

Examples of Inductive and Coinductive Definitions in HOL

Stefan Berghofer
Tobias Nipkow
Lawrence C Paulson
Markus Wenzel

December 12, 2021

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	2
1.1	Variations on statement structure	2
1.1.1	Local facts and parameters	2
1.1.2	Local definitions	2
1.1.3	Simple simultaneous goals	3
1.1.4	Compound simultaneous goals	3
1.2	Multiple rules	3
1.3	Inductive predicates	5
2	Nested datatypes	6
2.1	Terms and substitution	6
2.2	Alternative induction	6
3	Defining an Initial Algebra by Quotienting a Free Algebra	7
3.1	Defining the Free Algebra	7
3.2	Some Functions on the Free Algebra	8
3.2.1	The Set of Nonces	8
3.2.2	The Left Projection	8
3.2.3	The Right Projection	8
3.2.4	The Discriminator for Constructors	9
3.3	The Initial Algebra: A Quotiented Message Type	9
3.3.1	Characteristic Equations for the Abstract Constructors	10

3.4	The Abstract Function to Return the Set of Nonces	10
3.5	The Abstract Function to Return the Left Part	11
3.6	The Abstract Function to Return the Right Part	11
3.7	Injectivity Properties of Some Constructors	12
3.8	The Abstract Discriminator	13
4	Quotienting a Free Algebra Involving Nested Recursion	13
4.1	Defining the Free Algebra	14
4.2	Some Functions on the Free Algebra	15
4.2.1	The Set of Variables	15
4.2.2	Functions for Freeness	15
4.3	The Initial Algebra: A Quotiented Message Type	16
4.4	Every list of abstract expressions can be expressed in terms of a list of concrete expressions	17
4.4.1	Characteristic Equations for the Abstract Constructors	17
4.5	The Abstract Function to Return the Set of Variables	18
4.6	Injectivity Properties of Some Constructors	18
4.7	Injectivity of <i>FnCall</i>	19
4.8	The Abstract Discriminator	20
5	Terms over a given alphabet	20
6	Extended List Theory (old)	24
7	Arithmetic and boolean expressions	31
8	Infinitely branching trees	32
8.1	The Brouwer ordinals, as in ZF/Induct/Brouwer.thy.	33
8.2	A WF Ordering for The Brouwer ordinals (Michael Compton)	33
9	Ordinals	34
10	Sigma algebras	35
11	Combinatory Logic example: the Church-Rosser Theorem	36
11.1	Definitions	36
11.2	Reflexive/Transitive closure preserves Church-Rosser property	37
11.3	Non-contraction results	37
11.4	Results about Parallel Contraction	38
11.5	Basic properties of parallel contraction	38
11.6	Equivalence of $p \rightarrow q$ and $p \Rightarrow q$	39

12	Meta-theory of propositional logic	39
12.1	The datatype of propositions	39
12.2	The proof system	39
12.3	The semantics	40
12.3.1	Semantics of propositional logic.	40
12.3.2	Logical consequence	40
12.4	Proof theory of propositional logic	40
12.4.1	Weakening, left and right	40
12.4.2	The deduction theorem	41
12.4.3	The cut rule	41
12.4.4	Soundness of the rules wrt truth-table semantics	41
12.5	Completeness	41
12.5.1	Towards the completeness proof	41
12.6	Completeness – lemmas for reducing the set of assumptions	42
12.6.1	Completeness theorem	42
13	Mutual Induction via Iterated Inductive Definitions	43
13.1	Commands	43
13.2	Expressions	45
13.3	Equivalence of IF e THEN c;;(WHILE e DO c) ELSE SKIP and WHILE e DO c	46
13.4	Equivalence of (IF e THEN c1 ELSE c2);;c and IF e THEN (c1;;c) ELSE (c2;;c)	47
13.5	Equivalence of VALOF c1 RESULTIS (VALOF c2 RESULTIS e) and VALOF c1;;c2 RESULTIS e	47
13.6	Equivalence of VALOF SKIP RESULTIS e and e	47
13.7	Equivalence of VALOF x:=e RESULTIS x and e	48

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: $\llbracket P\ 0; \bigwedge nat. P\ nat \implies P\ (Suc\ nat) \rrbracket \implies P\ nat$, 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 \longrightarrow 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  $P\ n\ x$  <proof>
```

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  $P\ (a\ x)$  <proof>
```

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)
```

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

```
lemma  
  fixes  $n :: nat$   
  shows  $A\ n \implies P\ n$   
  and  $B\ n \implies Q\ n$   
(proof)
```

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)
```

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:

$$\begin{aligned} & \llbracket \bigwedge x. P1 (Foo1 x); \bigwedge x. P2 x \implies P1 (Foo2 x); \bigwedge x. P2 (Bar1 x); \\ & \bigwedge x. P3 x \implies P2 (Bar2 x); \bigwedge x. P1 x \implies P3 (Bazar x) \rrbracket \\ & \implies P1 \text{ foo} \\ & \llbracket \bigwedge x. P1 (Foo1 x); \bigwedge x. P2 x \implies P1 (Foo2 x); \bigwedge x. P2 (Bar1 x); \\ & \bigwedge x. P3 x \implies P2 (Bar2 x); \bigwedge x. P1 x \implies P3 (Bazar x) \rrbracket \\ & \implies P2 \text{ bar} \\ & \llbracket \bigwedge x. P1 (Foo1 x); \bigwedge x. P2 x \implies P1 (Foo2 x); \bigwedge x. P2 (Bar1 x); \\ & \bigwedge x. P3 x \implies P2 (Bar2 x); \bigwedge x. P1 x \implies P3 (Bazar x) \rrbracket \\ & \implies P3 \text{ bazar} \end{aligned}$$

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⟩
```

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 x foo  $\implies$  P x foo
and
  B1 y bar  $\implies$  Q1 y bar
  B2 y bar  $\implies$  Q2 y bar
and
  C1 z bazar  $\implies$  R1 z bazar
  C2 z bazar  $\implies$  R2 z bazar
  C3 z bazar  $\implies$  R3 z bazar
⟨proof⟩
```

1.3 Inductive predicates

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

```
inductive Even :: nat ⇒ bool where  
  zero: Even 0  
| double: Even (2 * n) if Even n for n
```

```
lemma  
  assumes Even n  
  shows P n  
  ⟨proof⟩
```

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

```
lemma Even n ⇒ P n  
  ⟨proof⟩
```

Simultaneous goals do not introduce anything new.

```
lemma  
  assumes Even n  
  shows P1 n and P2 n  
  ⟨proof⟩
```

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: Odd (Suc n) if Evn n for n  
| succ-Odd: Evn (Suc n) if Odd n for n
```

```
lemma  
  Evn n ⇒ P1 n  
  Evn n ⇒ P2 n  
  Evn n ⇒ P3 n  
and  
  Odd n ⇒ Q1 n  
  Odd n ⇒ Q2 n  
  ⟨proof⟩
```

Cases and hypotheses in each case can be named explicitly.

```
inductive star :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool for r  
where  
  refl: star r x x for x  
| step: star r x z if r x y and star r y z for x y z
```

Underscores are replaced by the default name hyps:

```
lemmas star-induct = star.induct [case-names base step[r - IH]]
```

```
lemma star r x y  $\implies$  star r y z  $\implies$  star r x z  
<proof>
```

```
end
```

2 Nested datatypes

```
theory Nested-Datatype  
imports Main  
begin
```

2.1 Terms and substitution

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

```
primrec subst-term :: ('a  $\Rightarrow$  ('a, 'b) term)  $\Rightarrow$  ('a, 'b) term  $\Rightarrow$  ('a, 'b) term  
  and subst-term-list :: ('a  $\Rightarrow$  ('a, 'b) term)  $\Rightarrow$  ('a, 'b) term list  $\Rightarrow$  ('a, 'b) term  
  list
```

```
where
```

```
  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
```

```
lemmas subst-simps = subst-term.simps subst-term-list.simps
```

A simple lemma about composition of substitutions.

