-rec (Numb k) c d h = d(k)
apply (subst sexp-rec-unfold)
apply (rule sexp-case-Numb)
done

lemma sexp-rec-Scons: [M ∈ sexp; N ∈ sexp] ==>
  sexp-rec (Scons M N) c d h = h M N (sexp-rec M c d h) (sexp-rec N c d h)
apply (rule sexp-rec-unfold [THEN trans])
apply (simp add: pred-sexpI1 pred-sexpI2)
done

end

```

```

theory SList
imports Sexp
begin

```

```

definition

```

NIL :: 'a item **where**
NIL = *In0*(*Numb*(0))

definition

CONS :: ['a item, 'a item] => 'a item **where**
CONS *M N* = *In1*(*Scons* *M N*)

inductive-set

list :: 'a item set => 'a item set
for *A* :: 'a item set
where
NIL-I: *NIL*: *list* *A*
| *CONS-I*: [| *a*: *A*; *M*: *list* *A* |] ==> *CONS* *a M* : *list* *A*

typedef (*List*)

'a list = *list*(*range Leaf*) :: 'a item set
by (*blast intro*: *list.NIL-I*)

abbreviation *Case* == *Datatype.Case*

abbreviation *Split* == *Datatype.Split*

definition

List-case :: ['b, ['a item, 'a item]=>'b, 'a item] => 'b **where**
List-case *c d* = *Case*(%*x. c*)(*Split*(*d*))

definition

List-rec :: ['a item, 'b, ['a item, 'a item, 'b]=>'b] => 'b **where**
List-rec *M c d* = *wfrec* (*pred-sexp* ^+)
(%*g. List-case* *c* (%*x y. d x y* (*g y*))) *M*

definition

Nil :: 'a list ([]) **where**
Nil = *Abs-List*(*NIL*)

definition

Cons :: ['a, 'a list] => 'a list (infixr # 65) **where**
x#xs = *Abs-List*(*CONS* (*Leaf* *x*)(*Rep-List* *xs*))

definition

$list-rec :: ['a\ list, 'b, ['a, 'a\ list, 'b] => 'b] => 'b\ \mathbf{where}$
 $list-rec\ l\ c\ d =$
 $List-rec(Rep-List\ l)\ c\ (\%x\ y\ r.\ d(inv\ Leaf\ x)(Abs-List\ y)\ r)$

definition

$list-case :: ['b, ['a, 'a\ list] => 'b, 'a\ list] => 'b\ \mathbf{where}$
 $list-case\ a\ f\ xs = list-rec\ xs\ a\ (\%x\ xs\ r.\ f\ x\ xs)$

translations

$[x, xs] == x\#[xs]$
 $[x] == x\#[]$

$case\ xs\ of\ [] => a\ |\ y\#ys => b == CONST\ list-case(a, \%y\ ys.\ b, xs)$

definition

$Rep-map :: ('b => 'a\ item) => ('b\ list => 'a\ item)\ \mathbf{where}$
 $Rep-map\ f\ xs = list-rec\ xs\ NIL(\%x\ l\ r.\ CONS(f\ x)\ r)$

definition

$Abs-map :: ('a\ item => 'b) => 'a\ item => 'b\ list\ \mathbf{where}$
 $Abs-map\ g\ M = List-rec\ M\ Nil\ (\%N\ L\ r.\ g(N)\#r)$

definition

$null :: 'a\ list => bool\ \mathbf{where}$
 $null\ xs = list-rec\ xs\ True\ (\%x\ xs\ r.\ False)$

definition

$hd :: 'a\ list => 'a\ \mathbf{where}$
 $hd\ xs = list-rec\ xs\ (@x.\ True)\ (\%x\ xs\ r.\ x)$

definition

$tl :: 'a\ list => 'a\ list\ \mathbf{where}$
 $tl\ xs = list-rec\ xs\ (@xs.\ True)\ (\%x\ xs\ r.\ xs)$

definition

tll :: 'a list => 'a list **where**
tll xs = list-rec xs [] (%x xs r. xs)

definition

member :: ['a, 'a list] => bool (infixl mem 55) **where**
x mem xs = list-rec xs False (%y ys r. if y=x then True else r)

definition

list-all :: ('a => bool) => ('a list => bool) **where**
list-all P xs = list-rec xs True (%x l r. P(x) & r)

definition

map :: ('a=>'b) => ('a list => 'b list) **where**
map f xs = list-rec xs [] (%x l r. f(x)#r)

definition

append :: ['a list, 'a list] => 'a list (infixr @ 65) **where**
xs@ys = list-rec xs ys (%x l r. x#r)

definition

filter :: ['a => bool, 'a list] => 'a list **where**
filter P xs = list-rec xs [] (%x xs r. if P(x) then x#r else r)

definition

foldl :: [['b,'a] => 'b, 'b, 'a list] => 'b **where**
foldl f a xs = list-rec xs (%a. a)(%x xs r.%a. r(f a x))(a)

definition

foldr :: [['a,'b] => 'b, 'b, 'a list] => 'b **where**
foldr f a xs = list-rec xs a (%x xs r. (f x r))

definition

length :: 'a list => nat **where**
length xs = list-rec xs 0 (%x xs r. Suc r)

definition

drop :: ['a list, nat] => 'a list **where**
drop t n = (nat-rec (%x. x)(%m r xs. r(tll xs)))(n)(t)

definition

copy :: ['a, nat] => 'a list **where**
copy t = nat-rec [] (%m xs. t # xs)

definition

flat :: 'a list list => 'a list **where**
flat = foldr (op @) []

definition

nth :: [nat, 'a list] => 'a **where**

$nth = nat-rec\ hd\ (\%m\ r\ xs.\ r(tl\ xs))$

definition

$rev :: 'a\ list \Rightarrow 'a\ list$ **where**
 $rev\ xs = list-rec\ xs\ []\ (\%x\ xs\ xsa.\ xsa\ @\ [x])$

definition

$zipWith :: ['a * 'b \Rightarrow 'c,\ 'a\ list * 'b\ list] \Rightarrow 'c\ list$ **where**
 $zipWith\ f\ S = (list-rec\ (fst\ S)\ (\%T.\ []))$
 $(\%x\ xs\ r.\ \%T.\ if\ null\ T\ then\ []$
 $else\ f(x,hd\ T)\ \#\ r(tl\ T))) (snd(S))$

definition

$zip :: 'a\ list * 'b\ list \Rightarrow ('a * 'b)\ list$ **where**
 $zip = zipWith\ (\%s.\ s)$

definition

$unzip :: ('a * 'b)\ list \Rightarrow ('a\ list * 'b\ list)$ **where**
 $unzip = foldr(\%(a,b)(c,d).(a\ \#\ c, b\ \#\ d))([], [])$

consts $take :: ['a\ list, nat] \Rightarrow 'a\ list$

primrec

$take-0:$ $take\ xs\ 0 = []$
 $take-Suc:$ $take\ xs\ (Suc\ n) = list-case\ []\ (\%x\ l.\ x\ \#\ take\ l\ n)\ xs$

consts $enum :: [nat, nat] \Rightarrow nat\ list$

primrec

$enum-0:$ $enum\ i\ 0 = []$
 $enum-Suc:$ $enum\ i\ (Suc\ j) = (if\ i\ <= j\ then\ enum\ i\ j\ @\ [j]\ else\ [])$

no-translations

$[x \leftarrow xs.\ P] == filter\ (\%x.\ P)\ xs$

syntax

$@Alls :: [idt,\ 'a\ list,\ bool] \Rightarrow bool$ $((2Alls\ -:-./\ -)\ 10)$

translations

$[x \leftarrow xs.\ P] == CONST\ filter(\%x.\ P)\ xs$
 $Alls\ x:xs.\ P == CONST\ list-all(\%x.\ P)\ xs$

lemma $ListI: x : list\ (range\ Leaf) ==> x : List$
by $(simp\ add: List-def)$

lemma $ListD: x : List ==> x : list\ (range\ Leaf)$

by (*simp add: List-def*)

lemma *list-unfold*: $list(A) = usum \{Numb(0)\} (uprod A (list(A)))$
by (*fast intro!: list.intros [unfolded NIL-def CONS-def]*
elim: list.cases [unfolded NIL-def CONS-def])

lemma *list-mono*: $A \leq B \implies list(A) \leq list(B)$
apply (*rule subsetI*)
apply (*erule list.induct*)
apply (*auto intro!: list.intros*)
done

lemma *list-sexp*: $list(sexp) \leq sexp$
apply (*rule subsetI*)
apply (*erule list.induct*)
apply (*unfold NIL-def CONS-def*)
apply (*auto intro: sexp.intros sexp-In0I sexp-In1I*)
done

lemmas *list-subset-sexp* = *subset-trans [OF list-mono list-sexp]*

lemma *list-induct*:
 $[[P(Nil); !!x xs. P(xs) \implies P(x \# xs)]] \implies P(l)$
apply (*unfold Nil-def Cons-def*)
apply (*rule Rep-List-inverse [THEN subst]*)
apply (*rule Rep-List [unfolded List-def, THEN list.induct], simp*)
apply (*erule Abs-List-inverse [unfolded List-def, THEN subst], blast*)
done

lemma *inj-on-Abs-list*: $inj-on Abs-List (list(range Leaf))$
apply (*rule inj-on-inverseI*)
apply (*erule Abs-List-inverse [unfolded List-def]*)
done

lemma *CONS-not-NIL [iff]*: $CONS M N \sim= NIL$
by (*simp add: NIL-def CONS-def*)

```

lemmas NIL-not-CONS [iff] = CONS-not-NIL [THEN not-sym]
lemmas CONS-neq-NIL = CONS-not-NIL [THEN notE, standard]
lemmas NIL-neq-CONS = sym [THEN CONS-neq-NIL]

lemma Cons-not-Nil [iff]:  $x \# xs \sim Nil$ 
apply (unfold Nil-def Cons-def)
apply (rule CONS-not-NIL [THEN inj-on-Abs-list [THEN inj-on-contrad]])
apply (simp-all add: list.intros rangeI Rep-List [unfolded List-def])
done

lemmas Nil-not-Cons [iff] = Cons-not-Nil [THEN not-sym, standard]
lemmas Cons-neq-Nil = Cons-not-Nil [THEN notE, standard]
lemmas Nil-neq-Cons = sym [THEN Cons-neq-Nil]

lemma CONS-CONS-eq [iff]:  $(CONS\ K\ M) = (CONS\ L\ N) = (K=L \ \&\ M=N)$ 
by (simp add: CONS-def)

declare Rep-List [THEN ListD, intro] ListI [intro]
declare list.intros [intro, simp]
declare Leaf-inject [dest!]

lemma Cons-Cons-eq [iff]:  $(x \# xs = y \# ys) = (x = y \ \&\ xs = ys)$ 
apply (simp add: Cons-def)
apply (subst Abs-List-inject)
apply (auto simp add: Rep-List-inject)
done

lemmas Cons-inject2 = Cons-Cons-eq [THEN iffD1, THEN conjE, standard]

lemma CONS-D:  $CONS\ M\ N: list(A) ==> M: A \ \&\ N: list(A)$ 
by (induct L == CONS\ M\ N set: list) auto

lemma sexp-CONS-D:  $CONS\ M\ N: sexp ==> M: sexp \ \&\ N: sexp$ 
apply (simp add: CONS-def In1-def)
apply (fast dest!: Scons-D)
done

lemma not-CONS-self:  $N: list(A) ==> !M. N \sim CONS\ M\ N$ 
by (erule list.induct, simp-all)

lemma not-Cons-self2:  $\forall x. l \sim x \# l$ 
by (induct-tac l rule: list-induct, simp-all)

```


lemma *neq-Nil-conv2*: $(xs \sim = []) = (\exists y \text{ } ys. xs = y \# ys)$
by (*induct-tac xs rule: list-induct, auto*)

lemma *List-case-NIL* [*simp*]: *List-case c h NIL = c*
by (*simp add: List-case-def NIL-def*)

lemma *List-case-CONS* [*simp*]: *List-case c h (CONS M N) = h M N*
by (*simp add: List-case-def CONS-def*)

lemma *List-rec-unfold-lemma*:
 $(\%M. \text{List-rec } M \text{ } c \text{ } d) ==$
 $\text{wfrec } (\text{pred-sexp } ^+) (\%g. \text{List-case } c \text{ } (\%x \text{ } y. d \text{ } x \text{ } y \text{ } (g \text{ } y)))$
by (*simp add: List-rec-def*)

lemmas *List-rec-unfold =*
 $\text{def-wfrec } [OF \text{ List-rec-unfold-lemma wf-pred-sexp } [THEN \text{ wf-trancl}],$
 $\text{standard}]$

lemma *pred-sexp-CONS-I1*:
 $[M : \text{sexp}; N : \text{sexp}] ==> (M, \text{CONS } M \text{ } N) : \text{pred-sexp } ^+$
by (*simp add: CONS-def In1-def*)

lemma *pred-sexp-CONS-I2*:
 $[M : \text{sexp}; N : \text{sexp}] ==> (N, \text{CONS } M \text{ } N) : \text{pred-sexp } ^+$
by (*simp add: CONS-def In1-def*)

lemma *pred-sexp-CONS-D*:
 $(\text{CONS } M1 \text{ } M2, N) : \text{pred-sexp } ^+ ==>$
 $(M1, N) : \text{pred-sexp } ^+ \ \& \ (M2, N) : \text{pred-sexp } ^+$
apply (*frule pred-sexp-subset-Sigma [THEN trancl-subset-Sigma, THEN subsetD]*)
apply (*blast dest!: sexp-CONS-D intro: pred-sexp-CONS-I1 pred-sexp-CONS-I2*
 $\text{trans-trancl } [THEN \text{ transD}]$)
done

