

Examples of Inductive and Coinductive Definitions in HOL

Stefan Berghofer
Tobias Nipkow
Lawrence C Paulson
Markus Wenzel

September 11, 2023

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	4
1.1	Variations on statement structure	4
1.1.1	Local facts and parameters	4
1.1.2	Local definitions	4
1.1.3	Simple simultaneous goals	5
1.1.4	Compound simultaneous goals	5
1.2	Multiple rules	5
1.3	Inductive predicates	7
2	Nested datatypes	8
2.1	Terms and substitution	8
2.2	Alternative induction	8
3	Defining an Initial Algebra by Quotienting a Free Algebra	9
3.1	Defining the Free Algebra	9
3.2	Some Functions on the Free Algebra	10
3.2.1	The Set of Nonces	10
3.2.2	The Left Projection	10
3.2.3	The Right Projection	10
3.2.4	The Discriminator for Constructors	11
3.3	The Initial Algebra: A Quotiented Message Type	11
3.3.1	Characteristic Equations for the Abstract Constructors	12

3.4	The Abstract Function to Return the Set of Nonces	12
3.5	The Abstract Function to Return the Left Part	13
3.6	The Abstract Function to Return the Right Part	13
3.7	Injectivity Properties of Some Constructors	14
3.8	The Abstract Discriminator	15
4	Quotienting a Free Algebra Involving Nested Recursion	15
4.1	Defining the Free Algebra	16
4.2	Some Functions on the Free Algebra	17
4.2.1	The Set of Variables	17
4.2.2	Functions for Freeness	17
4.3	The Initial Algebra: A Quotiented Message Type	18
4.4	Every list of abstract expressions can be expressed in terms of a list of concrete expressions	19
4.4.1	Characteristic Equations for the Abstract Constructors	19
4.5	The Abstract Function to Return the Set of Variables	20
4.6	Injectivity Properties of Some Constructors	20
4.7	Injectivity of <i>FnCall</i>	21
4.8	The Abstract Discriminator	21
5	Terms over a given alphabet	22
6	Extended List Theory (old)	26
7	Arithmetic and boolean expressions	33
8	Infinitely branching trees	34
8.1	The Brouwer ordinals, as in ZF/Induct/Brouwer.thy.	35
8.2	A WF Ordering for The Brouwer ordinals (Michael Compton)	35
9	Ordinals	36
10	Sigma algebras	37
11	Combinatory Logic example: the Church-Rosser Theorem	38
11.1	Definitions	38
11.2	Reflexive/Transitive closure preserves Church-Rosser property	39
11.3	Non-contraction results	39
11.4	Results about Parallel Contraction	40
11.5	Basic properties of parallel contraction	40
11.6	Equivalence of $p \rightarrow q$ and $p \Rightarrow q$	40

12	Meta-theory of propositional logic	41
12.1	The datatype of propositions	41
12.2	The proof system	41
12.3	The semantics	41
12.3.1	Semantics of propositional logic.	41
12.3.2	Logical consequence	42
12.4	Proof theory of propositional logic	42
12.4.1	Weakening, left and right	42
12.4.2	The deduction theorem	42
12.4.3	The cut rule	43
12.4.4	Soundness of the rules wrt truth-table semantics	43
12.5	Completeness	43
12.5.1	Towards the completeness proof	43
12.6	Completeness – lemmas for reducing the set of assumptions	44
12.6.1	Completeness theorem	44
13	Mutual Induction via Iterated Inductive Definitions	45
13.1	Commands	45
13.2	Expressions	46
13.3	Equivalence of IF e THEN c;;(WHILE e DO c) ELSE SKIP and WHILE e DO c	48
13.4	Equivalence of (IF e THEN c1 ELSE c2);;c and IF e THEN (c1;;c) ELSE (c2;;c)	48
13.5	Equivalence of VALOF c1 RESULTIS (VALOF c2 RESULTIS e) and VALOF c1;;c2 RESULTIS e	49
13.6	Equivalence of VALOF SKIP RESULTIS e and e	49
13.7	Equivalence of VALOF x:=e RESULTIS x and e	50

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

For Lawrence Paulson’s paper “Defining functions on equivalence classes” *ACM Transactions on Computational Logic* **7**:40 (2006), 658–675, illustrating bare-bones quotient constructions. Any comparison using lifting and transfer should be done in a separate theory.