```
lemma
```

```
  subst-term (subst-term f1  $\circ$  f2) t =  
    subst-term f1 (subst-term f2 t)  
  and  
  subst-term-list (subst-term f1  $\circ$  f2) ts =  
    subst-term-list f1 (subst-term-list f2 ts)  
  <proof>
```

```
lemma subst-term (subst-term f1  $\circ$  f2) t = subst-term f1 (subst-term f2 t)  
<proof>
```

2.2 Alternative induction

```
lemma subst-term (subst-term f1  $\circ$  f2) t = subst-term f1 (subst-term f2 t)  
<proof>
```

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*: $X \sim X$

<proof>

theorem *equiv-msgrel*: *equiv UNIV msgrel*

<proof>

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.

primrec *freenonces* :: *freemsg* \Rightarrow *nat set* **where**
 freenonces (*NONCE* *N*) = {*N*}
| *freenonces* (*MPAIR* *X* *Y*) = *freenonces* *X* \cup *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*: $U \sim V \Longrightarrow \text{freenonces } U = \text{freenonces } V$
 <proof>

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.

primrec *freeleft* :: *freemsg* \Rightarrow *freemsg* **where**
 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-eq-freeleft*:
 $U \sim V \Longrightarrow \text{freeleft } U \sim \text{freeleft } V$
 <proof>

3.2.3 The Right Projection

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

primrec *freeright* :: *freemsg* \Rightarrow *freemsg* **where**
 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-eqv-freeright*:
 $U \sim V \implies \text{freeright } U \sim \text{freeright } V$
 ⟨proof⟩

3.2.4 The Discriminator for Constructors

A function to distinguish nonces, mpairs and encryptions

primrec *freediscrim* :: *freemsg* \Rightarrow *int* **where**
 $\text{freediscrim } (\text{NONCE } N) = 0$
 $|\text{ freediscrim } (\text{MPAIR } X Y) = 1$
 $|\text{ freediscrim } (\text{CRYPT } K X) = \text{freediscrim } X + 2$
 $|\text{ freediscrim } (\text{DECRYPT } K X) = \text{freediscrim } X - 2$

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

theorem *msgrel-imp-eq-freediscrim*:
 $U \sim V \implies \text{freediscrim } U = \text{freediscrim } V$
 ⟨proof⟩

3.3 The Initial Algebra: A Quotiented Message Type

definition $\text{Msg} = \text{UNIV} // \text{msgrel}$

typedef *msg* = *Msg*
morphisms *Rep-Msg Abs-Msg*
 ⟨proof⟩

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*]: $\text{msgrel}\{U\} \in \text{Msg}$
(*proof*)

lemma *inj-on-Abs-Msg*: *inj-on Abs-Msg Msg*
(*proof*)

Reduces equality on abstractions to equality on representatives

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

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

3.3.1 Characteristic Equations for the Abstract Constructors

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

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

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

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

lemma *eq-Abs-Msg* [*case-names Abs-Msg, cases type: msg*]:
 $(!!U. z = \text{Abs-Msg}(\text{msgrel}\{U\}) \implies P) \implies P$
(*proof*)

Establishing these two equations is the point of the whole exercise

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

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

3.4 The Abstract Function to Return the Set of Nonces

definition

nonces :: $\text{msg} \Rightarrow \text{nat set}$ **where**
 $\text{nonces } X = (\bigcup U \in \text{Rep-Msg } X. \text{freenonces } U)$

lemma *nonces-congruent*: *freenonces respects msgrel*

$\langle proof \rangle$

Now prove the four equations for *nonces*

lemma *nonces-Nonce* [simp]: $nonces (Nonce N) = \{N\}$
 $\langle proof \rangle$

lemma *nonces-MPair* [simp]: $nonces (MPair X Y) = nonces X \cup nonces Y$
 $\langle proof \rangle$

lemma *nonces-Crypt* [simp]: $nonces (Crypt K X) = nonces X$
 $\langle proof \rangle$

lemma *nonces-Decrypt* [simp]: $nonces (Decrypt K X) = nonces X$
 $\langle proof \rangle$

3.5 The Abstract Function to Return the Left Part

definition

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

lemma *left-congruent*: $(\lambda U. msgrel \{freeleft U\})$ respects *msgrel*
 $\langle proof \rangle$

Now prove the four equations for *left*

lemma *left-Nonce* [simp]: $left (Nonce N) = Nonce N$
 $\langle proof \rangle$

lemma *left-MPair* [simp]: $left (MPair X Y) = X$
 $\langle proof \rangle$

lemma *left-Crypt* [simp]: $left (Crypt K X) = left X$
 $\langle proof \rangle$

lemma *left-Decrypt* [simp]: $left (Decrypt K X) = left X$
 $\langle proof \rangle$

3.6 The Abstract Function to Return the Right Part

definition

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

lemma *right-congruent*: $(\lambda U. msgrel \{freeright U\})$ respects *msgrel*
 $\langle proof \rangle$

Now prove the four equations for *right*

lemma *right-Nonce* [simp]: $right (Nonce N) = Nonce N$
 $\langle proof \rangle$

lemma *right-MPair* [simp]: $\text{right } (\text{MPair } X \ Y) = Y$
<proof>

lemma *right-Crypt* [simp]: $\text{right } (\text{Crypt } K \ X) = \text{right } X$
<proof>

lemma *right-Decrypt* [simp]: $\text{right } (\text{Decrypt } K \ X) = \text{right } X$
<proof>

3.7 Injectivity Properties of Some Constructors

lemma *NONCE-imp-eq*: $\text{NONCE } m \sim \text{NONCE } n \implies m = n$
<proof>

Can also be proved using the function *nonces*

lemma *Nonce-Nonce-eq* [iff]: $(\text{Nonce } m = \text{Nonce } n) = (m = n)$
<proof>

lemma *MPAIR-imp-eqv-left*: $\text{MPAIR } X \ Y \sim \text{MPAIR } X' \ Y' \implies X \sim X'$
<proof>

lemma *MPair-imp-eq-left*:
 assumes *eq*: $\text{MPair } X \ Y = \text{MPair } X' \ Y'$ **shows** $X = X'$
<proof>

lemma *MPAIR-imp-eqv-right*: $\text{MPAIR } X \ Y \sim \text{MPAIR } X' \ Y' \implies Y \sim Y'$
<proof>

lemma *MPair-imp-eq-right*: $\text{MPair } X \ Y = \text{MPair } X' \ Y' \implies Y = Y'$
<proof>

theorem *MPair-MPair-eq* [iff]: $(\text{MPair } X \ Y = \text{MPair } X' \ Y') = (X=X' \ \& \ Y=Y')$
<proof>

lemma *NONCE-neq-MPAIR*: $\text{NONCE } m \sim \text{MPAIR } X \ Y \implies \text{False}$
<proof>

theorem *Nonce-neq-MPair* [iff]: $\text{Nonce } N \neq \text{MPair } X \ Y$
<proof>

Example suggested by a referee

theorem *Crypt-Nonce-neq-Nonce*: $\text{Crypt } K \ (\text{Nonce } M) \neq \text{Nonce } N$
<proof>

...and many similar results

theorem *Crypt2-Nonce-neq-Nonce*: $\text{Crypt } K \ (\text{Crypt } K' \ (\text{Nonce } M)) \neq \text{Nonce } N$
<proof>

theorem *Crypt-Crypt-eq* [iff]: $(\text{Crypt } K \ X = \text{Crypt } K \ X') = (X=X')$
 ⟨proof⟩

theorem *Decrypt-Decrypt-eq* [iff]: $(\text{Decrypt } K \ X = \text{Decrypt } K \ X') = (X=X')$
 ⟨proof⟩

lemma *msg-induct* [case-names Nonce MPair Crypt Decrypt, cases type: msg]:
 assumes $N: \bigwedge N. P (\text{Nonce } N)$
 and $M: \bigwedge X \ Y. \llbracket P \ X; P \ Y \rrbracket \implies P (\text{MPair } X \ Y)$
 and $C: \bigwedge K \ X. P \ X \implies P (\text{Crypt } K \ X)$
 and $D: \bigwedge K \ X. P \ X \implies P (\text{Decrypt } K \ X)$
 shows $P \ \text{msg}$
 ⟨proof⟩

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

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

lemma *discrim-congruent*: $(\lambda U. \{\text{freediscrim } U\})$ respects msgrel
 ⟨proof⟩

Now prove the four equations for *discrim*

lemma *discrim-Nonce* [simp]: $\text{discrim } (\text{Nonce } N) = 0$
 ⟨proof⟩

lemma *discrim-MPair* [simp]: $\text{discrim } (\text{MPair } X \ Y) = 1$
 ⟨proof⟩

lemma *discrim-Crypt* [simp]: $\text{discrim } (\text{Crypt } K \ X) = \text{discrim } X + 2$
 ⟨proof⟩

lemma *discrim-Decrypt* [simp]: $\text{discrim } (\text{Decrypt } K \ X) = \text{discrim } X - 2$
 ⟨proof⟩

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

```

datatype-compat *freeExp*

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 transitivity.