```

lemma List-rec-NIL [simp]: List-rec NIL c h = c
apply (rule List-rec-unfold [THEN trans])
apply (simp add: List-case-NIL)
done

```

```

lemma List-rec-CONS [simp]:
  [| M: sexp; N: sexp |]
  ==> List-rec (CONS M N) c h = h M N (List-rec N c h)
apply (rule List-rec-unfold [THEN trans])
apply (simp add: pred-sexp-CONS-I2)
done

```

```

lemmas Rep-List-in-sexp =
  subsetD [OF range-Leaf-subset-sexp [THEN list-subset-sexp]
    Rep-List [THEN ListD]]

```

```

lemma list-rec-Nil [simp]: list-rec Nil c h = c
by (simp add: list-rec-def ListI [THEN Abs-List-inverse] Nil-def)

```

```

lemma list-rec-Cons [simp]: list-rec (a#l) c h = h a l (list-rec l c h)
by (simp add: list-rec-def ListI [THEN Abs-List-inverse] Cons-def
  Rep-List-inverse Rep-List [THEN ListD] inj-Leaf Rep-List-in-sexp)

```

```

lemma List-rec-type:
  [| M: list(A);
    A <= sexp;
    c: C(NIL);
    !!x y r. [| x: A; y: list(A); r: C(y) |] ==> h x y r: C(CONS x y)
  |] ==> List-rec M c h : C(M :: 'a item)
apply (erule list.induct, simp)
apply (insert list-subset-sexp)
apply (subst List-rec-CONS, blast+)
done

```

```

lemma Rep-map-Nil [simp]: Rep-map f Nil = NIL
by (simp add: Rep-map-def)

```

```

lemma Rep-map-Cons [simp]:

```

$Rep\text{-}map\ f(x\#xs) = CONS(f\ x)(Rep\text{-}map\ f\ xs)$
by (*simp add: Rep-map-def*)

lemma *Rep-map-type*: $(!!x. f(x): A) ==> Rep\text{-}map\ f\ xs: list(A)$
apply (*simp add: Rep-map-def*)
apply (*rule list-induct, auto*)
done

lemma *Abs-map-NIL* [*simp*]: $Abs\text{-}map\ g\ NIL = Nil$
by (*simp add: Abs-map-def*)

lemma *Abs-map-CONS* [*simp*]:
 $[! M: sexp; N: sexp] ==> Abs\text{-}map\ g\ (CONS\ M\ N) = g(M) \# Abs\text{-}map\ g\ N$
by (*simp add: Abs-map-def*)

lemma *def-list-rec-NilCons*:
 $[! xs. f(xs) = list\text{-}rec\ xs\ c\ h] ==> f [] = c \ \& \ f(x\#xs) = h\ x\ xs\ (f\ xs)$
by *simp*

lemma *Abs-map-inverse*:
 $[! M: list(A); A <= sexp; !z. z: A ==> f(g(z)) = z] ==> Rep\text{-}map\ f\ (Abs\text{-}map\ g\ M) = M$
apply (*erule list.induct, simp-all*)
apply (*insert list-subset-sexp*)
apply (*subst Abs-map-CONS, blast*)
apply *blast*
apply *simp*
done

Better to have a single theorem with a conjunctive conclusion.

declare *def-list-rec-NilCons* [*OF list-case-def, simp*]

lemma *expand-list-case*:
 $P(list\text{-}case\ a\ f\ xs) = ((xs=[] \longrightarrow P\ a) \ \& \ (!y\ ys. xs=y\#ys \longrightarrow P(f\ y\ ys)))$
by (*induct-tac xs rule: list-induct, simp-all*)

declare *def-list-rec-NilCons* [*OF null-def, simp*]
declare *def-list-rec-NilCons* [*OF hd-def, simp*]
declare *def-list-rec-NilCons* [*OF tl-def, simp*]
declare *def-list-rec-NilCons* [*OF ttl-def, simp*]

```

declare def-list-rec-NilCons [OF append-def, simp]
declare def-list-rec-NilCons [OF member-def, simp]
declare def-list-rec-NilCons [OF map-def, simp]
declare def-list-rec-NilCons [OF filter-def, simp]
declare def-list-rec-NilCons [OF list-all-def, simp]

```

```

lemma def-nat-rec-0-eta:
  [| !!n. f = nat-rec c h |] ==> f(0) = c
by simp

```

```

lemma def-nat-rec-Suc-eta:
  [| !!n. f = nat-rec c h |] ==> f(Suc(n)) = h n (f n)
by simp

```

```

declare def-nat-rec-0-eta [OF nth-def, simp]
declare def-nat-rec-Suc-eta [OF nth-def, simp]

```

```

lemma length-Nil [simp]: length([]) = 0
by (simp add: length-def)

```

```

lemma length-Cons [simp]: length(a#xs) = Suc(length(xs))
by (simp add: length-def)

```

```

lemma append-assoc [simp]: (xs@ys)@zs = xs@(ys@zs)
by (induct-tac xs rule: list-induct, simp-all)

```

```

lemma append-Nil2 [simp]: xs @ [] = xs
by (induct-tac xs rule: list-induct, simp-all)

```

```

lemma mem-append [simp]: x mem (xs@ys) = (x mem xs | x mem ys)
by (induct-tac xs rule: list-induct, simp-all)

```

```

lemma mem-filter [simp]: x mem [x←xs. P x] = (x mem xs & P(x))
by (induct-tac xs rule: list-induct, simp-all)

```

```

lemma list-all-True [simp]: (Alls x:xs. True) = True

```

by (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)

lemma *list-all-conj* [*simp*]:

$list-all\ p\ (xs@ys) = ((list-all\ p\ xs) \& (list-all\ p\ ys))$

by (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)

lemma *list-all-mem-conv*: ($\text{Alls } x:xs. P(x) = (!x. x\ mem\ xs \longrightarrow P(x))$)

apply (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)

apply *blast*

done

lemma *nat-case-dist* : ($!n. P\ n = (P\ 0 \& (!n. P\ (Suc\ n)))$)

apply *auto*

apply (*induct-tac* *n*, *auto*)

done

lemma *alls-P-eq-P-nth*: ($\text{Alls } u:A. P\ u = (!n. n < length\ A \longrightarrow P(nth\ n\ A))$)

apply (*induct-tac* *A* *rule*: *list-induct*, *simp-all*)

apply (*rule trans*)

apply (*rule-tac* [2] *nat-case-dist* [*symmetric*], *simp-all*)

done

lemma *list-all-imp*:

$[!x. P\ x \longrightarrow Q\ x; (\text{Alls } x:xs. P(x)) \implies (\text{Alls } x:xs. Q(x))]$

by (*simp add*: *list-all-mem-conv*)

lemma *Abs-Rep-map*:

$(!x. f(x): sexp) \implies$

$Abs-map\ g\ (Rep-map\ f\ xs) = map\ (\%t. g(f(t)))\ xs$

apply (*induct-tac* *xs* *rule*: *list-induct*)

apply (*simp-all add*: *Rep-map-type list-sexp* [*THEN subsetD*])

done

lemma *map-ident* [*simp*]: $map(\%x. x)(xs) = xs$

by (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)

lemma *map-append* [*simp*]: $map\ f\ (xs@ys) = map\ f\ xs\ @\ map\ f\ ys$

by (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)

lemma *map-compose*: $map(f\ o\ g)(xs) = map\ f\ (map\ g\ xs)$

apply (*simp add*: *o-def*)

apply (*induct-tac xs rule: list-induct, simp-all*)
done

lemma *mem-map-aux1* [*rule-format*]:
 $x \text{ mem } (\text{map } f \ q) \dashrightarrow (\exists y. y \text{ mem } q \ \& \ x = f \ y)$
by (*induct-tac q rule: list-induct, simp-all, blast*)

lemma *mem-map-aux2* [*rule-format*]:
 $(\exists y. y \text{ mem } q \ \& \ x = f \ y) \dashrightarrow x \text{ mem } (\text{map } f \ q)$
by (*induct-tac q rule: list-induct, auto*)

lemma *mem-map*: $x \text{ mem } (\text{map } f \ q) = (\exists y. y \text{ mem } q \ \& \ x = f \ y)$
apply (*rule iffI*)
apply (*erule mem-map-aux1*)
apply (*erule mem-map-aux2*)
done

lemma *hd-append* [*rule-format*]: $A \sim = [] \dashrightarrow \text{hd}(A @ B) = \text{hd}(A)$
by (*induct-tac A rule: list-induct, auto*)

lemma *tl-append* [*rule-format*]: $A \sim = [] \dashrightarrow \text{tl}(A @ B) = \text{tl}(A) @ B$
by (*induct-tac A rule: list-induct, auto*)

lemma *take-Suc1* [*simp*]: $\text{take } [] \ (Suc \ x) = []$
by *simp*

lemma *take-Suc2* [*simp*]: $\text{take}(a \# xs)(Suc \ x) = a \# \text{take } xs \ x$
by *simp*

lemma *drop-0* [*simp*]: $\text{drop } xs \ 0 = xs$
by (*simp add: drop-def*)

lemma *drop-Suc1* [*simp*]: $\text{drop } [] \ (Suc \ x) = []$
apply (*simp add: drop-def*)
apply (*induct-tac x, auto*)
done

lemma *drop-Suc2* [*simp*]: $\text{drop}(a \# xs)(Suc \ x) = \text{drop } xs \ x$
by (*simp add: drop-def*)

lemma *copy-0* [simp]: *copy* *x* 0 = []

by (simp add: *copy-def*)

lemma *copy-Suc* [simp]: *copy* *x* (Suc *y*) = *x* # *copy* *x* *y*

by (simp add: *copy-def*)

lemma *foldl-Nil* [simp]: *foldl* *f* *a* [] = *a*

by (simp add: *foldl-def*)

lemma *foldl-Cons* [simp]: *foldl* *f* *a* (*x* # *xs*) = *foldl* *f* (*f* *a* *x*) *xs*

by (simp add: *foldl-def*)

lemma *foldr-Nil* [simp]: *foldr* *f* *a* [] = *a*

by (simp add: *foldr-def*)

lemma *foldr-Cons* [simp]: *foldr* *f* *z* (*x* # *xs*) = *f* *x* (*foldr* *f* *z* *xs*)

by (simp add: *foldr-def*)

lemma *flat-Nil* [simp]: *flat* [] = []

by (simp add: *flat-def*)

lemma *flat-Cons* [simp]: *flat* (*x* # *xs*) = *x* @ *flat* *xs*

by (simp add: *flat-def*)

lemma *rev-Nil* [simp]: *rev* [] = []

by (simp add: *rev-def*)

lemma *rev-Cons* [simp]: *rev* (*x* # *xs*) = *rev* *xs* @ [*x*]

by (simp add: *rev-def*)

lemma *zipWith-Cons-Cons* [simp]:

zipWith *f* (*a* # *as*, *b* # *bs*) = *f* (*a*, *b*) # *zipWith* *f* (*as*, *bs*)

by (simp add: *zipWith-def*)

lemma *zipWith-Nil-Nil* [simp]: *zipWith* *f* ([], []) = []

by (simp add: *zipWith-def*)

```

lemma zipWith-Cons-Nil [simp]: zipWith f (x,[]) = []
apply (simp add: zipWith-def)
apply (induct-tac x rule: list-induct, simp-all)
done

```

```

lemma zipWith-Nil-Cons [simp]: zipWith f ([],x) = []
by (simp add: zipWith-def)

```

```

lemma unzip-Nil [simp]: unzip [] = ([],[])
by (simp add: unzip-def)

```

```

lemma map-compose-ext: map(f o g) = ((map f) o (map g))
apply (simp add: o-def)
apply (rule ext)
apply (simp add: map-compose [symmetric] o-def)
done

```

```

lemma map-flat: map f (flat S) = flat(map (map f) S)
by (induct-tac S rule: list-induct, simp-all)

```

```

lemma list-all-map-eq: (ALL u:xs. f(u) = g(u))  $\longrightarrow$  map f xs = map g xs
by (induct-tac xs rule: list-induct, simp-all)

```

```

lemma filter-map-d: filter p (map f xs) = map f (filter(p o f)(xs))
by (induct-tac xs rule: list-induct, simp-all)

```

```

lemma filter-compose: filter p (filter q xs) = filter( $\%x$ . p x & q x) xs
by (induct-tac xs rule: list-induct, simp-all)

```

```

lemma filter-append [rule-format, simp]:
   $\forall B$ . filter p (A @ B) = (filter p A @ filter p B)
by (induct-tac A rule: list-induct, simp-all)

```