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*

⟨*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*]:
 $(\bigwedge U. z = \text{Abs-Msg} (\text{msgrel}\{\! \{U\}\! \}) \implies P) \implies P$
<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]: $(Crypt\ K\ X = Crypt\ K\ X') = (X=X')$
 ⟨proof⟩

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

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

$discrim :: msg \Rightarrow int$ **where**
 $discrim\ X = the_elem\ (\bigcup U \in Rep_Msg\ X. \{freediscrim\ U\})$

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

Now prove the four equations for *discrim*

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

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

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

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

end

4 Quotienting a Free Algebra Involving Nested Recursion

This is the development promised in Lawrence Paulson's paper "Defining functions on equivalence classes" *ACM Transactions on Computational Logic*

7:40 (2006), 658–675, illustrating bare-bones quotient constructions. Any comparison using lifting and transfer should be done in a separate theory.

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-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*
 \langle *proof* \rangle

4.3 The Initial Algebra: A Quotiented Message Type

definition $Exp = UNIV // exprel$

typedef $exp = Exp$
morphisms *Rep-Exp Abs-Exp*
 \langle *proof* \rangle

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

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

lemma *inj-on-Abs-Exp*: *inj-on Abs-Exp Exp*
 \langle *proof* \rangle

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*]:
 $(\bigwedge U. z = \text{Abs-Exp}(\text{exprel}\{\{U\}\}) \implies P) \implies P$
(*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* ($\lambda U. \text{Abs-Exp}(\text{exprel}\{\{U\}\})$) *Xs*

lemma *Abs-ExpList-Nil* [*simp*]: *Abs-ExpList* [] = []
(*proof*)

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

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

4.4.1 Characteristic Equations for the Abstract Constructors

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

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*: *FnCall* *F* [] = *Abs-Exp* (*exprel*{*FNCALL F []*})
(*proof*)

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

lemma *FnCall-sing*:
FnCall *F* [*Abs-Exp*(*exprel*{*U*})] = *Abs-Exp* (*exprel*{*FNCALL F [U]*})
(*proof*)

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

$listset (map Rep-Exp (Abs-ExpList Us)) = listrel\ exprel\ \{\{Us\}$
 $\langle proof \rangle$

lemma *FnCall*:

$FnCall\ F\ (Abs-ExpList\ Us) = Abs-Exp\ (exprel\ \{\{FNCALL\ F\ Us\})$
 $\langle proof \rangle$

Establishing this equation is the point of the whole exercise

theorem *Plus-assoc*: $Plus\ X\ (Plus\ Y\ Z) = Plus\ (Plus\ X\ Y)\ Z$
 $\langle proof \rangle$

4.5 The Abstract Function to Return the Set of Variables

definition

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

lemma *vars-respects*: *freevars respects exprel*
 $\langle proof \rangle$

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*]: $vars\ (Var\ N) = \{N\}$
 $\langle proof \rangle$

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

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

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

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

4.6 Injectivity Properties of Some Constructors

lemma *VAR-imp-eq*: $VAR\ m \sim VAR\ n \implies m = n$
 $\langle proof \rangle$

Can also be proved using the function *vars*

lemma *Var-Var-eq* [*iff*]: $(Var\ m = Var\ n) = (m = n)$
 $\langle proof \rangle$

lemma *VAR-neqv-PLUS*: $VAR\ m \sim PLUS\ X\ Y \implies False$
 ⟨*proof*⟩

theorem *Var-neq-Plus* [*iff*]: $Var\ N \neq Plus\ X\ Y$
 ⟨*proof*⟩

theorem *Var-neq-FnCall* [*iff*]: $Var\ N \neq FnCall\ F\ Xs$
 ⟨*proof*⟩

4.7 Injectivity of *FnCall*

definition

fun :: $exp \Rightarrow nat$
where $fun\ X \equiv the_elem\ (\bigcup U \in Rep_Exp\ X.\ \{freefun\ U\})$

lemma *fun-respects*: $(\lambda U.\ \{freefun\ U\})$ respects *exprel*
 ⟨*proof*⟩