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: [[X ~ X'; Y ~ Y']] ==> PLUS X Y ~ PLUS X' Y'
  | FNCALL: (Xs, Xs') ∈ listrel exprel ==> FNCALL F Xs ~ FNCALL F Xs'
  | SYM: X ~ Y ==> Y ~ X
  | TRANS: [[X ~ Y; Y ~ Z]] ==> X ~ Z
monos listrel-mono

```

Proving that it is an equivalence relation

lemma *exprel-refl*: X ~ X
and *list-exprel-refl*: (Xs, Xs) ∈ *listrel(exprel)*
 ⟨*proof*⟩

theorem *equiv-exprel*: *equiv UNIV exprel*
 ⟨*proof*⟩

theorem *equiv-list-exprel*: *equiv UNIV (listrel exprel)*
 ⟨*proof*⟩

lemma *FNCALL-Nil*: FNCALL F [] ~ FNCALL F []
 ⟨*proof*⟩

lemma *FNCALL-Cons*:
 [[X ~ X'; (Xs, Xs') ∈ *listrel(exprel)*]]
 ==> FNCALL F (X#Xs) ~ FNCALL F (X'#Xs')
 ⟨*proof*⟩

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.

```
primrec freevars :: freeExp  $\Rightarrow$  nat set
  and freevars-list :: freeExp list  $\Rightarrow$  nat set where
    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*: $U \sim V \implies \text{freevars } U = \text{freevars } V$
(*proof*)

4.2.2 Functions for Freeness

A discriminator function to distinguish vars, sums and function calls

```
primrec freediscrim :: freeExp  $\Rightarrow$  int where
  freediscrim (VAR N) = 0
  | freediscrim (PLUS X Y) = 1
  | freediscrim (FNCALL F Xs) = 2
```

theorem *exprel-imp-eq-freediscrim*:
 $U \sim V \implies \text{freediscrim } U = \text{freediscrim } V$
(*proof*)

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

```
primrec freefun :: freeExp  $\Rightarrow$  nat where
  freefun (VAR N) = 0
  | freefun (PLUS X Y) = 0
  | freefun (FNCALL F Xs) = F
```

theorem *exprel-imp-eq-freefun*:
 $U \sim V \implies \text{freefun } U = \text{freefun } V$
(*proof*)

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

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

theorem *exprel-imp-eqv-freeargs*:
assumes $U \sim V$
shows (*freeargs* *U*, *freeargs* *V*) \in *listrel exprel*
<proof>

4.3 The Initial Algebra: A Quotiented Message Type

definition *Exp* = *UNIV* // *exprel*

typedef *exp* = *Exp*
morphisms *Rep-Exp Abs-Exp*
<proof>

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 ($\bigcup U \in \text{Rep-Exp } X. \bigcup V \in \text{Rep-Exp } Y. \text{exprel} \text{ “ } \{PLUS\ U\ V\}$)

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

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*
<proof>

lemma *inj-on-Abs-Exp*: *inj-on Abs-Exp Exp*
<proof>

Reduces equality on abstractions to equality on representatives

declare *inj-on-Abs-Exp* [*THEN inj-on-eq-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*]:
 $(!!U. z = \text{Abs-Exp}(\text{exprel}\{\{U\}\}) \implies P) \implies P$
<proof>

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*. *Abs-Exp*(*exprel*\{\{*U*\}\})) *Xs*

lemma *Abs-ExpList-Nil* [*simp*]: *Abs-ExpList* [] == []
<proof>

lemma *Abs-ExpList-Cons* [*simp*]:
 $\text{Abs-ExpList } (X\#Xs) == \text{Abs-Exp } (\text{exprel}\{\{X\}\}) \# \text{Abs-ExpList } Xs$
<proof>

lemma *ExpList-rep*: $\exists Us. z = \text{Abs-ExpList } Us$
<proof>

lemma *eq-Abs-ExpList* [*case-names Abs-ExpList*]:
 $(!!Us. z = \text{Abs-ExpList } Us \implies P) \implies P$
<proof>

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>

It is not clear what to do with *FnCall*: its 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*: $\text{FnCall } F [] = \text{Abs-Exp } (\text{exprel}\{\{FNCALL\} F []\})$
<proof>

lemma *FnCall-respects*:
 $(\lambda Us. \text{exprel } \{\{FNCALL\} F Us\}) \text{ respects } (\text{listrel exprel})$
<proof>

lemma *FnCall-sing*:

$FnCall\ F\ [Abs-Exp(\text{exprel}\ \{\ U\})] = Abs-Exp(\text{exprel}\ \{FNCALL\ F\ [U]\})$
 ⟨proof⟩

lemma *listset-Rep-Exp-Abs-Exp*:

$listset(\text{map}\ Rep-Exp\ (Abs-ExpList\ Us)) = listrel\ \text{exprel}\ \{\ Us\}$
 ⟨proof⟩

lemma *FnCall*:

$FnCall\ F\ (Abs-ExpList\ Us) = Abs-Exp(\text{exprel}\ \{FNCALL\ F\ Us\})$
 ⟨proof⟩

Establishing this equation is the point of the whole exercise

theorem *Plus-assoc*: $Plus\ X\ (Plus\ Y\ Z) = Plus\ (Plus\ X\ Y)\ Z$
 ⟨proof⟩

4.5 The Abstract Function to Return the Set of Variables

definition

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

lemma *vars-respects*: *freevars respects exprel*
 ⟨proof⟩

The extension of the function *vars* to lists

primrec *vars-list* :: $exp\ list \Rightarrow nat\ set$ **where**

$vars-list\ [] = \{\}$
 $| vars-list(E\ #\ Es) = vars\ E \cup vars-list\ Es$

Now prove the three equations for *vars*

lemma *vars-Variable* [*simp*]: $vars\ (Var\ N) = \{N\}$
 ⟨proof⟩

lemma *vars-Plus* [*simp*]: $vars\ (Plus\ X\ Y) = vars\ X \cup vars\ Y$
 ⟨proof⟩

lemma *vars-FnCall* [*simp*]: $vars\ (FnCall\ F\ Xs) = vars-list\ Xs$
 ⟨proof⟩

lemma *vars-FnCall-Nil*: $vars\ (FnCall\ F\ Nil) = \{\}$
 ⟨proof⟩

lemma *vars-FnCall-Cons*: $vars\ (FnCall\ F\ (X\ #\ Xs)) = vars\ X \cup vars-list\ Xs$
 ⟨proof⟩

4.6 Injectivity Properties of Some Constructors

lemma *VAR-imp-eq*: $VAR\ m \sim VAR\ n \implies m = n$
 ⟨proof⟩

Can also be proved using the function *vars*

lemma *Var-Var-eq* [iff]: $(\text{Var } m = \text{Var } n) = (m = n)$
<proof>

lemma *VAR-neqv-PLUS*: $\text{VAR } m \sim \text{PLUS } X Y \implies \text{False}$
<proof>

theorem *Var-neq-Plus* [iff]: $\text{Var } N \neq \text{Plus } X Y$
<proof>

theorem *Var-neq-FnCall* [iff]: $\text{Var } N \neq \text{FnCall } F Xs$
<proof>

4.7 Injectivity of *FnCall*

definition

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

lemma *fun-respects*: $(\%U. \{\text{freefun } U\})$ respects *exprel*
<proof>

lemma *fun-FnCall* [simp]: $\text{fun } (\text{FnCall } F Xs) = F$
<proof>

definition

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

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

lemma *Abs-ExpList-eq*:
 $(y, z) \in \text{listrel } \text{exprel} \implies \text{Abs-ExpList } (y) = \text{Abs-ExpList } (z)$
<proof>

lemma *args-respects*: $(\%U. \{\text{Abs-ExpList } (\text{freeargs } U)\})$ respects *exprel*
<proof>

lemma *args-FnCall* [simp]: $\text{args } (\text{FnCall } F Xs) = Xs$
<proof>

lemma *FnCall-FnCall-eq* [iff]:
 $(\text{FnCall } F Xs = \text{FnCall } F' Xs') = (F=F' \ \& \ Xs=Xs')$
<proof>

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* ⇒ *int* **where**
discrim *X* = *the-elem* (∪ *U* ∈ *Rep-Exp* *X*. {*freediscrim* *U*})

lemma *discrim-respects*: (λ*U*. {*freediscrim* *U*}) *respects exprel*
 ⟨*proof*⟩

Now prove the four equations for *discrim*

lemma *discrim-Var* [*simp*]: *discrim* (*Var* *N*) = 0
 ⟨*proof*⟩

lemma *discrim-Plus* [*simp*]: *discrim* (*Plus* *X* *Y*) = 1
 ⟨*proof*⟩

lemma *discrim-FnCall* [*simp*]: *discrim* (*FnCall* *F* *Xs*) = 2
 ⟨*proof*⟩

The structural induction rule for the abstract type

theorem *exp-inducts*:

assumes *V*: ∧ *nat*. *P1* (*Var* *nat*)
and *P*: ∧ *exp1 exp2*. [[*P1* *exp1*; *P1* *exp2*]] ⇒ *P1* (*Plus* *exp1* *exp2*)
and *F*: ∧ *nat list*. *P2* *list* ⇒ *P1* (*FnCall* *nat* *list*)
and *Nil*: *P2* []
and *Cons*: ∧ *exp list*. [[*P1* *exp*; *P2* *list*]] ⇒ *P2* (*exp* # *list*)
shows *P1* *exp* **and** *P2* *list*
 ⟨*proof*⟩