```

lemma length-append: length(xs@ys) = length(xs)+length(ys)
by (induct-tac xs rule: list-induct, simp-all)

```


lemma *length-map*: $\text{length}(\text{map } f \text{ } xs) = \text{length}(xs)$
by (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)

lemma *take-Nil* [*simp*]: $\text{take } [] \text{ } n = []$
by (*induct-tac* *n*, *simp-all*)

lemma *take-take-eq* [*simp*]: $\forall n. \text{take } (\text{take } xs \text{ } n) \text{ } n = \text{take } xs \text{ } n$
apply (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)
apply (*rule allI*)
apply (*induct-tac* *n*, *auto*)
done

lemma *take-take-Suc-eq1* [*rule-format*]:
 $\forall n. \text{take } (\text{take } xs (\text{Suc}(n+m))) \text{ } n = \text{take } xs \text{ } n$
apply (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)
apply (*rule allI*)
apply (*induct-tac* *n*, *auto*)
done

declare *take-Suc* [*simp del*]

lemma *take-take-1*: $\text{take } (\text{take } xs \text{ } (n+m)) \text{ } n = \text{take } xs \text{ } n$
apply (*induct-tac* *m*)
apply (*simp-all add*: *take-take-Suc-eq1*)
done

lemma *take-take-Suc-eq2* [*rule-format*]:
 $\forall n. \text{take } (\text{take } xs \text{ } n) (\text{Suc}(n+m)) = \text{take } xs \text{ } n$
apply (*induct-tac* *xs* *rule*: *list-induct*, *simp-all*)
apply (*rule allI*)
apply (*induct-tac* *n*, *auto*)
done

lemma *take-take-2*: $\text{take}(\text{take } xs \text{ } n)(n+m) = \text{take } xs \text{ } n$
apply (*induct-tac* *m*)
apply (*simp-all add*: *take-take-Suc-eq2*)
done

lemma *drop-Nil* [*simp*]: $\text{drop } [] \text{ } n = []$
by (*induct-tac* *n*, *auto*)

lemma *drop-drop* [*rule-format*]: $\forall xs. \text{drop } (\text{drop } xs \text{ } m) \text{ } n = \text{drop } xs(m+n)$
apply (*induct-tac* *m*, *auto*)
apply (*induct-tac* *xs* *rule*: *list-induct*, *auto*)
done

```

lemma take-drop [rule-format]:  $\forall xs. (take\ xs\ n) @ (drop\ xs\ n) = xs$ 
apply (induct-tac n, auto)
apply (induct-tac xs rule: list-induct, auto)
done

```

```

lemma copy-copy:  $copy\ x\ n @ copy\ x\ m = copy\ x\ (n+m)$ 
by (induct-tac n, auto)

```

```

lemma length-copy:  $length(copy\ x\ n) = n$ 
by (induct-tac n, auto)

```

```

lemma length-take [rule-format, simp]:
   $\forall xs. length(take\ xs\ n) = min\ (length\ xs)\ n$ 
apply (induct-tac n)
apply auto
apply (induct-tac xs rule: list-induct)
apply auto
done

```

```

lemma length-take-drop:  $length(take\ A\ k) + length(drop\ A\ k) = length(A)$ 
by (simp only: length-append [symmetric] take-drop)

```

```

lemma take-append [rule-format]:  $\forall A. length(A) = n \dashrightarrow take(A @ B)\ n = A$ 
apply (induct-tac n)
apply (rule allI)
apply (rule-tac [2] allI)
apply (induct-tac A rule: list-induct)
apply (induct-tac [3] A rule: list-induct, simp-all)
done

```

```

lemma take-append2 [rule-format]:
   $\forall A. length(A) = n \dashrightarrow take(A @ B)\ (n+k) = A @ take\ B\ k$ 
apply (induct-tac n)
apply (rule allI)
apply (rule-tac [2] allI)
apply (induct-tac A rule: list-induct)
apply (induct-tac [3] A rule: list-induct, simp-all)
done

```

```

lemma take-map [rule-format]:  $\forall n. take\ (map\ f\ A)\ n = map\ f\ (take\ A\ n)$ 
apply (induct-tac A rule: list-induct, simp-all)
apply (rule allI)
apply (induct-tac n, simp-all)
done

```

```

lemma drop-append [rule-format]:  $\forall A. length(A) = n \dashrightarrow drop(A @ B)\ n = B$ 
apply (induct-tac n)
apply (rule allI)

```

```

apply (rule-tac [2] allI)
apply (induct-tac A rule: list-induct)
apply (induct-tac [3] A rule: list-induct, simp-all)
done

```

```

lemma drop-append2 [rule-format]:
   $\forall A. \text{length}(A) = n \dashv\vdash \text{drop}(A @ B)(n+k) = \text{drop } B \ k$ 
apply (induct-tac n)
apply (rule allI)
apply (rule-tac [2] allI)
apply (induct-tac A rule: list-induct)
apply (induct-tac [3] A rule: list-induct, simp-all)
done

```

```

lemma drop-all [rule-format]:  $\forall A. \text{length}(A) = n \dashv\vdash \text{drop } A \ n = []$ 
apply (induct-tac n)
apply (rule allI)
apply (rule-tac [2] allI)
apply (induct-tac A rule: list-induct)
apply (induct-tac [3] A rule: list-induct, auto)
done

```

```

lemma drop-map [rule-format]:  $\forall n. \text{drop } (\text{map } f \ A) \ n = \text{map } f \ (\text{drop } A \ n)$ 
apply (induct-tac A rule: list-induct, simp-all)
apply (rule allI)
apply (induct-tac n, simp-all)
done

```

```

lemma take-all [rule-format]:  $\forall A. \text{length}(A) = n \dashv\vdash \text{take } A \ n = A$ 
apply (induct-tac n)
apply (rule allI)
apply (rule-tac [2] allI)
apply (induct-tac A rule: list-induct)
apply (induct-tac [3] A rule: list-induct, auto)
done

```

```

lemma foldl-single:  $\text{foldl } f \ a \ [b] = f \ a \ b$ 
by simp-all

```

```

lemma foldl-append [rule-format, simp]:
   $\forall a. \text{foldl } f \ a \ (A @ B) = \text{foldl } f \ (\text{foldl } f \ a \ A) \ B$ 
by (induct-tac A rule: list-induct, simp-all)

```

```

lemma foldl-map [rule-format]:
   $\forall e. \text{foldl } f \ e \ (\text{map } g \ S) = \text{foldl } (\%x \ y. f \ x \ (g \ y)) \ e \ S$ 
by (induct-tac S rule: list-induct, simp-all)

```

```

lemma foldl-neutr-distr [rule-format]:

```

```

assumes r-neutr:  $\forall a. f\ a\ e = a$ 
and r-neutl:  $\forall a. f\ e\ a = a$ 
and assoc:  $\forall a\ b\ c. f\ a\ (f\ b\ c) = f\ (f\ a\ b)\ c$ 
shows  $\forall y. f\ y\ (foldl\ f\ e\ A) = foldl\ f\ y\ A$ 
apply (induct-tac A rule: list-induct)
apply (simp-all add: r-neutr r-neutl, clarify)
apply (erule all-dupE)
apply (rule trans)
prefer 2 apply assumption
apply (simp (no-asm-use) add: assoc [THEN spec, THEN spec, THEN spec, THEN sym])
apply simp
done

```

```

lemma foldl-append-sym:
[[ !a.  $f\ a\ e = a$ ; !a.  $f\ e\ a = a$ ;
  !a b c.  $f\ a\ (f\ b\ c) = f\ (f\ a\ b)\ c$  ]]
==>  $foldl\ f\ e\ (A\ @\ B) = f\ (foldl\ f\ e\ A)\ (foldl\ f\ e\ B)$ 
apply (rule trans)
apply (rule foldl-append)
apply (rule sym)
apply (rule foldl-neutr-distr, auto)
done

```

```

lemma foldr-append [rule-format, simp]:
 $\forall a. foldr\ f\ a\ (A\ @\ B) = foldr\ f\ (foldr\ f\ a\ B)\ A$ 
apply (induct-tac A rule: list-induct, simp-all)
done

```

```

lemma foldr-map [rule-format]:  $\forall e. foldr\ f\ e\ (map\ g\ S) = foldr\ (f\ o\ g)\ e\ S$ 
apply (simp add: o-def)
apply (induct-tac S rule: list-induct, simp-all)
done

```

```

lemma foldr-Un-eq-UN:  $foldr\ op\ Un\ \{\}\ S = (UN\ X: \{t. t\ mem\ S\}.X)$ 
by (induct-tac S rule: list-induct, auto)

```

```

lemma foldr-neutr-distr:
[[ !a.  $f\ e\ a = a$ ; !a b c.  $f\ a\ (f\ b\ c) = f\ (f\ a\ b)\ c$  ]]
==>  $foldr\ f\ y\ S = f\ (foldr\ f\ e\ S)\ y$ 
by (induct-tac S rule: list-induct, auto)

```

```

lemma foldr-append2:
[[ !a.  $f\ e\ a = a$ ; !a b c.  $f\ a\ (f\ b\ c) = f\ (f\ a\ b)\ c$  ]]
==>  $foldr\ f\ e\ (A\ @\ B) = f\ (foldr\ f\ e\ A)\ (foldr\ f\ e\ B)$ 
apply auto
apply (rule foldr-neutr-distr, auto)
done

```

lemma *foldr-flat*:

$[[!a. f\ e\ a = a; !a\ b\ c. f\ a\ (f\ b\ c) = f(f\ a\ b)\ c\]]$ ==>
 $foldr\ f\ e\ (flat\ S) = (foldr\ f\ e)(map\ (foldr\ f\ e)\ S)$
apply (*induct-tac* *S* *rule*: *list-induct*)
apply (*simp-all* *del*: *foldr-append* *add*: *foldr-append2*)
done

lemma *list-all-map*: $(Alls\ x:map\ f\ xs.\ P(x)) = (Alls\ x:xs.(P\ o\ f)(x))$
by (*induct-tac* *xs* *rule*: *list-induct*, *auto*)

lemma *list-all-and*:

$(Alls\ x:xs.\ P(x) \& Q(x)) = ((Alls\ x:xs.\ P(x)) \& (Alls\ x:xs.\ Q(x)))$
by (*induct-tac* *xs* *rule*: *list-induct*, *auto*)

lemma *nth-map* [*rule-format*]:

$\forall i. i < length(A) \implies nth\ i\ (map\ f\ A) = f(nth\ i\ A)$
apply (*induct-tac* *A* *rule*: *list-induct*, *simp-all*)
apply (*rule* *allI*)
apply (*induct-tac* *i*, *auto*)
done

lemma *nth-app-cancel-right* [*rule-format*]:

$\forall i. i < length(A) \implies nth\ i\ (A @ B) = nth\ i\ A$
apply (*induct-tac* *A* *rule*: *list-induct*, *simp-all*)
apply (*rule* *allI*)
apply (*induct-tac* *i*, *simp-all*)
done

lemma *nth-app-cancel-left* [*rule-format*]:

$\forall n. n = length(A) \implies nth(n+i)(A @ B) = nth\ i\ B$
by (*induct-tac* *A* *rule*: *list-induct*, *simp-all*)

lemma *flat-append* [*simp*]: $flat(xs @ ys) = flat(xs) @ flat(ys)$

by (*induct-tac* *xs* *rule*: *list-induct*, *auto*)

lemma *filter-flat*: $filter\ p\ (flat\ S) = flat(map\ (filter\ p)\ S)$

by (*induct-tac* *S* *rule*: *list-induct*, *auto*)

lemma *rev-append* [*simp*]: $rev(xs @ ys) = rev(ys) @ rev(xs)$

by (*induct-tac* *xs* *rule*: *list-induct*, *auto*)

```

lemma rev-rev-ident [simp]:  $\text{rev}(\text{rev } l) = l$ 
by (induct-tac l rule: list-induct, auto)

lemma rev-flat:  $\text{rev}(\text{flat } ls) = \text{flat } (\text{map } \text{rev } (\text{rev } ls))$ 
by (induct-tac ls rule: list-induct, auto)

lemma rev-map-distrib:  $\text{rev}(\text{map } f \ l) = \text{map } f \ (\text{rev } l)$ 
by (induct-tac l rule: list-induct, auto)

lemma foldl-rev:  $\text{foldl } f \ b \ (\text{rev } l) = \text{foldr } (\%x \ y. f \ y \ x) \ b \ l$ 
by (induct-tac l rule: list-induct, auto)

lemma foldr-rev:  $\text{foldr } f \ b \ (\text{rev } l) = \text{foldl } (\%x \ y. f \ y \ x) \ b \ l$ 
apply (rule sym)
apply (rule trans)
apply (rule-tac [2] foldl-rev, simp)
done

end

```

12 Definition of type llist by a greatest fixed point

theory *LList* **imports** *SList* **begin**

```

coinductive-set
  llist :: 'a item set => 'a item set
  for A :: 'a item set
  where
    NIL-I:  $NIL \in \text{llist}(A)$ 
    | CONS-I:  $[| a \in A; M \in \text{llist}(A) |] \implies \text{CONS } a \ M \in \text{llist}(A)$ 

```

```

coinductive-set
  LListD :: ('a item * 'a item) set => ('a item * 'a item) set
  for r :: ('a item * 'a item) set
  where
    NIL-I:  $(NIL, NIL) \in \text{LListD}(r)$ 
    | CONS-I:  $[| (a, b) \in r; (M, N) \in \text{LListD}(r) |] \implies (\text{CONS } a \ M, \text{CONS } b \ N) \in \text{LListD}(r)$ 

```

```

typedef (LList)
  'a llist = llist(range Leaf) :: 'a item set
  by (blast intro: llist.NIL-I)

```

definition