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

definition

args :: $exp \Rightarrow exp\ list$ **where**
 $args\ X = the_elem\ (\bigcup U \in Rep_Exp\ X.\ \{Abs_ExpList\ (freeargs\ U)\})$

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

lemma *Abs-ExpList-eq*:
 $(y, z) \in listrel\ exprel \implies Abs_ExpList\ (y) = Abs_ExpList\ (z)$
 ⟨*proof*⟩

lemma *args-respects*: $(\lambda U.\ \{Abs_ExpList\ (freeargs\ U)\})$ respects *exprel*
 ⟨*proof*⟩

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

lemma *FnCall-FnCall-eq* [*iff*]: $(FnCall\ F\ Xs = FnCall\ F'\ Xs') \longleftrightarrow (F=F' \wedge Xs=Xs')$
 ⟨*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 :: $exp \Rightarrow int$ **where**
 $discrim\ X = the_elem\ (\bigcup U \in Rep_Exp\ X.\ \{freediscrim\ U\})$

lemma *discrim-respects*: $(\lambda U. \{freediscrim\ U\})$ 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 $V: \bigwedge nat. P1\ (Var\ nat)$

and $P: \bigwedge exp1\ exp2. \llbracket P1\ exp1; P1\ exp2 \rrbracket \implies P1\ (Plus\ exp1\ exp2)$

and $F: \bigwedge nat\ list. P2\ list \implies P1\ (FnCall\ nat\ list)$

and $Nil: P2\ []$

and $Cons: \bigwedge exp\ list. \llbracket P1\ exp; P2\ list \rrbracket \implies P2\ (exp\ \# list)$

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* :: $('a \Rightarrow ('a, 'b)$ *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

$subst-term\ f\ (Var\ a) = f\ a$

| $subst-term\ f\ (App\ b\ ts) = App\ b\ (subst-term-list\ f\ ts)$

| $subst-term-list\ f\ [] = []$

| $subst-term-list\ f\ (t\ \#\ ts) = subst-term\ f\ t\ \# subst-term-list\ f\ ts$

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 $subst-term-list (subst-term f1 \circ f2) ts =$
 $subst-term-list f1 (subst-term-list f2 ts)$
 $\langle proof \rangle$

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 $\forall t \in set\ ts. P t$
 $\langle proof \rangle$

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

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

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

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

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*: $M \in \text{sexp} \implies \text{In0}(M) \in \text{sexp}$
⟨*proof*⟩

lemma *sexp-In1I*: $M \in \text{sexp} \implies \text{In1}(M) \in \text{sexp}$
⟨*proof*⟩

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

lemma *range-Leaf-subset-sexp*: $\text{range}(\text{Leaf}) \leq \text{sexp}$
⟨*proof*⟩

lemma *Scons-D*: $\text{Scons } M \ N \in \text{sexp} \implies M \in \text{sexp} \ \& \ N \in \text{sexp}$
⟨*proof*⟩

lemma *pred-sexp-subset-Sigma*: $\text{pred-sexp} \leq \text{sexp} \times \text{sexp}$
⟨*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*:
 [$M \in \text{sexp}; N \in \text{sexp}$] $\implies (M, \text{Scons } M \ N) \in \text{pred-sexp}$
⟨*proof*⟩

lemma *pred-sexpI2*:

$\llbracket M \in \text{sexp}; N \in \text{sexp} \rrbracket \implies (N, \text{Scons } M N) \in \text{pred-sexp}$
 $\langle \text{proof} \rangle$

lemmas pred-sexp-t1 [simp] = pred-sexpI1 [THEN $r\text{-into-trancl}$]
and pred-sexp-t2 [simp] = pred-sexpI2 [THEN $r\text{-into-trancl}$]