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

primrec *subst-term* :: ('*a* ⇒ ('*a*, '*b*) *term*) ⇒ ('*a*, '*b*) *term* ⇒ ('*a*, '*b*) *term*
and *subst-term-list* :: ('*a* ⇒ ('*a*, '*b*) *term*) ⇒ ('*a*, '*b*) *term* *list* ⇒ ('*a*, '*b*) *term* *list*

where

$subst-term\ f\ (Var\ a) = f\ a$
| $subst-term\ f\ (App\ b\ ts) = App\ b\ (subst-term-list\ f\ ts)$
| $subst-term-list\ f\ [] = []$
| $subst-term-list\ f\ (t\ \#\ ts) = subst-term\ f\ t\ \#\ subst-term-list\ f\ ts$

A simple theorem about composition of substitutions

lemma *subst-comp*:

$subst-term\ (subst-term\ f1\ \circ\ f2)\ t =$
 $subst-term\ f1\ (subst-term\ f2\ t)$
and $subst-term-list\ (subst-term\ f1\ \circ\ f2)\ ts =$
 $subst-term-list\ f1\ (subst-term-list\ f2\ ts)$
{*proof*}

Alternative induction rule

lemma

assumes $var: \bigwedge v. P\ (Var\ v)$
and $app: \bigwedge f\ ts. (\forall t \in set\ ts. P\ t) \implies P\ (App\ f\ ts)$
shows *term-induct2*: $P\ t$
and $\forall t \in set\ ts. P\ t$
{*proof*}

end

theory *Sexp*

imports *HOL-Library.Old-Datatype*

begin

type-synonym $'a\ item = 'a\ Old-Datatype.item$

abbreviation $Leaf == Old-Datatype.Leaf$

abbreviation $Numb == Old-Datatype.Numb$

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 $:: [a \implies b, nat \implies b, [a\ item, 'a\ item] \implies b,$
 $'a\ item] \implies b$ **where**
 $sexp-case\ c\ d\ e\ M = (THE\ z. (\exists x. M = Leaf(x) \ \&\ z = c(x))$
 $| (\exists k. M = Numb(k) \ \&\ z = d(k))$
 $| (\exists N1\ N2. M = Scons\ N1\ N2 \ \&\ z = e\ N1\ N2))$

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)*
 <proof>

lemma *sexp-case-Numb* [*simp*]: *sexp-case c d e* (*Numb k*) = *d(k)*
 <proof>

lemma *sexp-case-Scons* [*simp*]: *sexp-case c d e* (*Scons M N*) = *e M N*
 <proof>

lemma *sexp-In0I*: *M* ∈ *sexp* ==> *In0(M)* ∈ *sexp*
 <proof>

lemma *sexp-In1I*: *M* ∈ *sexp* ==> *In1(M)* ∈ *sexp*
 <proof>

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

lemma *range-Leaf-subset-sexp*: *range(Leaf)* <= *sexp*
 <proof>

lemma *Scons-D*: *Scons M N* ∈ *sexp* ==> *M* ∈ *sexp* & *N* ∈ *sexp*
 <proof>

lemma *pred-sexp-subset-Sigma*: *pred-sexp* <= *sexp* × *sexp*
 <proof>

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

pred-sexp-subset-Sigma
 [THEN trancl-subset-Sigma, THEN subsetD, THEN SigmaD2]

lemma *pred-sexpI1*:
 [| M ∈ sexp; N ∈ sexp |] ==> (M, Scons M N) ∈ pred-sexp
 <proof>

lemma *pred-sexpI2*:
 [| M ∈ sexp; N ∈ sexp |] ==> (N, Scons M N) ∈ pred-sexp
 <proof>

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

lemmas *pred-sexp-trans1* [simp] = *trans-trancl* [THEN transD, OF -pred-sexp-t1]
and *pred-sexp-trans2* [simp] = *trans-trancl* [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
 <proof>

lemma *wf-pred-sexp*: *wf(pred-sexp)*
 <proof>

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)))
 <proof>

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)*
 <proof>

lemma *sexp-rec-Numb*: *sexp-rec (Numb k) c d h = d(k)*

<proof>

lemma *sexp-rec-Scons*: [| $M \in \text{sexp}$; $N \in \text{sexp}$ |] ==>
 $\text{sexp-rec } (\text{Scons } M \ N) \ c \ d \ h = h \ M \ N \ (\text{sexp-rec } M \ c \ d \ h) \ (\text{sexp-rec } N \ c \ d \ h)$
<proof>

end

6 Extended List Theory (old)

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*: [| $a \in A$; $M \in \text{list } A$ |] ==> *CONS* *a M* ∈ *list A*

definition *List* = *list* (*range Leaf*)

typedef 'a list = *List* :: 'a item set

morphisms *Rep-List Abs-List*

<proof>

abbreviation *Case* == *Old-Datatype.Case*

abbreviation *Split* == *Old-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*

no-translations

[*x*, *xs*] == *x#[xs]*

[*x*] == *x#[]*

no-notation

Nil ([]) **and**

Cons (**infixr** # 65)

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* **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* **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

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

definition

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

definition

$map :: ('a \Rightarrow 'b) \Rightarrow ('a \text{ list} \Rightarrow 'b \text{ list})$ **where**
 $map f xs = list\text{-}rec\ xs\ [](\%x\ l\ r.\ f(x)\#r)$

primrec $take :: ['a \text{ list}, nat] \Rightarrow 'a \text{ list}$ **where**

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

lemma $ListI:$ $x \in list\ (range\ Leaf) \Longrightarrow x \in List$
 $\langle proof \rangle$

lemma $ListD:$ $x \in List \Longrightarrow x \in list\ (range\ Leaf)$
 $\langle proof \rangle$

lemma $list\text{-}unfold:$ $list(A) = usum\ \{Numb(0)\}\ (uprod\ A\ (list(A)))$
 $\langle proof \rangle$

lemma $list\text{-}mono:$ $A \leq B \Longrightarrow list(A) \leq list(B)$
 $\langle proof \rangle$

lemma $list\text{-}sexp:$ $list(sexp) \leq sexp$
 $\langle proof \rangle$

lemmas $list\text{-}subset\text{-}sexp = subset\text{-}trans\ [OF\ list\text{-}mono\ list\text{-}sexp]$

lemma *list-induct*:

[[$P(\text{Nil})$;
!! $x\ xs. P(xs) \implies P(x \# xs)$]] $\implies P(l)$
<proof>

lemma *inj-on-Abs-list*: *inj-on Abs-List (list(range Leaf))*
<proof>

lemma *CONS-not-NIL* [iff]: $\text{CONS } M\ N \sim = \text{NIL}$
<proof>

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

lemma *Cons-not-Nil* [iff]: $x \# xs \sim = \text{Nil}$
<proof>

lemmas *Nil-not-Cons* = *Cons-not-Nil* [THEN not-sym]
declare *Nil-not-Cons* [iff]
lemmas *Cons-neq-Nil* = *Cons-not-Nil* [THEN notE]
lemmas *Nil-neq-Cons* = *sym* [THEN *Cons-neq-Nil*]

lemma *CONS-CONS-eq* [iff]: $(\text{CONS } K\ M) = (\text{CONS } L\ N) = (K=L \ \& \ M=N)$
<proof>

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

lemma *Cons-Cons-eq* [iff]: $(x \# xs = y \# ys) = (x=y \ \& \ xs=ys)$
<proof>

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

lemma *CONS-D*: $\text{CONS } M\ N \in \text{list}(A) \implies M \in A \ \& \ N \in \text{list}(A)$
<proof>

lemma *sexp-CONS-D*: $\text{CONS } M\ N \in \text{sexp} \implies M \in \text{sexp} \wedge N \in \text{sexp}$
<proof>

lemma *not-CONS-self*: $N \in \text{list}(A) \implies \forall M. N \neq \text{CONS } M \ N$
<proof>

lemma *not-Cons-self2*: $\forall x. l \neq x\#l$
<proof>

lemma *neq-Nil-conv2*: $(xs \neq []) = (\exists y \ ys. xs = y\#ys)$
<proof>

lemma *List-case-NIL* [*simp*]: $\text{List-case } c \ h \ \text{NIL} = c$
<proof>

lemma *List-case-CONS* [*simp*]: $\text{List-case } c \ h \ (\text{CONS } M \ N) = h \ M \ N$
<proof>

lemma *List-rec-unfold-lemma*:
 $(\lambda M. \text{List-rec } M \ c \ d) \equiv$
 $\text{wfrec } (\text{pred-sexp}^+) (\lambda g. \text{List-case } c \ (\lambda x \ y. d \ x \ y \ (g \ y)))$
<proof>