```

list-Fun :: ['a item set, 'a item set] => 'a item set where
  — Now used exclusively for abbreviating the coinduction rule

```

$list\text{-}Fun\ A\ X = \{z. z = NIL \mid (\exists M\ a. z = CONS\ a\ M \ \&\ a \in A \ \&\ M \in X)\}$

definition

$LListD\text{-}Fun ::$
 $[(\text{'a item} * \text{'a item})set, (\text{'a item} * \text{'a item})set] \Rightarrow$
 $(\text{'a item} * \text{'a item})set \textbf{ where}$
 $LListD\text{-}Fun\ r\ X =$
 $\{z. z = (NIL, NIL) \mid$
 $(\exists M\ N\ a\ b. z = (CONS\ a\ M, CONS\ b\ N) \ \&\ (a, b) \in r \ \&\ (M, N) \in X)\}$

definition

$LNil :: \text{'a llist} \textbf{ where}$
 $\text{--- abstract constructor}$
 $LNil = Abs\text{-}LList\ NIL$

definition

$LCons :: [\text{'a}, \text{'a llist}] \Rightarrow \text{'a llist} \textbf{ where}$
 $\text{--- abstract constructor}$
 $LCons\ x\ xs = Abs\text{-}LList\ (CONS\ (Leaf\ x)\ (Rep\text{-}LList\ xs))$

definition

$llist\text{-}case :: [\text{'b}, [\text{'a}, \text{'a llist}] \Rightarrow \text{'b}, \text{'a llist}] \Rightarrow \text{'b} \textbf{ where}$
 $llist\text{-}case\ c\ d\ l =$
 $List\text{-}case\ c\ (\%x\ y. d\ (inv\ Leaf\ x)\ (Abs\text{-}LList\ y))\ (Rep\text{-}LList\ l)$

definition

$LList\text{-}corec\text{-}fun :: [nat, \text{'a} \Rightarrow (\text{'b item} * \text{'a})\ option, \text{'a}] \Rightarrow \text{'b item} \textbf{ where}$
 $LList\text{-}corec\text{-}fun\ k\ f ==$
 $nat\text{-}rec\ (\%x. \{\})$
 $(\%j\ r\ x. case\ f\ x\ of\ None \Rightarrow NIL$
 $\mid Some(z, w) \Rightarrow CONS\ z\ (r\ w))$
 k

definition

$LList\text{-}corec :: [\text{'a}, \text{'a} \Rightarrow (\text{'b item} * \text{'a})\ option] \Rightarrow \text{'b item} \textbf{ where}$
 $LList\text{-}corec\ a\ f = (\bigcup k. LList\text{-}corec\text{-}fun\ k\ f\ a)$

definition

$llist\text{-}corec :: [\text{'a}, \text{'a} \Rightarrow (\text{'b} * \text{'a})\ option] \Rightarrow \text{'b llist} \textbf{ where}$
 $llist\text{-}corec\ a\ f =$
 $Abs\text{-}LList\ (LList\text{-}corec\ a$
 $(\%z. case\ f\ z\ of\ None \Rightarrow None$
 $\mid Some(v, w) \Rightarrow Some(Leaf(v), w)))$

definition

$llistD\text{-}Fun :: (\text{'a llist} * \text{'a llist})set \Rightarrow (\text{'a llist} * \text{'a llist})set \textbf{ where}$
 $llistD\text{-}Fun(r) =$
 $prod\text{-}fun\ Abs\text{-}LList\ Abs\text{-}LList\ \text{'}$
 $LListD\text{-}Fun\ (diag(range\ Leaf))$

(*prod-fun Rep-LList Rep-LList ' r*)

The case syntax for type *'a llist*

syntax

LNil :: *logic*

LCons :: *logic*

translations

case p of LNil => a | LCons x l => b == CONST llist-case a (%x l. b) p

12.0.2 Sample function definitions. Item-based ones start with *L*

definition

Lmap :: (*'a item => 'b item*) => (*'a item => 'b item*) **where**
Lmap f M = LList-corec M (List-case None (%x M'. Some((f(x), M'))))

definition

lmap :: (*'a => 'b*) => (*'a llist => 'b llist*) **where**
lmap f l = llist-corec l (%z. case z of LNil => None
| LCons y z => Some(f(y), z))

definition

iterates :: [*'a => 'a, 'a*] => *'a llist* **where**
iterates f a = llist-corec a (%x. Some((x, f(x))))

definition

Lconst :: *'a item => 'a item* **where**
Lconst(M) == lfp(%N. CONS M N)

definition

Lappend :: [*'a item, 'a item*] => *'a item* **where**
Lappend M N = LList-corec (M,N)
(split(List-case (List-case None (%N1 N2. Some((N1, (NIL,N2))))
(%M1 M2 N. Some((M1, (M2,N))))))

definition

lappend :: [*'a llist, 'a llist*] => *'a llist* **where**
lappend l n = llist-corec (l,n)
(split(llist-case (llist-case None (%n1 n2. Some((n1, (LNil,n2))))
(%l1 l2 n. Some((l1, (l2,n))))))

Append generates its result by applying *f*, where *f*((NIL,NIL)) = None
f((NIL, CONS N1 N2)) = Some((N1, (NIL,N2))) *f*((CONS M1 M2, N)) =
Some((M1, (M2,N)))

SHOULD *LListD-Fun-CONS-I*, etc., be equations (for rewriting)?

lemmas *UN1-I = UNIV-I [THEN UN-I, standard]*

12.0.3 Simplification

declare *option.split* [*split*]

This justifies using `llist` in other recursive type definitions

```

lemma llist-mono:
  assumes subset:  $A \subseteq B$ 
  shows  $llist\ A \subseteq llist\ B$ 
proof
  fix  $x$ 
  assume  $x \in llist\ A$ 
  then show  $x \in llist\ B$ 
  proof coinduct
    case llist
    then show ?case using subset
    by cases blast+
  qed
qed

```

```

lemma llist-unfold:  $llist(A) = usum\ \{Numb(0)\}\ (uprod\ A\ (llist\ A))$ 
  by (fast intro!: llist.intros [unfolded NIL-def CONS-def]
      elim: llist.cases [unfolded NIL-def CONS-def])

```

12.1 Type checking by coinduction

... using *list-Fun* THE COINDUCTIVE DEFINITION PACKAGE COULD DO THIS!

```

lemma llist-coinduct:
   $[[\ M \in X;\ X \subseteq list-Fun\ A\ (X\ Un\ llist(A))\ ]]\ ==>\ M \in llist(A)$ 
apply (simp add: list-Fun-def)
apply (erule llist.coinduct)
apply (blast intro: elim:)
done

```

```

lemma list-Fun-NIL-I [iff]:  $NIL \in list-Fun\ A\ X$ 
by (simp add: list-Fun-def NIL-def)

```

```

lemma list-Fun-CONS-I [intro!, simp]:
   $[[\ M \in A;\ N \in X\ ]]\ ==>\ CONS\ M\ N \in list-Fun\ A\ X$ 
by (simp add: list-Fun-def CONS-def)

```

Utilise the “strong” part, i.e. *gfp*(*f*)

```

lemma list-Fun-llist-I:  $M \in llist(A) ==>\ M \in list-Fun\ A\ (X\ Un\ llist(A))$ 
apply (unfold list-Fun-def)
apply (erule llist.cases)
apply auto
done

```

12.2 *LList-corec* satisfies the desired recursion equation

A continuity result?

lemma *CONS-UN1*: $CONS\ M\ (\bigcup x. f(x)) = (\bigcup x. CONS\ M\ (f\ x))$
apply (*simp add: CONS-def In1-UN1 Scons-UN1-y*)
done

lemma *CONS-mono*: $[[\ M \subseteq M';\ N \subseteq N' \] \implies CONS\ M\ N \subseteq CONS\ M'\ N']$
apply (*simp add: CONS-def In1-mono Scons-mono*)
done

declare *LList-corec-fun-def* [*THEN def-nat-rec-0, simp*]
LList-corec-fun-def [*THEN def-nat-rec-Suc, simp*]

12.2.1 The directions of the equality are proved separately

lemma *LList-corec-subset1*:
 $LList-corec\ a\ f \subseteq$
 $(case\ f\ a\ of\ None \implies NIL \mid Some(z,w) \implies CONS\ z\ (LList-corec\ w\ f))$
apply (*unfold LList-corec-def*)
apply (*rule UN-least*)
apply (*case-tac k*)
apply (*simp-all (no-asm-simp)*)
apply (*rule allI impI subset-refl [THEN CONS-mono] UNIV-I [THEN UN-upper]*) +
done

lemma *LList-corec-subset2*:
 $(case\ f\ a\ of\ None \implies NIL \mid Some(z,w) \implies CONS\ z\ (LList-corec\ w\ f)) \subseteq$
 $LList-corec\ a\ f$
apply (*simp add: LList-corec-def*)
apply (*simp add: CONS-UN1, safe*)
apply (*rule-tac a=Suc(?k) in UN-I, simp, simp*) +
done

the recursion equation for *LList-corec* – NOT SUITABLE FOR REWRITING!

lemma *LList-corec*:
 $LList-corec\ a\ f =$
 $(case\ f\ a\ of\ None \implies NIL \mid Some(z,w) \implies CONS\ z\ (LList-corec\ w\ f))$
by (*rule equalityI LList-corec-subset1 LList-corec-subset2*) +

definitional version of same

lemma *def-LList-corec*:
 $[[\ !x. h(x) = LList-corec\ x\ f \]]$
 $\implies h(a) = (case\ f\ a\ of\ None \implies NIL \mid Some(z,w) \implies CONS\ z\ (h\ w))$
by (*simp add: LList-corec*)

A typical use of co-induction to show membership in the *gfp*. Bisimulation is $range(\%x. LList-corec\ x\ f)$

lemma *LList-corec-type*: $LList-corec\ a\ f \in llist\ UNIV$
apply (*rule-tac X = range (%x. LList-corec\ x\ ?g) in llist-coinduct*)
apply (*rule rangeI, safe*)

```

apply (subst LList-corec, simp)
done

```

12.3 *l*list equality as a *gfp*; the bisimulation principle

This theorem is actually used, unlike the many similar ones in ZF

```

lemma LListD-unfold: LListD r = dsum (diag {Numb 0}) (dprod r (LListD r))
  by (fast intro!: LListD.intros [unfolded NIL-def CONS-def]
    elim: LListD.cases [unfolded NIL-def CONS-def])

```

```

lemma LListD-implies-ntrunc-equality [rule-format]:
   $\forall M\ N. (M, N) \in \text{LListD}(\text{diag } A) \longrightarrow \text{ntrunc } k\ M = \text{ntrunc } k\ N$ 
apply (induct-tac k rule: nat-less-induct)
apply (safe del: equalityI)
apply (erule LListD.cases)
apply (safe del: equalityI)
apply (case-tac n, simp)
apply (rename-tac n')
apply (case-tac n')
apply (simp-all add: CONS-def less-Suc-eq)
done

```

The domain of the *LListD* relation

```

lemma Domain-LListD:
   $\text{Domain } (\text{LListD}(\text{diag } A)) \subseteq \text{l}list(A)$ 
apply (rule subsetI)
apply (erule llist.coinduct)
apply (simp add: NIL-def CONS-def)
apply (drule-tac  $P = \%x. xa \in \text{Domain } x$  in LListD-unfold [THEN subst], auto)
done

```

This inclusion justifies the use of coinduction to show $M = N$

```

lemma LListD-subset-diag:  $\text{LListD}(\text{diag } A) \subseteq \text{diag}(\text{l}list(A))$ 
apply (rule subsetI)
apply (rule-tac  $p = x$  in PairE, safe)
apply (rule diag-eqI)
apply (rule LListD-implies-ntrunc-equality [THEN ntrunc-equality], assumption)
apply (erule DomainI [THEN Domain-LListD [THEN subsetD]])
done

```

12.3.1 Coinduction, using *LListD-Fun*

THE COINDUCTIVE DEFINITION PACKAGE COULD DO THIS!

```

lemma LListD-Fun-mono:  $A \subseteq B \implies \text{LListD-Fun } r\ A \subseteq \text{LListD-Fun } r\ B$ 
apply (simp add: LListD-Fun-def)
apply (assumption | rule basic-monos) +
done

```

lemma *LListD-coinduct*:

```

  [| M ∈ X; X ⊆ LListD-Fun r (X Un LListD(r)) |] ==> M ∈ LListD(r)
apply (cases M)
apply (simp add: LListD-Fun-def)
apply (erule LListD.coinduct)
apply (auto)
done

```

lemma *LListD-Fun-NIL-I*: $(NIL, NIL) \in LListD-Fun\ r\ s$

by (simp add: LListD-Fun-def NIL-def)

lemma *LListD-Fun-CONS-I*:

```

  [| x ∈ A; (M, N):s |] ==> (CONS x M, CONS x N) ∈ LListD-Fun (diag A) s
by (simp add: LListD-Fun-def CONS-def, blast)

```

Utilise the "strong" part, i.e. $gfp(f)$

lemma *LListD-Fun-LListD-I*:

```

  M ∈ LListD(r) ==> M ∈ LListD-Fun r (X Un LListD(r))
apply (cases M)
apply (simp add: LListD-Fun-def)
apply (erule LListD.cases)
apply auto
done

```

This converse inclusion helps to strengthen *LList-equalityI*

lemma *diag-subset-LListD*: $diag(llist(A)) \subseteq LListD(diag\ A)$

```

apply (rule subsetI)
apply (erule LListD-coinduct)
apply (rule subsetI)
apply (erule diagE)
apply (erule ssubst)
apply (erule llist.cases)
apply (simp-all add: diagI LListD-Fun-NIL-I LListD-Fun-CONS-I)
done

```

lemma *LListD-eq-diag*: $LListD(diag\ A) = diag(llist(A))$

```

apply (rule equalityI LListD-subset-diag diag-subset-LListD)
done

```

lemma *LListD-Fun-diag-I*: $M \in llist(A) ==> (M, M) \in LListD-Fun\ (diag\ A)\ (X\ Un\ diag(llist(A)))$

```

apply (rule LListD-eq-diag [THEN subst])
apply (rule LListD-Fun-LListD-I)
apply (simp add: LListD-eq-diag diagI)
done

```

12.3.2 To show two LLists are equal, exhibit a bisimulation! [also admits true equality] Replace A by some particular set, like $\{x. \text{True}\}$???