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 :

$\llbracket p \in \text{pred-sexp};$
 $\quad \llbracket M N. \llbracket p = (M, \text{Scons } M N); M \in \text{sexp}; N \in \text{sexp} \rrbracket \implies R;$
 $\quad \llbracket M N. \llbracket p = (N, \text{Scons } M N); M \in \text{sexp}; N \in \text{sexp} \rrbracket \implies R$
 $\rrbracket \implies R$
 $\langle \text{proof} \rangle$

lemma wf-pred-sexp : $\text{wf}(\text{pred-sexp})$
 $\langle \text{proof} \rangle$

lemma $\text{sexp-rec-unfold-lemma}$:

$(\%M. \text{sexp-rec } M c d e) ==$
 $\text{wfrec } \text{pred-sexp } (\%g. \text{sexp-case } c d (\%N1 N2. e N1 N2 (g N1) (g N2)))$
 $\langle \text{proof} \rangle$

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

lemma sexp-rec-Leaf : $\text{sexp-rec } (\text{Leaf } a) c d h = c(a)$
 $\langle \text{proof} \rangle$

lemma sexp-rec-Numb : $\text{sexp-rec } (\text{Numb } k) c d h = d(k)$
 $\langle \text{proof} \rangle$

lemma sexp-rec-Scons : $\llbracket M \in \text{sexp}; N \in \text{sexp} \rrbracket \implies$
 $\text{sexp-rec } (\text{Scons } M N) c d h = h M N (\text{sexp-rec } M c d h) (\text{sexp-rec } N c d h)$
 $\langle \text{proof} \rangle$

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 ∈ A; M ∈ 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 \ (\[]) \text{ and}$$

$$Cons \ (\text{infixr } \# \ 65)$$
definition

$$Nil \quad \quad \quad :: 'a \ list \quad \quad \quad (\[]) \text{ where}$$

$$Nil = Abs-List(NIL)$$
definition

$$Cons \quad \quad \quad :: ['a, 'a \ list] => 'a \ list \quad \quad \quad (\text{infixr } \# \ 65) \text{ where}$$

$$x\#xs = Abs-List(CONS (Leaf x)(Rep-List xs))$$
definition

$$list-rec \quad \quad \quad :: ['a \ list, 'b, ['a, 'a \ list, 'b] => 'b] => 'b \text{ where}$$

$$list-rec \ l \ c \ d =$$

$$List-rec(Rep-List \ l) \ c \ (\%x \ y \ r. \ d(inv \ Leaf \ x)(Abs-List \ y) \ r)$$
definition

$$list-case \quad \quad \quad :: ['b, ['a, 'a \ list] => 'b, 'a \ list] => 'b \text{ where}$$

$$list-case \ a \ f \ xs = list-rec \ xs \ a \ (\%x \ xs \ r. \ f \ x \ xs)$$
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) **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) **where**
Abs-map g M = List-rec M Nil (%N L r. g(N)#r)

definition

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

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

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

lemma *ListI*: $x \in \text{list}(\text{range Leaf}) \implies x \in \text{List}$
<proof>

lemma *ListD*: $x \in \text{List} \implies x \in \text{list}(\text{range Leaf})$
<proof>

lemma *list-unfold*: $\text{list}(A) = \text{usum} \{ \text{Numb}(0) \} (\text{uprod } A (\text{list}(A)))$
<proof>

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

lemma *list-sexp*: $\text{list}(\text{sexp}) \leq \text{sexp}$
<proof>

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

lemma *list-induct*:

[P(Nil);
!!x xs. P(xs) ==> P(x # xs)] ==> P(l)
<proof>

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

lemma *CONS-not-NIL* [*iff*]: *CONS M N \sim = 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 = 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*]: *(CONS K M)=(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*: *CONS M N \in list(A) \implies M \in A & N \in list(A)*
<proof>

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

lemma *not-CONS-self*: *N \in list(A) \implies $\forall M. N \neq CONS M N$*
<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*]: *List-case c h NIL = c*
<proof>