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

lemma *pred-sexp-CONS-I1*:
 $[M \in \text{sexp}; N \in \text{sexp}] \implies (M, \text{CONS } M \ N) \in \text{pred-sexp}^+$
<proof>

lemma *pred-sexp-CONS-I2*:
 $[M \in \text{sexp}; N \in \text{sexp}] \implies (N, \text{CONS } M \ N) \in \text{pred-sexp}^+$
<proof>

lemma *pred-sexp-CONS-D*:
 $(\text{CONS } M1 \ M2, N) \in \text{pred-sexp}^+ \implies$

$(M1, N) \in \text{pred-sexp}^+ \wedge (M2, N) \in \text{pred-sexp}^+$
 $\langle \text{proof} \rangle$

lemma *List-rec-NIL* [*simp*]: *List-rec NIL c h = c*
 $\langle \text{proof} \rangle$

lemma *List-rec-CONS* [*simp*]:
 $\llbracket M \in \text{sexp}; N \in \text{sexp} \rrbracket$
 $\implies \text{List-rec} (\text{CONS } M N) c h = h M N (\text{List-rec } N c h)$
 $\langle \text{proof} \rangle$

lemmas *Rep-List-in-sexp* =
 $\text{subsetD} [\text{OF range-Leaf-subset-sexp} [\text{THEN list-subset-sexp}]]$
 $\text{Rep-List} [\text{THEN ListD}]$

lemma *list-rec-Nil* [*simp*]: *list-rec Nil c h = c*
 $\langle \text{proof} \rangle$

lemma *list-rec-Cons* [*simp*]: *list-rec (a#l) c h = h a l (list-rec l c h)*
 $\langle \text{proof} \rangle$

lemma *List-rec-type*:
 $\llbracket M \in \text{list}(A);$
 $A \leq \text{sexp};$
 $c \in C(\text{NIL});$
 $\bigwedge x y r. \llbracket x \in A; y \in \text{list}(A); r \in C(y) \rrbracket \implies h x y r \in C(\text{CONS } x y)$
 $\rrbracket \implies \text{List-rec } M c h \in C(M :: \text{'a item})$
 $\langle \text{proof} \rangle$

lemma *Rep-map-Nil* [*simp*]: *Rep-map f Nil = NIL*
 $\langle \text{proof} \rangle$

lemma *Rep-map-Cons* [*simp*]:
 $\text{Rep-map } f(x\#xs) = \text{CONS}(f x)(\text{Rep-map } f xs)$
 $\langle \text{proof} \rangle$

lemma *Rep-map-type*: $(\bigwedge x. f(x) \in A) \implies \text{Rep-map } f \text{ } xs \in \text{list}(A)$
 ⟨proof⟩

lemma *Abs-map-NIL* [*simp*]: $\text{Abs-map } g \text{ } \text{NIL} = \text{Nil}$
 ⟨proof⟩

lemma *Abs-map-CONS* [*simp*]:
 $[\![M \in \text{sexp}; N \in \text{sexp}]\!] \implies \text{Abs-map } g \text{ } (\text{CONS } M \text{ } N) = g(M) \# \text{Abs-map } g \text{ } N$
 ⟨proof⟩

lemma *def-list-rec-NilCons*:
 $[\![\bigwedge xs. f(xs) = \text{list-rec } xs \text{ } c \text{ } h]\!] \implies f [] = c \wedge f(x\#xs) = h \text{ } x \text{ } xs \text{ } (f \text{ } xs)$
 ⟨proof⟩

lemma *Abs-map-inverse*:
 $[\![M \in \text{list}(A); A \leq \text{sexp}; \bigwedge z. z \in A \implies f(g(z)) = z]\!] \implies \text{Rep-map } f \text{ } (\text{Abs-map } g \text{ } M) = M$
 ⟨proof⟩

Better to have a single theorem with a conjunctive conclusion.

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

lemma *expand-list-case*:
 $P(\text{list-case } a \text{ } f \text{ } xs) = ((xs = [] \longrightarrow P \text{ } a) \wedge (\forall y \text{ } ys. xs = y\#ys \longrightarrow P(f \text{ } y \text{ } ys)))$
 ⟨proof⟩

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

lemma *Abs-Rep-map*:
 $(\bigwedge x. f(x) \in \text{sexp}) \implies \text{Abs-map } g \text{ } (\text{Rep-map } f \text{ } xs) = \text{map } (\lambda t. g(f(t))) \text{ } xs$
 ⟨proof⟩

lemma *map-ident* [*simp*]: $\text{map}(\%x. x)(xs) = xs$
 ⟨proof⟩

lemma *map-compose*: $\text{map}(f \circ g)(xs) = \text{map } f (\text{map } g \text{ } xs)$
<proof>

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

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

lemma *take-Nil* [simp]: $\text{take } [] \text{ } n = []$
<proof>

lemma *take-take-eq* [simp]: $\forall n. \text{take } (\text{take } xs \text{ } n) \text{ } n = \text{take } xs \text{ } n$
<proof>

end

7 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

primrec *evala* :: ('a \Rightarrow *nat*) \Rightarrow 'a *aexp* \Rightarrow *nat*
and *evalb* :: ('a \Rightarrow *nat*) \Rightarrow 'a *bexp* \Rightarrow *bool*

where

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

```

primrec subst :: ('a ⇒ 'b aexp) ⇒ 'a aexp ⇒ 'b aexp
and subst :: ('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp

```

where

```

  subst f (IF b a1 a2) = IF (subst 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

```

```

| subst f (Less a1 a2) = Less (subst f a1) (subst f a2)
| subst f (And b1 b2) = And (subst f b1) (subst f b2)
| subst f (Neg b) = Neg (subst f b)

```

lemma subst1-aexp:

```

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

```

and subst1-bexp:

```

  evalb env (subst (Var (v := a')) b) = evalb (env (v := evala env a')) b

```

— one variable

<proof>

lemma subst-all-aexp:

```

  evala env (subst s a) = evala (λx. evala env (s x)) a

```

and subst-all-bexp:

```

  evalb env (subst s b) = evalb (λx. evala env (s x)) b

```

<proof>

end

8 Infinitely branching trees

theory *Infinitely-Branching-Tree*

imports *Main*

begin

datatype 'a tree =

```

  Atom 'a

```

```

| Branch nat ⇒ 'a tree

```

primrec map-tree :: ('a ⇒ 'b) ⇒ 'a tree ⇒ 'b tree

where

```

  map-tree f (Atom a) = Atom (f a)

```

```

| map-tree f (Branch ts) = Branch (λx. map-tree f (ts x))

```

lemma *tree-map-compose*: $\text{map-tree } g (\text{map-tree } f t) = \text{map-tree } (g \circ f) t$
 ⟨*proof*⟩

primrec *exists-tree* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ tree} \Rightarrow \text{bool}$
where
 exists-tree $P (\text{Atom } a) = P a$
 | *exists-tree* $P (\text{Branch } ts) = (\exists x. \text{exists-tree } P (ts x))$

lemma *exists-map*:
 $(\bigwedge x. P x \Longrightarrow Q (f x)) \Longrightarrow$
 $\text{exists-tree } P ts \Longrightarrow \text{exists-tree } Q (\text{map-tree } f ts)$
 ⟨*proof*⟩

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

datatype *brouwer* = *Zero* | *Succ brouwer* | *Lim nat* \Rightarrow *brouwer*

Addition of ordinals

primrec *add* :: *brouwer* \Rightarrow *brouwer* \Rightarrow *brouwer*
where
 add $i \text{ Zero} = i$
 | *add* $i (\text{Succ } j) = \text{Succ } (\text{add } i j)$
 | *add* $i (\text{Lim } f) = \text{Lim } (\lambda n. \text{add } i (f n))$

lemma *add-assoc*: $\text{add } (\text{add } i j) k = \text{add } i (\text{add } j k)$
 ⟨*proof*⟩

Multiplication of ordinals

primrec *mult* :: *brouwer* \Rightarrow *brouwer* \Rightarrow *brouwer*
where
 mult $i \text{ Zero} = \text{Zero}$
 | *mult* $i (\text{Succ } j) = \text{add } (\text{mult } i j) i$
 | *mult* $i (\text{Lim } f) = \text{Lim } (\lambda n. \text{mult } i (f n))$

lemma *add-mult-distrib*: $\text{mult } i (\text{add } j k) = \text{add } (\text{mult } i j) (\text{mult } i k)$
 ⟨*proof*⟩

lemma *mult-assoc*: $\text{mult } (\text{mult } i j) k = \text{mult } i (\text{mult } j k)$
 ⟨*proof*⟩