```

lemma LList-equalityI:
  [| (M,N) ∈ r; r ⊆ LListD-Fun (diag A) (r Un diag(llist(A))) |]
  ==> M=N
apply (rule LListD-subset-diag [THEN subsetD, THEN diagE])
apply (erule LListD-coinduct)
apply (simp add: LListD-eq-diag, safe)
done

```

12.4 Finality of $l\text{list}(A)$: Uniqueness of functions defined by corecursion

We must remove *Pair-eq* because it may turn an instance of reflexivity ($h1\ b, h2\ b) = (h1\ ?x17, h2\ ?x17)$) into a conjunction! (or strengthen the Solver?)

```

declare Pair-eq [simp del]

```

abstract proof using a bisimulation

```

lemma LList-corec-unique:
  [| !!x. h1(x) = (case f x of None => NIL | Some(z,w) => CONS z (h1 w));
    !!x. h2(x) = (case f x of None => NIL | Some(z,w) => CONS z (h2 w)) |]
  ==> h1=h2
apply (rule ext)

```

next step avoids an unknown (and flexflex pair) in simplification

```

apply (rule-tac A = UNIV and r = range(%u. (h1 u, h2 u))
  in LList-equalityI)
apply (rule rangeI, safe)
apply (simp add: LListD-Fun-NIL-I UNIV-I [THEN LListD-Fun-CONS-I])
done

```

```

lemma equals-LList-corec:
  [| !!x. h(x) = (case f x of None => NIL | Some(z,w) => CONS z (h w)) |]
  ==> h = (%x. LList-corec x f)
by (simp add: LList-corec-unique LList-corec)

```

12.4.1 Obsolete proof of *LList-corec-unique*: complete induction, not coinduction

```

lemma ntrunc-one-CONS [simp]: ntrunc (Suc 0) (CONS M N) = {}
by (simp add: CONS-def ntrunc-one-In1)

```

```

lemma ntrunc-CONS [simp]:
  ntrunc (Suc(Suc(k))) (CONS M N) = CONS (ntrunc k M) (ntrunc k N)
by (simp add: CONS-def)

```

```

lemma
  assumes prem1:
    !!x. h1 x = (case f x of None => NIL | Some(z,w) => CONS z (h1 w))
  and prem2:
    !!x. h2 x = (case f x of None => NIL | Some(z,w) => CONS z (h2 w))
  shows h1=h2
apply (rule ntrunc-equality [THEN ext])
apply (rule-tac x = x in spec)
apply (induct-tac k rule: nat-less-induct)
apply (rename-tac n)
apply (rule allI)
apply (subst prem1)
apply (subst prem2, simp)
apply (intro strip)
apply (case-tac n)
apply (rename-tac [2] m)
apply (case-tac [2] m, simp-all)
done

```

12.5 Lconst: defined directly by lfp

But it could be defined by corecursion.

```

lemma Lconst-fun-mono: mono(CONS(M))
apply (rule monoI subset-refl CONS-mono)
apply assumption
done

```

$$Lconst(M) = CONS\ M\ (Lconst\ M)$$

```

lemmas Lconst = Lconst-fun-mono [THEN Lconst-def [THEN def-lfp-unfold]]

```

A typical use of co-induction to show membership in the gfp. The containing set is simply the singleton $\{Lconst(M)\}$.

```

lemma Lconst-type:  $M \in A \implies Lconst(M) : llist(A)$ 
apply (rule singletonI [THEN llist-coinduct], safe)
apply (rule-tac  $P = \%u. u \in ?A$  in Lconst [THEN ssubst])
apply (assumption | rule list-Fun-CONS-I singletonI UnI1)
done

```

```

lemma Lconst-eq-LList-corec:  $Lconst(M) = LList-corec\ M\ (\%x. Some(x,x))$ 
apply (rule equals-LList-corec [THEN fun-cong], simp)
apply (rule Lconst)
done

```

Thus we could have used gfp in the definition of Lconst

```

lemma gfp-Lconst-eq-LList-corec:  $gfp(\%N. CONS\ M\ N) = LList-corec\ M\ (\%x. Some(x,x))$ 
apply (rule equals-LList-corec [THEN fun-cong], simp)

```

apply (rule *Lconst-fun-mono* [THEN *gfp-unfold*])
done

12.6 Isomorphisms

lemma *LListI*: $x \in \text{llist } (\text{range } \text{Leaf}) \implies x \in \text{LList}$
by (simp add: *LList-def*)

lemma *LListD*: $x \in \text{LList} \implies x \in \text{llist } (\text{range } \text{Leaf})$
by (simp add: *LList-def*)

12.6.1 Distinctness of constructors

lemma *LCons-not-LNil* [iff]: $\sim \text{LCons } x \text{ } xs = \text{LNil}$
apply (simp add: *LNil-def* *LCons-def*)
apply (subst *Abs-LList-inject*)
apply (rule *llist.intros* *CONS-not-NIL* *rangeI* *LListI* *Rep-LList* [THEN *LListD*])
done

lemmas *LNil-not-LCons* [iff] = *LCons-not-LNil* [THEN *not-sym*, *standard*]

12.6.2 llist constructors

lemma *Rep-LList-LNil*: $\text{Rep-LList } \text{LNil} = \text{NIL}$
apply (simp add: *LNil-def*)
apply (rule *llist.NIL-I* [THEN *LListI*, THEN *Abs-LList-inverse*])
done

lemma *Rep-LList-LCons*: $\text{Rep-LList}(\text{LCons } x \text{ } l) = \text{CONS } (\text{Leaf } x) (\text{Rep-LList } l)$
apply (simp add: *LCons-def*)
apply (rule *llist.CONS-I* [THEN *LListI*, THEN *Abs-LList-inverse*]
 $\text{rangeI } \text{Rep-LList}$ [THEN *LListD*])
done

12.6.3 Injectiveness of CONS and LCons

lemma *CONS-CONS-eq2*: $(\text{CONS } M \text{ } N = \text{CONS } M' \text{ } N') = (M = M' \ \& \ N = N')$
apply (simp add: *CONS-def*)
done

lemmas *CONS-inject* = *CONS-CONS-eq* [THEN *iffD1*, THEN *conjE*, *standard*]

For reasoning about abstract llist constructors

declare *Rep-LList* [THEN *LListD*, *intro*] *LListI* [*intro*]
declare *llist.intros* [*intro*]

lemma *LCons-LCons-eq* [iff]: $(\text{LCons } x \text{ } xs = \text{LCons } y \text{ } ys) = (x = y \ \& \ xs = ys)$
apply (simp add: *LCons-def*)
apply (subst *Abs-LList-inject*)
apply (auto simp add: *Rep-LList-inject*)

done

```

lemma CONS-D2: CONS M N ∈ llist(A) ==> M ∈ A & N ∈ llist(A)
apply (erule llist.cases)
apply (erule CONS-neq-NIL, fast)
done

```

12.7 Reasoning about $llist(A)$

A special case of *list-equality* for functions over lazy lists

```

lemma LList-fun-equalityI:
  [| M ∈ llist(A); g(NIL): llist(A);
    f(NIL)=g(NIL);
    !!x l. [| x∈A; l ∈ llist(A) |] ==>
      (f(CONS x l),g(CONS x l)) ∈
        LListD-Fun (diag A) ((%u.(f(u),g(u))) `llist(A) Un
          diag(llist(A)))
  |] ==> f(M) = g(M)
apply (rule LList-equalityI)
apply (erule imageI)
apply (rule image-subsetI)
apply (erule-tac a=x in llist.cases)
apply (erule ssbst, erule ssbst, erule LListD-Fun-diag-I, blast)
done

```

12.8 The functional $Lmap$

```

lemma Lmap-NIL [simp]: Lmap f NIL = NIL
by (rule Lmap-def [THEN def-LList-corec, THEN trans], simp)

lemma Lmap-CONS [simp]: Lmap f (CONS M N) = CONS (f M) (Lmap f N)
by (rule Lmap-def [THEN def-LList-corec, THEN trans], simp)

```

Another type-checking proof by coinduction

```

lemma Lmap-type:
  [| M ∈ llist(A); !!x. x∈A ==> f(x):B |] ==> Lmap f M ∈ llist(B)
apply (erule imageI [THEN llist-coinduct], safe)
apply (erule llist.cases, simp-all)
done

```

This type checking rule synthesises a sufficiently large set for f

```

lemma Lmap-type2: M ∈ llist(A) ==> Lmap f M ∈ llist(f`A)
apply (erule Lmap-type)
apply (erule imageI)
done

```


12.8.1 Two easy results about *Lmap*

```
lemma Lmap-compose:  $M \in \text{llist}(A) \implies \text{Lmap } (f \circ g) \ M = \text{Lmap } f \ (\text{Lmap } g \ M)$   
apply (simp add: o-def)  
apply (drule imageI)  
apply (erule LList-equalityI, safe)  
apply (erule llist.cases, simp-all)  
apply (rule LListD-Fun-NIL-I imageI UnI1 rangeI [THEN LListD-Fun-CONS-I]) +  
apply assumption  
done
```

```
lemma Lmap-ident:  $M \in \text{llist}(A) \implies \text{Lmap } (\%x. x) \ M = M$   
apply (drule imageI)  
apply (erule LList-equalityI, safe)  
apply (erule llist.cases, simp-all)  
apply (rule LListD-Fun-NIL-I imageI UnI1 rangeI [THEN LListD-Fun-CONS-I]) +  
apply assumption  
done
```

12.9 *Lappend* – its two arguments cause some complications!

```
lemma Lappend-NIL-NIL [simp]:  $\text{Lappend } \text{NIL } \text{NIL} = \text{NIL}$   
apply (simp add: Lappend-def)  
apply (rule LList-corec [THEN trans], simp)  
done
```

```
lemma Lappend-NIL-CONS [simp]:  
   $\text{Lappend } \text{NIL } (\text{CONS } N \ N') = \text{CONS } N \ (\text{Lappend } \text{NIL } N')$   
apply (simp add: Lappend-def)  
apply (rule LList-corec [THEN trans], simp)  
done
```

```
lemma Lappend-CONS [simp]:  
   $\text{Lappend } (\text{CONS } M \ M') \ N = \text{CONS } M \ (\text{Lappend } M' \ N)$   
apply (simp add: Lappend-def)  
apply (rule LList-corec [THEN trans], simp)  
done
```

```
declare llist.intros [simp] LListD-Fun-CONS-I [simp]  
  range-eqI [simp] image-eqI [simp]
```

```
lemma Lappend-NIL [simp]:  $M \in \text{llist}(A) \implies \text{Lappend } \text{NIL } M = M$   
by (erule LList-fun-equalityI, simp-all)
```

```
lemma Lappend-NIL2:  $M \in \text{llist}(A) \implies \text{Lappend } M \ \text{NIL} = M$   
by (erule LList-fun-equalityI, simp-all)
```

12.9.1 Alternative type-checking proofs for *Lappend*

weak co-induction: bisimulation and case analysis on both variables

```

lemma Lappend-type: [|  $M \in \text{llist}(A)$ ;  $N \in \text{llist}(A)$  |] ==> Lappend  $M N \in \text{llist}(A)$ 
apply (rule-tac  $X = \bigcup u \in \text{llist}(A) . \bigcup v \in \text{llist}(A) . \{Lappend\ u\ v\}$  in llist-coinduct)
apply fast
apply safe
apply (erule-tac  $a = u$  in llist.cases)
apply (erule-tac  $a = v$  in llist.cases, simp-all, blast)
done

```

strong co-induction: bisimulation and case analysis on one variable

```

lemma Lappend-type': [|  $M \in \text{llist}(A)$ ;  $N \in \text{llist}(A)$  |] ==> Lappend  $M N \in \text{llist}(A)$ 
apply (rule-tac  $X = (\%u. Lappend\ u\ N)$  'llist ( $A$ )' in llist-coinduct)
apply (erule imageI)
apply (rule image-subsetI)
apply (erule-tac  $a = x$  in llist.cases)
apply (simp add: list-Fun-llist-I, simp)
done

```

12.10 Lazy lists as the type '*a llist* – strongly typed versions of above

12.10.1 *llist-case*: case analysis for '*a llist*

```

declare LListI [THEN Abs-LList-inverse, simp]
declare Rep-LList-inverse [simp]
declare Rep-LList [THEN LListD, simp]
declare rangeI [simp] inj-Leaf [simp]

```

```

lemma llist-case-LNil [simp]: llist-case  $c\ d\ LNil = c$ 
by (simp add: llist-case-def LNil-def)

```

```

lemma llist-case-LCons [simp]: llist-case  $c\ d\ (LCons\ M\ N) = d\ M\ N$ 
by (simp add: llist-case-def LCons-def)

```

Elimination is case analysis, not induction.

```

lemma llistE: [|  $l=LNil \implies P$ ;  $\forall x\ l'.\ l=LCons\ x\ l' \implies P$  |] ==>  $P$ 
apply (rule Rep-LList [THEN LListD, THEN llist.cases])
  apply (simp add: Rep-LList-LNil [symmetric] Rep-LList-inject, blast)
apply (erule LListI [THEN Rep-LList-cases], clarify)
apply (simp add: Rep-LList-LCons [symmetric] Rep-LList-inject, blast)
done

```

12.10.2 *llist-corec*: corecursion for '*a llist*

Lemma for the proof of *llist-corec*

```

lemma LList-corec-type2:
  LList-corec a
    (%z. case f z of None => None | Some(v,w) => Some(Leaf(v),w))
    ∈ llist(range Leaf)
apply (rule-tac X = range (%x. LList-corec x ?g) in llist-coinduct)
apply (rule rangeI, safe)
apply (subst LList-corec, force)
done

```

```

lemma llist-corec:
  llist-corec a f =
    (case f a of None => LNil | Some(z,w) => LCons z (llist-corec w f))
apply (unfold llist-corec-def LNil-def LCons-def)
apply (subst LList-corec)
apply (case-tac f a)
apply (simp add: LList-corec-type2)
apply (force simp add: LList-corec-type2)
done

```

definitional version of same

```

lemma def-llist-corec:
  [| !!x. h(x) = llist-corec x f |] ==>
    h(a) = (case f a of None => LNil | Some(z,w) => LCons z (h w))
by (simp add: llist-corec)

```

12.11 Proofs about type 'a llist functions

12.12 Deriving *llist-equalityI* – *llist* equality is a bisimulation

```

lemma LListD-Fun-subset-Times-llist:
  r ⊆ (llist A) <*> (llist A)
    ==> LListD-Fun (diag A) r ⊆ (llist A) <*> (llist A)
by (auto simp add: LListD-Fun-def)

```

```

lemma subset-Times-llist:
  prod-fun Rep-LList Rep-LList ‘ r ⊆
    (llist(range Leaf) <*> (llist(range Leaf)))
by (blast intro: Rep-LList [THEN LListD])

```

```

lemma prod-fun-lemma:
  r ⊆ (llist(range Leaf) <*> (llist(range Leaf)))
    ==> prod-fun (Rep-LList o Abs-LList) (Rep-LList o Abs-LList) ‘ r ⊆ r
apply safe
apply (erule subsetD [THEN SigmaE2], assumption)
apply (simp add: LListI [THEN Abs-LList-inverse])
done

```

```

lemma prod-fun-range-eq-diag:
  prod-fun Rep-LList Rep-LList ‘ range(%x. (x, x)) =
    diag(llist(range Leaf))

```

```

apply (rule equalityI, blast)
apply (fast elim: LListI [THEN Abs-LList-inverse, THEN subst])
done

```

Used with *lfilter*

```

lemma llistD-Fun-mono:
   $A \subseteq B \implies \text{llistD-Fun } A \subseteq \text{llistD-Fun } B$ 
apply (simp add: llistD-Fun-def prod-fun-def, auto)
apply (rule image-eqI)
prefer 2 apply (blast intro: rev-subsetD [OF - LListD-Fun-mono], force)
done

```

12.12.1 To show two llists are equal, exhibit a bisimulation! [also admits true equality]

```

lemma llist-equalityI:
   $[(l1, l2) \in r; r \subseteq \text{llistD-Fun}(r \text{ Un range } (\%x.(x, x)))] \implies l1 = l2$ 
apply (simp add: llistD-Fun-def)
apply (rule Rep-LList-inject [THEN iffD1])
apply (rule-tac  $r = \text{prod-fun Rep-LList Rep-LList 'r and } A = \text{range (Leaf) in LList-equalityI}$ )
apply (erule prod-fun-imageI)
apply (erule image-mono [THEN subset-trans])
apply (rule image-compose [THEN subst])
apply (rule prod-fun-compose [THEN subst])
apply (subst image-Un)
apply (subst prod-fun-range-eq-diag)
apply (rule LListD-Fun-subset-Times-llist [THEN prod-fun-lemma])
apply (rule subset-Times-llist [THEN Un-least])
apply (rule diag-subset-Times)
done

```

12.12.2 Rules to prove the 2nd premise of *llist-equalityI*

```

lemma llistD-Fun-LNil-I [simp]:  $(LNil, LNil) \in \text{llistD-Fun}(r)$ 
apply (simp add: llistD-Fun-def LNil-def)
apply (rule LListD-Fun-NIL-I [THEN prod-fun-imageI])
done

```

```

lemma llistD-Fun-LCons-I [simp]:
   $(l1, l2):r \implies (LCons x l1, LCons x l2) \in \text{llistD-Fun}(r)$ 
apply (simp add: llistD-Fun-def LCons-def)
apply (rule rangeI [THEN LListD-Fun-CONS-I, THEN prod-fun-imageI])
apply (erule prod-fun-imageI)
done

```

Utilise the "strong" part, i.e. *gfp(f)*

```

lemma llistD-Fun-range-I:  $(l, l) \in \text{llistD-Fun}(r \text{ Un range } (\%x.(x, x)))$ 
apply (simp add: llistD-Fun-def)

```

```

apply (rule Rep-LList-inverse [THEN subst])
apply (rule prod-fun-imageI)
apply (subst image-Un)
apply (subst prod-fun-range-eq-diag)
apply (rule Rep-LList [THEN LListD, THEN LListD-Fun-diag-I])
done

```

A special case of *list-equality* for functions over lazy lists

```

lemma llist-fun-equalityI:
  [|  $f(LNil) = g(LNil)$ ;
    !! $x\ l.$  ( $f(LCons\ x\ l), g(LCons\ x\ l)$ )
       $\in llistD-Fun(range(\%u. (f(u), g(u)))\ Un\ range(\%v. (v, v)))$ 
  |] ==>  $f(l) = (g(l :: 'a\ llist) :: 'b\ llist)$ 
apply (rule-tac  $r = range\ (\%u. (f\ (u), g\ (u)))$  in llist-equalityI)
apply (rule rangeI, clarify)
apply (rule-tac  $l = u$  in llistE)
apply (simp-all add: llistD-Fun-range-I)
done

```

12.13 The functional *lmap*

```

lemma lmap-LNil [simp]:  $lmap\ f\ LNil = LNil$ 
by (rule lmap-def [THEN def-llist-corec, THEN trans], simp)

```

```

lemma lmap-LCons [simp]:  $lmap\ f\ (LCons\ M\ N) = LCons\ (f\ M)\ (lmap\ f\ N)$ 
by (rule lmap-def [THEN def-llist-corec, THEN trans], simp)

```

12.13.1 Two easy results about *lmap*

```

lemma lmap-compose [simp]:  $lmap\ (f\ o\ g)\ l = lmap\ f\ (lmap\ g\ l)$ 
by (rule-tac  $l = l$  in llist-fun-equalityI, simp-all)

```

```

lemma lmap-ident [simp]:  $lmap\ (\%x. x)\ l = l$ 
by (rule-tac  $l = l$  in llist-fun-equalityI, simp-all)

```

12.14 *iterates* – *llist-fun-equalityI* cannot be used!

```

lemma iterates:  $iterates\ f\ x = LCons\ x\ (iterates\ f\ (f\ x))$ 
by (rule iterates-def [THEN def-llist-corec, THEN trans], simp)

```

```

lemma lmap-iterates [simp]:  $lmap\ f\ (iterates\ f\ x) = iterates\ f\ (f\ x)$ 
apply (rule-tac  $r = range\ (\%u. (lmap\ f\ (iterates\ f\ u), iterates\ f\ (f\ u)))$  in llist-equalityI)
apply (rule rangeI, safe)
apply (rule-tac  $x1 = f\ (u)$  in iterates [THEN ssubst])
apply (rule-tac  $x1 = u$  in iterates [THEN ssubst], simp)
done

```

```

lemma iterates-lmap:  $iterates\ f\ x = LCons\ x\ (lmap\ f\ (iterates\ f\ x))$ 
apply (subst lmap-iterates)
apply (rule iterates)

```

done

12.15 A rather complex proof about iterates – cf Andy Pitts

12.15.1 Two lemmas about $\text{natrec } n \ x \ (\%m. g)$, which is essentially $(g \hat{\ }^n)(x)$

lemma *fun-power-lmap*: $\text{nat-rec } (LCons \ b \ l) \ (\%m. \text{lmap}(f)) \ n =$
 $LCons \ (\text{nat-rec } b \ (\%m. f) \ n) \ (\text{nat-rec } l \ (\%m. \text{lmap}(f)) \ n)$
by (*induct-tac* n , *simp-all*)

lemma *fun-power-Suc*: $\text{nat-rec } (g \ x) \ (\%m. g) \ n = \text{nat-rec } x \ (\%m. g) \ (Suc \ n)$
by (*induct-tac* n , *simp-all*)

lemmas *Pair-cong* = *refl* [*THEN cong*, *THEN cong*, of **concl**: *Pair*]

The bisimulation consists of $\{(\text{lmap}(f) \hat{\ }^n (h(u)), \text{lmap}(f) \hat{\ }^n (\text{iterates}(f, u)))\}$
for all u and all $n::\text{nat}$.

lemma *iterates-equality*:

$(!!x. h(x) = LCons \ x \ (\text{lmap } f \ (h \ x))) ==> h = \text{iterates}(f)$

apply (*rule ext*)

apply (*rule-tac*

$r = \bigcup u. \text{range } (\%n. (\text{nat-rec } (h \ u) \ (\%m \ y. \text{lmap } f \ y) \ n,$
 $\text{nat-rec } (\text{iterates } f \ u) \ (\%m \ y. \text{lmap } f \ y) \ n))$

in *llist-equalityI*)

apply (*rule UN1-I range-eqI Pair-cong nat-rec-0 [symmetric]*)+

apply *clarify*

apply (*subst iterates, atomize*)

apply (*drule-tac x=u in spec*)

apply (*erule ssubst*)

apply (*subst fun-power-lmap*)

apply (*subst fun-power-lmap*)

apply (*rule llistD-Fun-LCons-I*)

apply (*rule lmap-iterates [THEN subst]*)

apply (*subst fun-power-Suc*)

apply (*subst fun-power-Suc, blast*)

done

12.16 *lappend* – its two arguments cause some complications!

lemma *lappend-LNil-LNil [simp]*: $\text{lappend } LNil \ LNil = LNil$

apply (*simp add: lappend-def*)

apply (*rule llist-corec [THEN trans], simp*)

done

lemma *lappend-LNil-LCons [simp]*:

$\text{lappend } LNil \ (LCons \ l \ l') = LCons \ l \ (\text{lappend } LNil \ l')$

apply (*simp add: lappend-def*)

apply (*rule llist-corec [THEN trans], simp*)

done

lemma *lappend-LCons* [simp]:
 lappend (LCons *l l'*) *N* = LCons *l* (*lappend l' N*)
apply (*simp add: lappend-def*)
apply (*rule llist-corec [THEN trans], simp*)
done

lemma *lappend-LNil* [simp]: *lappend LNil l* = *l*
by (*rule-tac l = l in llist-fun-equalityI, simp-all*)

lemma *lappend-LNil2* [simp]: *lappend l LNil* = *l*
by (*rule-tac l = l in llist-fun-equalityI, simp-all*)

The infinite first argument blocks the second

lemma *lappend-iterates* [simp]: *lappend (iterates f x) N* = *iterates f x*
apply (*rule-tac r = range (%u. (lappend (iterates f u) N, iterates f u))*)
 in *llist-equalityI*)
 apply (*rule rangeI, safe*)
 apply (*subst (1 2) iterates*)
 apply *simp*
done

12.16.1 Two proofs that *lmap* distributes over *lappend*

Long proof requiring case analysis on both both arguments

lemma *lmap-lappend-distrib*:
 lmap f (lappend l n) = *lappend (lmap f l) (lmap f n)*
apply (*rule-tac r = $\bigcup n.$ range (%l. (lmap f (lappend l n),*
 lappend (lmap f l) (lmap f n))))
 in *llist-equalityI*)
apply (*rule UN1-I*)
apply (*rule rangeI, safe*)
apply (*rule-tac l = l in llistE*)
apply (*rule-tac l = n in llistE, simp-all*)
apply (*blast intro: llistD-Fun-LCons-I*)
done

Shorter proof of theorem above using *llist-equalityI* as strong coinduction

lemma *lmap-lappend-distrib'*:
 lmap f (lappend l n) = *lappend (lmap f l) (lmap f n)*
by (*rule-tac l = l in llist-fun-equalityI, auto*)

Without strong coinduction, three case analyses might be needed

lemma *lappend-assoc'*: *lappend (lappend l1 l2) l3* = *lappend l1 (lappend l2 l3)*
by (*rule-tac l = l1 in llist-fun-equalityI, auto*)

end

13 The "filter" functional for coinductive lists – defined by a combination of induction and coinduction

theory *LFilter* **imports** *LList* **begin**

inductive-set

```
findRel      :: ('a => bool) => ('a llist * 'a llist) set
for p :: 'a => bool
where
  found: p x ==> (LCons x l, LCons x l) ∈ findRel p
| seek: [| ~p x; (l,l') ∈ findRel p |] ==> (LCons x l, l') ∈ findRel p
```

declare *findRel.intros* [intro]

definition