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

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

To use the function package 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-pred* = ($\bigcup i. \{(m, n). n = \text{Succ } m \vee (\exists f. n = \text{Lim } f \wedge m = f\ i)\}$)

definition *brouwer-order* :: (*brouwer* × *brouwer*) *set*
where *brouwer-order* = *brouwer-pred*⁺

lemma *wf-brouwer-pred*: *wf brouwer-pred*
 ⟨*proof*⟩

lemma *wf-brouwer-order[simp]*: *wf brouwer-order*
 ⟨*proof*⟩

lemma [*simp*]: (*j*, *Succ j*) ∈ *brouwer-order*
 ⟨*proof*⟩

lemma [*simp*]: (*f n*, *Lim f*) ∈ *brouwer-order*
 ⟨*proof*⟩

Example of a general function

function *add2* :: *brouwer* ⇒ *brouwer* ⇒ *brouwer*
where
add2 i Zero = *i*
 | *add2 i (Succ j)* = *Succ (add2 i j)*
 | *add2 i (Lim f)* = *Lim (λn. add2 i (f n))*
 ⟨*proof*⟩

termination
 ⟨*proof*⟩

lemma *add2-assoc*: *add2 (add2 i j) k* = *add2 i (add2 j k)*
 ⟨*proof*⟩

end

9 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*

```

primrec pred :: ordinal  $\Rightarrow$  nat  $\Rightarrow$  ordinal option
where
  pred Zero n = None
| pred (Succ a) n = Some a
| pred (Limit f) n = Some (f n)

abbreviation (input) iter :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  nat  $\Rightarrow$  ('a  $\Rightarrow$  'a)
  where iter f n  $\equiv$  f  $\hat{\sim}$  n

definition OpLim :: (nat  $\Rightarrow$  (ordinal  $\Rightarrow$  ordinal))  $\Rightarrow$  (ordinal  $\Rightarrow$  ordinal)
  where OpLim F a = Limit ( $\lambda$ n. F n a)

definition OpItw :: (ordinal  $\Rightarrow$  ordinal)  $\Rightarrow$  (ordinal  $\Rightarrow$  ordinal) ( $\sqcup$ )
  where  $\sqcup$ f = OpLim (iter f)

primrec cantor :: ordinal  $\Rightarrow$  ordinal  $\Rightarrow$  ordinal
where
  cantor a Zero = Succ a
| cantor a (Succ b) =  $\sqcup$ ( $\lambda$ x. cantor x b) a
| cantor a (Limit f) = Limit ( $\lambda$ n. cantor a (f n))

primrec NablA :: (ordinal  $\Rightarrow$  ordinal)  $\Rightarrow$  (ordinal  $\Rightarrow$  ordinal) ( $\nabla$ )
where
   $\nabla$ f Zero = f Zero
|  $\nabla$ f (Succ a) = f (Succ ( $\nabla$ f a))
|  $\nabla$ f (Limit h) = Limit ( $\lambda$ n.  $\nabla$ f (h n))

definition deriv :: (ordinal  $\Rightarrow$  ordinal)  $\Rightarrow$  (ordinal  $\Rightarrow$  ordinal)
  where deriv f =  $\nabla$ ( $\sqcup$ f)

primrec vebLen :: ordinal  $\Rightarrow$  ordinal  $\Rightarrow$  ordinal
where
  vebLen Zero =  $\nabla$ (OpLim (iter (cantor Zero)))
| vebLen (Succ a) =  $\nabla$ (OpLim (iter (vebLen a)))
| vebLen (Limit f) =  $\nabla$ (OpLim ( $\lambda$ n. vebLen (f n)))

definition veb a = vebLen a Zero
definition  $\varepsilon_0$  = veb Zero
definition  $\Gamma_0$  = Limit ( $\lambda$ n. iter veb n Zero)

end

```

10 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 σ -algebra :: 'a set set \Rightarrow 'a set set **for** A :: 'a set set

where

basic: $a \in \sigma$ -algebra A **if** $a \in A$ **for** a

| UNIV: UNIV $\in \sigma$ -algebra A

| complement: $- a \in \sigma$ -algebra A **if** $a \in \sigma$ -algebra A **for** a

| Union: $(\bigcup i. a i) \in \sigma$ -algebra A **if** $\bigwedge i::nat. a i \in \sigma$ -algebra A **for** 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*: $\{\} \in \sigma$ -algebra A

<proof>

theorem *sigma-algebra-Inter*:

$(\bigwedge i::nat. a i \in \sigma$ -algebra A) $\implies (\bigcap i. a i) \in \sigma$ -algebra A

<proof>

end

11 Combinatory Logic example: the Church-Rosser Theorem

theory Comb

imports Main

begin

Combinator terms do not have free variables. Example taken from [?].

11.1 Definitions

Datatype definition of combinators S and K .

datatype comb = K

| S

| Ap comb comb (**infixl** \cdot 90)

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

inductive contract1 :: [comb,comb] \Rightarrow bool (**infixl** \rightarrow^1 50)

where

K: $K \cdot x \cdot y \rightarrow^1 x$

| S: $S \cdot x \cdot y \cdot z \rightarrow^1 (x \cdot z) \cdot (y \cdot z)$

| Ap1: $x \rightarrow^1 y \implies x \cdot z \rightarrow^1 y \cdot z$

| Ap2: $x \rightarrow^1 y \implies z \cdot x \rightarrow^1 z \cdot y$

abbreviation

contract :: [comb,comb] \Rightarrow bool (**infixl** \rightarrow 50) **where**

contract \equiv contract1**

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

inductive *parcontract1* :: [*comb,comb*] \Rightarrow *bool* (**infixl** \Rightarrow^1 50)

where

refl: $x \Rightarrow^1 x$
| *K*: $K \cdot x \cdot y \Rightarrow^1 x$
| *S*: $S \cdot x \cdot y \cdot z \Rightarrow^1 (x \cdot z) \cdot (y \cdot z)$
| *Ap*: $\llbracket x \Rightarrow^1 y; z \Rightarrow^1 w \rrbracket \Longrightarrow x \cdot z \Rightarrow^1 y \cdot w$

abbreviation

parcontract :: [*comb,comb*] \Rightarrow *bool* (**infixl** \Rightarrow 50) **where**
parcontract \equiv *parcontract1***

Misc definitions.

definition

I :: *comb* **where**
I \equiv *S* · *K* · *K*

definition

diamond :: ([*comb,comb*] \Rightarrow *bool*) \Rightarrow *bool* **where**
— confluence; Lambda/Commutation treats this more abstractly
diamond *r* \equiv $\forall x y. r x y \longrightarrow$
 $(\forall y'. r x y' \longrightarrow$
 $(\exists z. r y z \wedge r y' z))$

11.2 Reflexive/Transitive closure preserves Church-Rosser property

Remark: So does the Transitive closure, with a similar proof

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

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

assumes *diamond* *r* **and** *r*: $r^{**} x y r x y'$
shows $\exists z. r^{**} y' z \wedge r y z$
 \langle *proof* \rangle

proposition *diamond-rtrancl*:

assumes *diamond* *r*
shows *diamond*(r^{**})
 \langle *proof* \rangle

11.3 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases

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

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

declare $contract1.K$ [intro!] $contract1.S$ [intro!]
declare $contract1.Ap1$ [intro] $contract1.Ap2$ [intro]

lemma $I\text{-contract-E}$ [iff]: $\neg I \rightarrow^1 z$
 $\langle proof \rangle$

lemma $K1\text{-contractD}$ [elim!]: $K \cdot x \rightarrow^1 z \implies (\exists x'. z = K \cdot x' \wedge x \rightarrow^1 x')$
 $\langle proof \rangle$

lemma $Ap\text{-reduce1}$ [intro]: $x \rightarrow y \implies x \cdot z \rightarrow y \cdot z$
 $\langle proof \rangle$

lemma $Ap\text{-reduce2}$ [intro]: $x \rightarrow y \implies z \cdot x \rightarrow z \cdot y$
 $\langle proof \rangle$

Counterexample to the diamond property for $x \rightarrow^1 y$

lemma $not\text{-diamond}\text{-contract}$: $\neg diamond(contract1)$
 $\langle proof \rangle$

11.4 Results about Parallel Contraction

Derive a case for each combinator constructor.

inductive-cases

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

declare $parcontract1.intros$ [intro]

11.5 Basic properties of parallel contraction

The rules below are not essential but make proofs much faster

lemma $K1\text{-parcontractD}$ [dest!]: $K \cdot x \Rightarrow^1 z \implies (\exists x'. z = K \cdot x' \wedge x \Rightarrow^1 x')$
 $\langle proof \rangle$

lemma $S1\text{-parcontractD}$ [dest!]: $S \cdot x \Rightarrow^1 z \implies (\exists x'. z = S \cdot x' \wedge x \Rightarrow^1 x')$
 $\langle proof \rangle$

lemma $S2\text{-parcontractD}$ [dest!]: $S \cdot x \cdot y \Rightarrow^1 z \implies (\exists x' y'. z = S \cdot x' \cdot y' \wedge x \Rightarrow^1 x' \wedge y \Rightarrow^1 y')$
 $\langle proof \rangle$

Church-Rosser property for parallel contraction

proposition $diamond\text{-parcontract}$: $diamond\ parcontract1$
 $\langle proof \rangle$

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

lemma *contract-imp-parcontract*: $x \rightarrow^1 y \Longrightarrow x \Rightarrow^1 y$
<proof>

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

proposition *reduce-I*: $I \cdot x \rightarrow x$
<proof>

lemma *parcontract-imp-reduce*: $x \Rightarrow^1 y \Longrightarrow x \rightarrow y$
<proof>

lemma *reduce-eq-parreduce*: $x \rightarrow y \longleftrightarrow x \Rightarrow y$
<proof>

theorem *diamond-reduce*: *diamond(contract)*
<proof>

end

12 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 $G \in \text{Fin}(H)$

12.1 The datatype of propositions

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

12.2 The proof system

```
inductive thms :: ['a pl set, 'a pl] => bool (infixl |- 50)  
  for H :: 'a pl set  
where  
  H:  $p \in H \implies H \vdash p$   
| K:  $H \vdash p \rightarrow q \rightarrow p$   
| S:  $H \vdash (p \rightarrow q \rightarrow r) \rightarrow (p \rightarrow q) \rightarrow p \rightarrow r$   
| DN:  $H \vdash ((p \rightarrow \text{false}) \rightarrow \text{false}) \rightarrow p$   
| MP:  $[| H \vdash p \rightarrow q; H \vdash p |] \implies H \vdash q$ 
```

12.3 The semantics

12.3.1 Semantics of propositional logic.

primrec *eval* :: [*'a set, 'a pl*] => *bool* (*-*[[*-*]] [100,0] 100)
where
 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*.

primrec *hyps* :: [*'a pl, 'a set*] => *'a pl set*
where
 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*

12.3.2 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*]])

12.4 Proof theory of propositional logic

lemma *thms-mono*: *G* <= *H* ==> *thms*(*G*) <= *thms*(*H*)
 ⟨*proof*⟩

lemma *thms-I*: *H* | - *p*->*p*
 — Called *I* for Identity Combinator, not for Introduction.
 ⟨*proof*⟩

12.4.1 Weakening, left and right

lemma *weaken-left*: [| *G* ⊆ *H*; *G* | - *p* |] ==> *H* | - *p*
 — Order of premises is convenient with *THEN*
 ⟨*proof*⟩

lemma *weaken-left-insert*: *G* | - *p* ==> *insert a G* | - *p*
 ⟨*proof*⟩

lemma *weaken-left-Un1*: *G* | - *p* ==> *G* ∪ *B* | - *p*
 ⟨*proof*⟩

lemma *weaken-left-Un2*: *G* | - *p* ==> *A* ∪ *G* | - *p*
 ⟨*proof*⟩

lemma *weaken-right*: *H* | - *q* ==> *H* | - *p*->*q*
 ⟨*proof*⟩

12.4.2 The deduction theorem

theorem *deduction*: $\text{insert } p \ H \ |- \ q \ ==> \ H \ |- \ p \ -> \ q$
(*proof*)

12.4.3 The cut rule

lemma *cut*: $\text{insert } p \ H \ |- \ q \ ==> \ H \ |- \ p \ ==> \ H \ |- \ q$
(*proof*)

lemma *thms-falseE*: $H \ |- \ \text{false} \ ==> \ H \ |- \ q$
(*proof*)

lemma *thms-notE*: $H \ |- \ p \ -> \ \text{false} \ ==> \ H \ |- \ p \ ==> \ H \ |- \ q$
(*proof*)

12.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: $H \ |- \ p \ ==> \ H \ |= \ p$
(*proof*)

12.5 Completeness

12.5.1 Towards the completeness proof

lemma *false-imp*: $H \ |- \ p \ -> \ \text{false} \ ==> \ H \ |- \ p \ -> \ q$
(*proof*)

lemma *imp-false*:
[[$H \ |- \ p$; $H \ |- \ q \ -> \ \text{false}$]] $\implies H \ |- \ (p \ -> \ q) \ -> \ \text{false}$
(*proof*)

lemma *hyps-thms-if*: $\text{hyps } p \ \text{tt} \ |- \ (\text{if } \text{tt}[[p]] \ \text{then } p \ \text{else } p \ -> \ \text{false})$
— Typical example of strengthening the induction statement.
(*proof*)

lemma *sat-thms-p*: $\{\} \ |= \ p \ ==> \ \text{hyps } p \ \text{tt} \ |- \ p$
— Key lemma for completeness; yields a set of assumptions satisfying p
(*proof*)

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*: $H \ |- \ (p \ -> \ q) \ -> \ ((p \ -> \ \text{false}) \ -> \ q) \ -> \ q$
(*proof*)

lemma *thms-excluded-middle-rule*:
[[$\text{insert } p \ H \ |- \ q$; $\text{insert } (p \ -> \ \text{false}) \ H \ |- \ q$]] $\implies H \ |- \ q$

— Hard to prove directly because it requires cuts
 ⟨proof⟩

12.6 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*: $\text{hyps } p \ (t - \{v\}) \leq \text{insert } (\#v \rightarrow \text{false}) \ ((\text{hyps } p \ t) - \{\#v\})$
 ⟨proof⟩

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*: $\text{hyps } p \ (\text{insert } v \ t) \leq \text{insert } (\#v) \ (\text{hyps } p \ t - \{\#v \rightarrow \text{false}\})$
 ⟨proof⟩

Two lemmas for use with *weaken-left*

lemma *insert-Diff-same*: $B - C \leq \text{insert } a \ (B - \text{insert } a \ C)$
 ⟨proof⟩

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

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

lemma *hyps-finite*: $\text{finite}(\text{hyps } p \ t)$
 ⟨proof⟩

lemma *hyps-subset*: $\text{hyps } p \ t \leq (\text{UN } v. \{\#v, \#v \rightarrow \text{false}\})$
 ⟨proof⟩

lemma *Diff-weaken-left*: $A \subseteq C \implies A - B \mid- p \implies C - B \mid- p$
 ⟨proof⟩

12.6.1 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*:
 $\{\} \mid= p \implies \forall t. \text{hyps } p \ t - \text{hyps } p \ t0 \mid- p$
 ⟨proof⟩

The base case for completeness

lemma *completeness-0*: $\{\} \mid= p \implies \{\} \mid- p$
 ⟨proof⟩

A semantic analogue of the Deduction Theorem

lemma *sat-imp*: $insert\ p\ H\ \models\ q\ ==>\ H\ \models\ p \rightarrow q$
 <proof>

theorem *completeness*: $finite\ H\ ==>\ H\ \models\ p\ ==>\ H\ \vdash\ p$
 <proof>

theorem *syntax-iff- semantics*: $finite\ H\ ==>\ (H\ \vdash\ p) = (H\ \models\ p)$
 <proof>

end

13 Mutual Induction via Iterated Inductive Definitions

theory *Com* imports *Main* begin

typedecl *loc*

type-synonym *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)

13.1 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*: $(SKIP, s) \text{ --[eval]--> } s$

| *Assign*: $(e, s) \text{ --[eval]--> } (v, s') \implies (x := e, s) \text{ --[eval]--> } s'(x:=v)$

| *Semi*: $[[(c0, s) \text{ --[eval]--> } s2; (c1, s2) \text{ --[eval]--> } s1]]$
 $\implies (c0 ;; c1, s) \text{ --[eval]--> } s1$

| *IfTrue*: $[[(e, s) \text{ --[eval]--> } (0, s'); (c0, s') \text{ --[eval]--> } s1]]$
 $\implies (IF\ e\ THEN\ c0\ ELSE\ c1, s) \text{ --[eval]--> } s1$

| *IfFalse*: $[[(e, s) \text{ --[eval]--> } (Suc\ 0, s'); (c1, s') \text{ --[eval]--> } s1]]$
 $\implies (IF\ e\ THEN\ c0\ ELSE\ c1, s) \text{ --[eval]--> } s1$

| *WhileFalse*: $(e, s) \text{ --[eval]--> } (Suc\ 0, s1)$
 $\implies (WHILE\ e\ DO\ c, s) \text{ --[eval]--> } s1$

| *WhileTrue*: $[[(e, s) \text{ --[eval]--> } (0, s1);$
 $(c, s1) \text{ --[eval]--> } s2; (WHILE\ e\ DO\ c, s2) \text{ --[eval]--> } s3]]$
 $\implies (WHILE\ e\ DO\ c, s) \text{ --[eval]--> } s3$

declare *exec.intros* [intro]

inductive-cases

[*elim!*]: $(SKIP, s) \text{ --[eval]--> } t$
and [*elim!*]: $(x:=a, s) \text{ --[eval]--> } t$
and [*elim!*]: $(c1;;c2, s) \text{ --[eval]--> } t$
and [*elim!*]: $(IF\ e\ THEN\ c1\ ELSE\ c2, s) \text{ --[eval]--> } t$
and *exec-WHILE-case*: $(WHILE\ b\ DO\ c, s) \text{ --[eval]--> } t$