```
find      :: ['a => bool, 'a llist] => 'a llist where
find p l = (SOME l'. (l,l'): findRel p | (l' = LNil & l ~: Domain(findRel p)))
```

definition

```
lfilter :: ['a => bool, 'a llist] => 'a llist where
lfilter p l = llist-corec l (%l. case find p l of
                                LNil => None
                                | LCons y z => Some(y,z))
```

13.1 *findRel*: basic laws

inductive-cases

```
findRel-LConsE [elim!]: (LCons x l, l'') ∈ findRel p
```

lemma *findRel-functional* [rule-format]:

```
(l,l'): findRel p ==> (l,l''): findRel p --> l'' = l'
```

by (erule *findRel.induct*, auto)

lemma *findRel-imp-LCons* [rule-format]:

```
(l,l'): findRel p ==> ∃ x l''. l' = LCons x l'' & p x
```

by (erule *findRel.induct*, auto)

lemma *findRel-LNil* [elim!]: (LNil,l): *findRel* p ==> R

by (blast elim: *findRel.cases*)

13.2 Properties of *Domain* (*findRel* p)

lemma *LCons-Domain-findRel* [simp]:

```
LCons x l ∈ Domain(findRel p) = (p x | l ∈ Domain(findRel p))
```

by auto

lemma *Domain-findRel-iff*:

$(l \in \text{Domain } (\text{findRel } p)) = (\exists x \, l'. (l, \text{LCons } x \, l') \in \text{findRel } p \ \& \ p \, x)$
by (*blast dest: findRel-imp-LCons*)

lemma *Domain-findRel-mono*:

$[!x. p \, x ==> q \, x] ==> \text{Domain } (\text{findRel } p) \leq \text{Domain } (\text{findRel } q)$
apply *clarify*
apply (*erule findRel.induct, blast+*)
done

13.3 *find*: basic equations

lemma *find-LNil [simp]*: $\text{find } p \, \text{LNil} = \text{LNil}$
by (*unfold find-def, blast*)

lemma *findRel-imp-find [simp]*: $(l, l') \in \text{findRel } p ==> \text{find } p \, l = l'$
apply (*unfold find-def*)
apply (*blast dest: findRel-functional*)
done

lemma *find-LCons-found*: $p \, x ==> \text{find } p \, (\text{LCons } x \, l) = \text{LCons } x \, l$
by (*blast intro: findRel-imp-find*)

lemma *diverge-find-LNil [simp]*: $l \sim: \text{Domain}(\text{findRel } p) ==> \text{find } p \, l = \text{LNil}$
by (*unfold find-def, blast*)

lemma *find-LCons-seek*: $\sim (p \, x) ==> \text{find } p \, (\text{LCons } x \, l) = \text{find } p \, l$
apply (*case-tac LCons x l \in Domain (findRel p)*)
apply *auto*
apply (*blast intro: findRel-imp-find*)
done

lemma *find-LCons [simp]*:
 $\text{find } p \, (\text{LCons } x \, l) = (\text{if } p \, x \text{ then } \text{LCons } x \, l \text{ else } \text{find } p \, l)$
by (*simp add: find-LCons-seek find-LCons-found*)

13.4 *lfilter*: basic equations

lemma *lfilter-LNil [simp]*: $\text{lfilter } p \, \text{LNil} = \text{LNil}$
by (*rule lfilter-def [THEN def-llist-corec, THEN trans], simp*)

lemma *diverge-lfilter-LNil [simp]*:
 $l \sim: \text{Domain}(\text{findRel } p) ==> \text{lfilter } p \, l = \text{LNil}$
by (*rule lfilter-def [THEN def-llist-corec, THEN trans], simp*)

lemma *lfilter-LCons-found*:
 $p \, x ==> \text{lfilter } p \, (\text{LCons } x \, l) = \text{LCons } x \, (\text{lfilter } p \, l)$
by (*rule lfilter-def [THEN def-llist-corec, THEN trans], simp*)

lemma *findRel-imp-lfilter [simp]*:
 $(l, \text{LCons } x \, l') \in \text{findRel } p ==> \text{lfilter } p \, l = \text{LCons } x \, (\text{lfilter } p \, l')$

```

by (rule lfilter-def [THEN def-llist-corec, THEN trans], simp)

lemma lfilter-LCons-seek:  $\sim (p\ x) \implies \text{lfilter } p\ (LCons\ x\ l) = \text{lfilter } p\ l$ 
apply (rule lfilter-def [THEN def-llist-corec, THEN trans], simp)
apply (case-tac  $LCons\ x\ l \in Domain\ (findRel\ p)$ )
  apply (simp add: Domain-findRel-iff, auto)
done

lemma lfilter-LCons [simp]:
   $\text{lfilter } p\ (LCons\ x\ l) =$ 
     $(if\ p\ x\ then\ LCons\ x\ (\text{lfilter } p\ l)\ else\ \text{lfilter } p\ l)$ 
by (simp add: lfilter-LCons-found lfilter-LCons-seek)

declare llistD-Fun-LNil-I [intro!] llistD-Fun-LCons-I [intro!]

lemma lfilter-eq-LNil:  $\text{lfilter } p\ l = LNil \implies l \sim: Domain(findRel\ p)$ 
apply (auto iff: Domain-findRel-iff)
done

lemma lfilter-eq-LCons [rule-format]:
   $\text{lfilter } p\ l = LCons\ x\ l' \implies$ 
     $(\exists l''.\ l' = \text{lfilter } p\ l'' \ \&\ (l, LCons\ x\ l'') \in findRel\ p)$ 
apply (subst lfilter-def [THEN def-llist-corec])
apply (case-tac  $l \in Domain\ (findRel\ p)$ )
  apply (auto iff: Domain-findRel-iff)
done

lemma lfilter-cases:  $\text{lfilter } p\ l = LNil \mid$ 
   $(\exists y\ l'.\ \text{lfilter } p\ l = LCons\ y\ (\text{lfilter } p\ l') \ \&\ p\ y)$ 
apply (case-tac  $l \in Domain\ (findRel\ p)$ )
apply (auto iff: Domain-findRel-iff)
done

```

13.5 *lfilter*: simple facts by coinduction

```

lemma lfilter-K-True:  $\text{lfilter } (\%x.\ True)\ l = l$ 
by (rule-tac  $l = l$  in llist-fun-equalityI, simp-all)

lemma lfilter-idem:  $\text{lfilter } p\ (\text{lfilter } p\ l) = \text{lfilter } p\ l$ 
apply (rule-tac  $l = l$  in llist-fun-equalityI, simp-all)
apply safe

```

Cases: $p\ x$ is true or false

```

apply (rule lfilter-cases [THEN disjE])
  apply (erule ssubst, auto)
done

```

13.6 Numerous lemmas required to prove *lfilter-conj*

lemma *findRel-conj-lemma* [rule-format]:

$(l, l') \in \text{findRel } q$
 $\implies l' = LCons\ x\ l'' \dashrightarrow p\ x \dashrightarrow (l, l') \in \text{findRel } (\%x. p\ x \ \&\ q\ x)$

by (*erule findRel.induct, auto*)

lemmas *findRel-conj* = *findRel-conj-lemma* [*OF - refl*]

lemma *findRel-not-conj-Domain* [rule-format]:

$(l, l') \in \text{findRel } (\%x. p\ x \ \&\ q\ x)$
 $\implies (l, LCons\ x\ l') \in \text{findRel } q \dashrightarrow \sim p\ x \dashrightarrow$
 $l' \in \text{Domain } (\text{findRel } (\%x. p\ x \ \&\ q\ x))$

by (*erule findRel.induct, auto*)

lemma *findRel-conj2* [rule-format]:

$(l, lxx) \in \text{findRel } q$
 $\implies lxx = LCons\ x\ lx \dashrightarrow (lx, lz) \in \text{findRel } (\%x. p\ x \ \&\ q\ x) \dashrightarrow \sim p\ x$
 $\dashrightarrow (l, lz) \in \text{findRel } (\%x. p\ x \ \&\ q\ x)$

by (*erule findRel.induct, auto*)

lemma *findRel-lfilter-Domain-conj* [rule-format]:

$(lx, ly) \in \text{findRel } p$
 $\implies \forall l. lx = \text{lfilter } q\ l \dashrightarrow l \in \text{Domain } (\text{findRel } (\%x. p\ x \ \&\ q\ x))$

apply (*erule findRel.induct*)

apply (*blast dest!: sym [THEN lfilter-eq-LCons] intro: findRel-conj, auto*)

apply (*drule sym [THEN lfilter-eq-LCons], auto*)

apply (*drule spec*)

apply (*drule refl [THEN rev-mp]*)

apply (*blast intro: findRel-conj2*)

done

lemma *findRel-conj-lfilter* [rule-format]:

$(l, l'') \in \text{findRel } (\%x. p\ x \ \&\ q\ x)$
 $\implies l'' = LCons\ y\ l' \dashrightarrow$
 $(\text{lfilter } q\ l, LCons\ y\ (\text{lfilter } q\ l')) \in \text{findRel } p$

by (*erule findRel.induct, auto*)

lemma *lfilter-conj-lemma*:

$(\text{lfilter } p\ (\text{lfilter } q\ l), \text{lfilter } (\%x. p\ x \ \&\ q\ x)\ l)$
 $\in \text{lListD-Fun } (\text{range } (\%u. (\text{lfilter } p\ (\text{lfilter } q\ u),$
 $\text{lfilter } (\%x. p\ x \ \&\ q\ x)\ u)))$

apply (*case-tac l \in Domain (findRel q)*)

apply (*subgoal-tac [2] l \sim: Domain (findRel (%x. p x \& q x))*)

prefer 3 apply (*blast intro: rev-subsetD [OF - Domain-findRel-mono]*)

There are no *qs* in *l*: both lists are *LNil*

apply (*simp-all add: Domain-findRel-iff, clarify*)

case $q\ x$
apply (*case-tac* $p\ x$)
apply (*simp-all add: findRel-conj [THEN findRel-imp-lfilter]*)
 case $q\ x$ and $\sim(p\ x)$
apply (*case-tac* $l' \in \text{Domain } (\text{findRel } (\%x. p\ x \ \&\ q\ x))$)
 subcase: there is no $p \ \&\ q$ in l' and therefore none in l
apply (*subgoal-tac* [2] $l \sim: \text{Domain } (\text{findRel } (\%x. p\ x \ \&\ q\ x))$)
prefer 3 **apply** (*blast intro: findRel-not-conj-Domain*)
apply (*subgoal-tac* [2] $l\text{filter } q\ l' \sim: \text{Domain } (\text{findRel } p)$)
prefer 3 **apply** (*blast intro: findRel-lfilter-Domain-conj*)
 ... and therefore too, no p in $l\text{filter } q\ l'$. Both results are *LNil*
apply (*simp-all add: Domain-findRel-iff, clarify*)
 subcase: there is a $p \ \&\ q$ in l' and therefore also one in l
apply (*subgoal-tac* ($l, L\text{Cons } xa\ l'a) \in \text{findRel } (\%x. p\ x \ \&\ q\ x)$)
prefer 2 **apply** (*blast intro: findRel-conj2*)
apply (*subgoal-tac* ($l\text{filter } q\ l', L\text{Cons } xa\ (l\text{filter } q\ l'a) \in \text{findRel } p$)
apply *simp*
apply (*blast intro: findRel-conj-lfilter*)
done

lemma *lfilter-conj*: $l\text{filter } p\ (l\text{filter } q\ l) = l\text{filter } (\%x. p\ x \ \&\ q\ x)\ l$
apply (*rule-tac* $l = l$ **in** *lfilter-fun-equalityI, simp-all*)
apply (*blast intro: lfilter-conj-lemma rev-subsetD [OF - llistD-Fun-mono]*)
done

13.7 Numerous lemmas required to prove $l\text{filter } p\ (l\text{map } f\ l) = l\text{map } f\ (l\text{filter } (\%x. p(f\ x))\ l)$

lemma *findRel-lmap-Domain*:
 $(l, l') \in \text{findRel } (\%x. p\ (f\ x)) \implies l\text{map } f\ l \in \text{Domain } (\text{findRel } p)$
by (*erule findRel.induct, auto*)

lemma *lmap-eq-LCons* [*rule-format*]: $l\text{map } f\ l = L\text{Cons } x\ l' \dashv\dashv$
 $(\exists y\ l''. x = f\ y \ \&\ l' = l\text{map } f\ l'' \ \&\ l = L\text{Cons } y\ l'')$
apply (*subst lmap-def [THEN def-llist-corec]*)
apply (*rule-tac* $l = l$ **in** *llistE, auto*)
done

lemma *lmap-LCons-findRel-lemma* [*rule-format*]:
 $(lx, ly) \in \text{findRel } p$
 $\implies \forall l. l\text{map } f\ l = lx \dashv\dashv ly = L\text{Cons } x\ l' \dashv\dashv$
 $(\exists y\ l''. x = f\ y \ \&\ l' = l\text{map } f\ l'' \ \&$

```

      (l, LCons y l'') ∈ findRel(%x. p(f x)))
apply (erule findRel.induct, simp-all)
apply (blast dest!: lmap-eq-LCons)+
done

lemmas lmap-LCons-findRel = lmap-LCons-findRel-lemma [OF - refl refl]

lemma lfilter-lmap: lfilter p (lmap f l) = lmap f (lfilter (p o f) l)
apply (rule-tac l = l in llist-fun-equalityI, simp-all)
apply safe
apply (case-tac lmap f l ∈ Domain (findRel p))
  apply (simp add: Domain-findRel-iff, clarify)
  apply (frule lmap-LCons-findRel, force)
apply (subgoal-tac l ~: Domain (findRel (%x. p (f x))), simp)
apply (blast intro: findRel-lmap-Domain)
done

end

```

14 Mutual Induction via Iterated Inductive Definitions

theory Com **imports** Main **begin**

typedecl loc

types state = loc => nat

datatype