Justifies using "exec" in the inductive definition of "eval"

lemma *exec-mono*: $A \leq B \implies \text{exec}(A) \leq \text{exec}(B)$
 <proof>

lemma [*pred-set-conv*]:
 $((\lambda x\ x'\ y\ y'. ((x, x'), (y, y')) \in R) \leq (\lambda x\ x'\ y\ y'. ((x, x'), (y, y')) \in S)) = (R \leq S)$
 <proof>

lemma [*pred-set-conv*]:
 $((\lambda x\ x'\ y. ((x, x'), y) \in R) \leq (\lambda x\ x'\ y. ((x, x'), y) \in S)) = (R \leq S)$
 <proof>

Command execution is functional (deterministic) provided evaluation is

theorem *single-valued-exec*: $\text{single-valued}\ ev \implies \text{single-valued}(\text{exec}\ ev)$
 <proof>

13.2 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))
 ⟨proof⟩

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 (case-prod (%v. case-prod (case-prod P v)))
 ⟨proof⟩

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.

```

$$\begin{aligned}
& \llbracket (c,s) \text{--}[eval \text{ Int } \{((e,s),(n,s'))\}] \text{--} s0; \\
& \quad (c,s) \text{--}[eval] \text{--} s0; \\
& \quad (e,s0) \text{--} \mid \text{--} (n,s1); P e s0 n s1 \rrbracket \\
& \implies P (\text{VALOF } c \text{ RESULTIS } e) s n s1 \\
& \llbracket \implies P e s n s' \\
\langle proof \rangle
\end{aligned}$$

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 $(c,s) \text{--}[eval] \text{--} s2$ can use unrestricted $eval!$ The reason is that the execution $(c,s) \text{--}[eval \text{ Int } \{\dots\}] \text{--} s1$ assures us that execution is functional on the argument (c,s) .

lemma *com-Unique*:

$$\begin{aligned}
& (c,s) \text{--}[eval \text{ Int } \{((e,s),(n,t)). \forall nt'. (e,s) \text{--} \mid \text{--} nt' \text{--} \mid \text{--} (n,t)=nt'\}] \text{--} s1 \\
& \implies \forall s2. (c,s) \text{--}[eval] \text{--} s2 \text{--} \mid \text{--} s2=s1 \\
\langle proof \rangle
\end{aligned}$$

Expression evaluation is functional, or deterministic

theorem *single-valued-eval: single-valued eval*

$\langle proof \rangle$

lemma *eval-N-E [dest!]*: $(N n, s) \text{--} \mid \text{--} (v, s') \implies (v = n \ \& \ s' = s)$

$\langle proof \rangle$

This theorem says that "WHILE TRUE DO c " cannot terminate

lemma *while-true-E*:

$$\begin{aligned}
& (c', s) \text{--}[eval] \text{--} t \implies c' = \text{WHILE } (N \ 0) \ \text{DO } c \implies \text{False} \\
\langle proof \rangle
\end{aligned}$$

13.3 Equivalence of IF e THEN c ;;(WHILE e DO c) ELSE SKIP and WHILE e DO c

lemma *while-if1*:

$$\begin{aligned}
& (c',s) \text{--}[eval] \text{--} t \\
& \implies c' = \text{WHILE } e \ \text{DO } c \implies \\
& \quad ((\text{IF } e \ \text{THEN } c;;c' \ \text{ELSE } \text{SKIP}, s) \text{--}[eval] \text{--} t) \\
\langle proof \rangle
\end{aligned}$$

lemma *while-if2*:

$$\begin{aligned}
& (c',s) \text{--}[eval] \text{--} t \\
& \implies c' = \text{IF } e \ \text{THEN } c;;(\text{WHILE } e \ \text{DO } c) \ \text{ELSE } \text{SKIP} \implies \\
& \quad (\text{WHILE } e \ \text{DO } c, s) \text{--}[eval] \text{--} t \\
\langle proof \rangle
\end{aligned}$$

theorem *while-if*:

$$\begin{aligned}
& ((\text{IF } e \ \text{THEN } c;;(\text{WHILE } e \ \text{DO } c) \ \text{ELSE } \text{SKIP}, s) \text{--}[eval] \text{--} t) = \\
& \quad ((\text{WHILE } e \ \text{DO } c, s) \text{--}[eval] \text{--} t) \\
\langle proof \rangle
\end{aligned}$$

13.4 Equivalence of (IF e THEN c1 ELSE c2);;c and IF e THEN (c1;;c) ELSE (c2;;c)

lemma *if-semi1*:

$$\begin{aligned} & (c',s) \text{ --[eval]--> } t \\ \implies & c' = (\text{IF } e \text{ THEN } c1 \text{ ELSE } c2);;c \implies \\ & (\text{IF } e \text{ THEN } (c1;;c) \text{ ELSE } (c2;;c), s) \text{ --[eval]--> } t \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *if-semi2*:

$$\begin{aligned} & (c',s) \text{ --[eval]--> } t \\ \implies & c' = \text{IF } e \text{ THEN } (c1;;c) \text{ ELSE } (c2;;c) \implies \\ & ((\text{IF } e \text{ THEN } c1 \text{ ELSE } c2);;c, s) \text{ --[eval]--> } t \\ & \langle \text{proof} \rangle \end{aligned}$$

theorem *if-semi*: $((\text{IF } e \text{ THEN } c1 \text{ ELSE } c2);;c, s) \text{ --[eval]--> } t =$

$$((\text{IF } e \text{ THEN } (c1;;c) \text{ ELSE } (c2;;c), s) \text{ --[eval]--> } t)$$

$\langle \text{proof} \rangle$

13.5 Equivalence of VALOF c1 RESULTIS (VALOF c2 RESULTIS e) and VALOF c1;;c2 RESULTIS e

lemma *valof-valof1*:

$$\begin{aligned} & (e',s) \text{ --|-> } (v,s') \\ \implies & e' = \text{VALOF } c1 \text{ RESULTIS } (\text{VALOF } c2 \text{ RESULTIS } e) \implies \\ & (\text{VALOF } c1;;c2 \text{ RESULTIS } e, s) \text{ --|-> } (v,s') \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *valof-valof2*:

$$\begin{aligned} & (e',s) \text{ --|-> } (v,s') \\ \implies & e' = \text{VALOF } c1;;c2 \text{ RESULTIS } e \implies \\ & (\text{VALOF } c1 \text{ RESULTIS } (\text{VALOF } c2 \text{ RESULTIS } e), s) \text{ --|-> } (v,s') \\ & \langle \text{proof} \rangle \end{aligned}$$

theorem *valof-valof*:

$$((\text{VALOF } c1 \text{ RESULTIS } (\text{VALOF } c2 \text{ RESULTIS } e), s) \text{ --|-> } (v,s')) =$$

$$((\text{VALOF } c1;;c2 \text{ RESULTIS } e, s) \text{ --|-> } (v,s'))$$

$\langle \text{proof} \rangle$

13.6 Equivalence of VALOF SKIP RESULTIS e and e

lemma *valof-skip1*:

$$\begin{aligned} & (e',s) \text{ --|-> } (v,s') \\ \implies & e' = \text{VALOF SKIP RESULTIS } e \implies \\ & (e, s) \text{ --|-> } (v,s') \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *valof-skip2*:

$$(e,s) \text{ --|-> } (v,s') \implies (\text{VALOF SKIP RESULTIS } e, s) \text{ --|-> } (v,s')$$

$\langle \text{proof} \rangle$

theorem *valof-skip*:

$$((VALOF\ SKIP\ RESULTIS\ e,\ s) \dashv\vdash (v,\ s')) = ((e,\ s) \dashv\vdash (v,\ s'))$$

<proof>

13.7 Equivalence of VALOF $x:=e$ RESULTIS x and e

lemma *valof-assign1*:

$$(e',s) \dashv\vdash (v,s'')$$
$$\implies e' = VALOF\ x:=e\ RESULTIS\ X\ x \implies$$
$$(\exists s'. (e,\ s) \dashv\vdash (v,s') \ \&\ (s'' = s'(x:=v)))$$

<proof>

lemma *valof-assign2*:

$$(e,s) \dashv\vdash (v,s') \implies (VALOF\ x:=e\ RESULTIS\ X\ x,\ s) \dashv\vdash (v,s'(x:=v))$$

<proof>

end