```

  exp = N nat
    | X loc
    | Op nat => nat => nat exp exp
    | valOf com exp      (VALOF - RESULTIS - 60)

```

and

```

  com = SKIP
    | Assign loc exp      (infixl := 60)
    | Semi com com        (-;;- [60, 60] 60)
    | Cond exp com com    (IF - THEN - ELSE - 60)
    | While exp com       (WHILE - DO - 60)

```

14.1 Commands

Execution of commands

abbreviation (input)

```

  generic-rel (-/ -|[-]-> - [50,0,50] 50) where
    esig -|[-]eval-> ns == (esig,ns) ∈ eval

```

Command execution. Natural numbers represent Booleans: 0=True, 1=False

inductive-set

```

exec :: ((exp*state) * (nat*state)) set => ((com*state)*state)set
and exec-rel :: com * state => ((exp*state) * (nat*state)) set => state => bool
  (-/ -[-]-> - [50,0,50] 50)
for eval :: ((exp*state) * (nat*state)) set
where
  csig -[eval]-> s == (csig,s) ∈ exec eval

| Skip:    (SKIP,s) -[eval]-> s

| Assign:  (e,s) -[eval]-> (v,s') ==> (x := e, s) -[eval]-> s'(x:=v)

| Semi:    [| (c0,s) -[eval]-> s2; (c1,s2) -[eval]-> s1 |]
           ==> (c0 ;; c1, s) -[eval]-> s1

| IfTrue:  [| (e,s) -[eval]-> (0,s'); (c0,s') -[eval]-> s1 |]
           ==> (IF e THEN c0 ELSE c1, s) -[eval]-> s1

| IfFalse: [| (e,s) -[eval]-> (Suc 0, s'); (c1,s') -[eval]-> s1 |]
           ==> (IF e THEN c0 ELSE c1, s) -[eval]-> s1

| WhileFalse: (e,s) -[eval]-> (Suc 0, s1)
              ==> (WHILE e DO c, s) -[eval]-> s1

| WhileTrue:  [| (e,s) -[eval]-> (0,s1);
                (c,s1) -[eval]-> s2; (WHILE e DO c, s2) -[eval]-> s3 |]
              ==> (WHILE e DO c, s) -[eval]-> s3

```

declare *exec.intros* [intro]

inductive-cases

```

[elim!]: (SKIP,s) -[eval]-> t
and [elim!]: (x:=a,s) -[eval]-> t
and [elim!]: (c1;;c2, s) -[eval]-> t
and [elim!]: (IF e THEN c1 ELSE c2, s) -[eval]-> t
and exec-WHILE-case: (WHILE b DO c,s) -[eval]-> t

```

Justifies using "exec" in the inductive definition of "eval"

lemma *exec-mono*: $A \leq B \implies \text{exec}(A) \leq \text{exec}(B)$

apply (rule subsetI)

apply (simp add: split-paired-all)

apply (erule exec.induct)

apply blast+

done

lemma [pred-set-conv]:

$((\lambda x x' y y'. ((x, x'), (y, y')) \in R) \leq (\lambda x x' y y'. ((x, x'), (y, y')) \in S)) = (R \leq S)$

```

by (auto simp add: le-fun-def le-bool-def)

lemma [pred-set-conv]:
  (( $\lambda x x' y. ((x, x'), y) \in R$ ) <= ( $\lambda x x' y. ((x, x'), y) \in S$ )) = ( $R$  <=  $S$ )
by (auto simp add: le-fun-def le-bool-def)

declare [[unify-trace-bound = 30, unify-search-bound = 60]]

```

Command execution is functional (deterministic) provided evaluation is

```

theorem single-valued-exec: single-valued  $ev \implies$  single-valued(exec  $ev$ )
apply (simp add: single-valued-def)
apply (intro allI)
apply (rule impI)
apply (erule exec.induct)
apply (blast elim: exec-WHILE-case)+
done

```

14.2 Expressions

Evaluation of arithmetic expressions

```

inductive-set
  eval :: ((exp*state) * (nat*state)) set
  and eval-rel :: [exp*state, nat*state] => bool (infixl -|-> 50)
  where
    esig -|-> ns == (esig, ns) ∈ eval

  | N [intro!]: (N(n),s) -|-> (n,s)

  | X [intro!]: (X(x),s) -|-> (s(x),s)

  | Op [intro]: [| (e0,s) -|-> (n0,s0); (e1,s0) -|-> (n1,s1) |]
    ==> (Op f e0 e1, s) -|-> (f n0 n1, s1)

  | valOf [intro]: [| (c,s) -[eval]-> s0; (e,s0) -|-> (n,s1) |]
    ==> (VALOF c RESULTIS e, s) -|-> (n, s1)

monos exec-mono

```

inductive-cases

```

  [elim!]: (N(n),sigma) -|-> (n',s')
  and [elim!]: (X(x),sigma) -|-> (n,s')
  and [elim!]: (Op f a1 a2,sigma) -|-> (n,s')
  and [elim!]: (VALOF c RESULTIS e, s) -|-> (n, s1)

```

```

lemma var-assign-eval [intro!]: (X x, s(x:=n)) -|-> (n, s(x:=n))
by (rule fun-upd-same [THEN subst], fast)

```

Make the induction rule look nicer – though *eta-contract* makes the new version look worse than it is...

lemma *split-lemma*:

```
{((e,s),(n,s')). P e s n s'} = Collect (split (%v. split (split P v)))
by auto
```

New induction rule. Note the form of the VALOF induction hypothesis

lemma *eval-induct*

```
[case-names N X Op valOf, consumes 1, induct set: eval]:
[[ (e,s) -|> (n,s');
  !!n s. P (N n) s n s;
  !!s x. P (X x) s (s x) s;
  !!e0 e1 f n0 n1 s s0 s1.
    [[ (e0,s) -|> (n0,s0); P e0 s n0 s0;
      (e1,s0) -|> (n1,s1); P e1 s0 n1 s1
    ]] ==> P (Op f e0 e1) s (f n0 n1) s1;
  !!c e n s s0 s1.
    [[ (c,s) -[eval Int {((e,s),(n,s')). P e s n s'}]-> s0;
      (c,s) -[eval]-> s0;
      (e,s0) -|> (n,s1); P e s0 n s1
    ]] ==> P (VALOF c RESULTIS e) s n s1
  ]] ==> P e s n s'
apply (induct set: eval)
apply blast
apply blast
apply blast
apply (frule Int-lower1 [THEN exec-mono, THEN subsetD])
apply (auto simp add: split-lemma)
done
```

Lemma for *Function-eval*. The major premise is that (c,s) executes to $s1$ using *eval* restricted to its functional part. Note that the execution $(c,s) -[eval]-> s2$ can use unrestricted *eval*! The reason is that the execution $(c,s) -[eval Int \{...\}]-> s1$ assures us that execution is functional on the argument (c,s) .

lemma *com-Unique*:

```
(c,s) -[eval Int {((e,s),(n,t)). ∀ nt'. (e,s) -|> nt' --> (n,t)=nt'}]-> s1
==> ∀ s2. (c,s) -[eval]-> s2 --> s2=s1
apply (induct set: exec)
  apply simp-all
  apply blast
  apply force
  apply blast
  apply blast
  apply blast
apply (blast elim: exec-WHILE-case)
apply (erule-tac V = (?c,s2) -[?ev]-> s3 in thin-rl)
apply clarify
```


apply (*erule exec-WHILE-case*, *blast+*)
done

Expression evaluation is functional, or deterministic

theorem *single-valued-eval: single-valued eval*
apply (*unfold single-valued-def*)
apply (*intro allI*, *rule impI*)
apply (*simp (no-asm-simp) only: split-tupled-all*)
apply (*erule eval-induct*)
apply (*drule-tac* [4] *com-Unique*)
apply (*simp-all (no-asm-use)*)
apply *blast+*
done

lemma *eval-N-E* [*dest!*]: $(N\ n, s) \dashv\!\rightarrow (v, s') \implies (v = n \ \& \ s' = s)$
by (*induct e == N n s v s' set: eval*) *simp-all*

This theorem says that "WHILE TRUE DO c" cannot terminate

lemma *while-true-E*:
 $(c', s) \dashv\!\rightarrow t \implies c' = \text{WHILE } (N\ 0) \text{ DO } c \implies \text{False}$
by (*induct set: exec*) *auto*

14.3 Equivalence of IF e THEN c;;(WHILE e DO c) ELSE SKIP and WHILE e DO c

lemma *while-if1*:
 $(c', s) \dashv\!\rightarrow t \implies c' = \text{WHILE } e \text{ DO } c \implies$
 $(\text{IF } e \text{ THEN } c;;c' \text{ ELSE SKIP}, s) \dashv\!\rightarrow t$
by (*induct set: exec*) *auto*

lemma *while-if2*:
 $(c', s) \dashv\!\rightarrow t \implies c' = \text{IF } e \text{ THEN } c;;(\text{WHILE } e \text{ DO } c) \text{ ELSE SKIP} \implies$
 $(\text{WHILE } e \text{ DO } c, s) \dashv\!\rightarrow t$
by (*induct set: exec*) *auto*

theorem *while-if*:
 $((\text{IF } e \text{ THEN } c;;(\text{WHILE } e \text{ DO } c) \text{ ELSE SKIP}, s) \dashv\!\rightarrow t) =$
 $((\text{WHILE } e \text{ DO } c, s) \dashv\!\rightarrow t)$
by (*blast intro: while-if1 while-if2*)

14.4 Equivalence of (IF e THEN c1 ELSE c2);;c and IF e THEN (c1;;c) ELSE (c2;;c)

lemma *if-semi1*:
 $(c', s) \dashv\!\rightarrow t \implies c' = (\text{IF } e \text{ THEN } c1 \text{ ELSE } c2);;c \implies$

$(IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c),\ s) \text{ --}[eval]\text{--> } t$
by (*induct set: exec*) *auto*

lemma *if-semi2*:

$(c',s) \text{ --}[eval]\text{--> } t$
 $\implies c' = IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c) \implies$
 $((IF\ e\ THEN\ c1\ ELSE\ c2);;c,\ s) \text{ --}[eval]\text{--> } t$
by (*induct set: exec*) *auto*

theorem *if-semi*: $((IF\ e\ THEN\ c1\ ELSE\ c2);;c,\ s) \text{ --}[eval]\text{--> } t =$
 $((IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c),\ s) \text{ --}[eval]\text{--> } t)$
by (*blast intro: if-semi1 if-semi2*)

14.5 Equivalence of VALOF c1 RESULTIS (VALOF c2 RESULTIS e) and VALOF c1;;c2 RESULTIS e

lemma *valof-valof1*:

$(e',s) \text{ --}|\text{--> } (v,s')$
 $\implies e' = VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e) \implies$
 $(VALOF\ c1;;c2\ RESULTIS\ e,\ s) \text{ --}|\text{--> } (v,s')$
by (*induct set: eval*) *auto*

lemma *valof-valof2*:

$(e',s) \text{ --}|\text{--> } (v,s')$
 $\implies e' = VALOF\ c1;;c2\ RESULTIS\ e \implies$
 $(VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e),\ s) \text{ --}|\text{--> } (v,s')$
by (*induct set: eval*) *auto*

theorem *valof-valof*:

$((VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e),\ s) \text{ --}|\text{--> } (v,s')) =$
 $((VALOF\ c1;;c2\ RESULTIS\ e,\ s) \text{ --}|\text{--> } (v,s'))$
by (*blast intro: valof-valof1 valof-valof2*)

14.6 Equivalence of VALOF SKIP RESULTIS e and e

lemma *valof-skip1*:

$(e',s) \text{ --}|\text{--> } (v,s')$
 $\implies e' = VALOF\ SKIP\ RESULTIS\ e \implies$
 $(e,\ s) \text{ --}|\text{--> } (v,s')$
by (*induct set: eval*) *auto*

lemma *valof-skip2*:

$(e,s) \text{ --}|\text{--> } (v,s') \implies (VALOF\ SKIP\ RESULTIS\ e,\ s) \text{ --}|\text{--> } (v,s')$
by *blast*

theorem *valof-skip*:

$((VALOF\ SKIP\ RESULTIS\ e,\ s) \text{ --}|\text{--> } (v,s')) = ((e,\ s) \text{ --}|\text{--> } (v,s'))$
by (*blast intro: valof-skip1 valof-skip2*)

14.7 Equivalence of VALOF $x:=e$ RESULTIS x and e

lemma *valof-assign1*:

$(e', s) \dashv\vdash (v, s'')$

$\implies e' = \text{VALOF } x:=e \text{ RESULTIS } X x \implies$

$(\exists s'. (e, s) \dashv\vdash (v, s') \ \& \ (s'' = s'(x:=v)))$

by (*induct set: eval*) (*simp-all del: fun-upd-apply, clarify, auto*)

lemma *valof-assign2*:

$(e, s) \dashv\vdash (v, s') \implies (\text{VALOF } x:=e \text{ RESULTIS } X x, s) \dashv\vdash (v, s'(x:=v))$

by *blast*

end