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

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

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

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

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

lemma *pred-sexp-CONS-D*:
 $(CONS M1 M2, N) \in pred-sexp^+ \implies$
 $(M1, N) \in pred-sexp^+ \wedge (M2, N) \in pred-sexp^+$
<proof>

lemma *List-rec-NIL* [*simp*]: *List-rec NIL c h = c*
<proof>

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

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

lemma *List-rec-type*:

$$\llbracket M \in \text{list}(A);$$

$$A \leq \text{sexp};$$

$$c \in C(\text{NIL});$$

$$\bigwedge x y r. \llbracket x \in A; y \in \text{list}(A); r \in C(y) \rrbracket \implies h x y r \in C(\text{CONS } x y)$$

$$\rrbracket \implies \text{List-rec } M c h \in C(M :: 'a \text{ item})$$
 $\langle \text{proof} \rangle$

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

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*: $(\bigwedge x. f(x) \in A) \implies \text{Rep-map } f xs \in \text{list}(A)$
 $\langle \text{proof} \rangle$

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

lemma *Abs-map-CONS* [*simp*]:

$\llbracket M \in \text{sexp}; N \in \text{sexp} \rrbracket \implies \text{Abs-map } g \text{ (CONS } M \ N) = g(M) \# \text{Abs-map } g \ N$
 $\langle \text{proof} \rangle$

lemma *def-list-rec-NilCons*:

$\llbracket \bigwedge xs. f(xs) = \text{list-rec } xs \ c \ h \ \rrbracket$
 $\implies f \ [] = c \wedge f(x\#xs) = h \ x \ xs \ (f \ xs)$
 $\langle \text{proof} \rangle$

lemma *Abs-map-inverse*:

$\llbracket M \in \text{list}(A); A \leq \text{sexp}; \bigwedge z. z \in A \implies f(g(z)) = z \ \rrbracket$
 $\implies \text{Rep-map } f \ (\text{Abs-map } g \ M) = M$
 $\langle \text{proof} \rangle$

Better to have a single theorem with a conjunctive conclusion.

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

lemma *expand-list-case*:

$P(\text{list-case } a \ f \ xs) = ((xs = [] \longrightarrow P \ a) \wedge (\forall y \ ys. xs = y\#ys \longrightarrow P(f \ y \ ys)))$
 $\langle \text{proof} \rangle$

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

lemma *Abs-Rep-map*:

$(\bigwedge x. f(x) \in \text{sexp}) \implies$
 $\text{Abs-map } g \ (\text{Rep-map } f \ xs) = \text{map } (\lambda t. g(f(t))) \ xs$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

<proof>

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

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

lemma *take-take-eq* [*simp*]: $\forall n. \text{take } (\text{take } xs \ n) \ n = \text{take } xs \ n$
<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* :: $('a \Rightarrow \text{nat}) \Rightarrow 'a \ \text{aexp} \Rightarrow \text{nat}$
and *evalb* :: $('a \Rightarrow \text{nat}) \Rightarrow 'a \ \text{bexp} \Rightarrow \text{bool}$

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 \wedge evalb env b2)*
| *evalb env (Neg b) = (\neg evalb env b)*

Substitution on arithmetic and boolean expressions

primrec *subst* :: $('a \Rightarrow 'b \ \text{aexp}) \Rightarrow 'a \ \text{aexp} \Rightarrow 'b \ \text{aexp}$
and *substb* :: $('a \Rightarrow 'b \ \text{aexp}) \Rightarrow 'a \ \text{bexp} \Rightarrow 'b \ \text{bexp}$

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*:

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

and *subst-all-bexp*:

$evalb\ env\ (subst\ s\ b) = evalb\ (\lambda x. evala\ env\ (s\ x))\ b$
{*proof*}

end

8 Infinitely branching trees

theory *Infinitely-Branching-Tree*

imports *Main*

begin

datatype *'a tree* =

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

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

where

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

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

{*proof*}

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

where

$exists-tree\ P\ (Atom\ a) = P\ a$
| $exists-tree\ P\ (Branch\ ts) = (\exists x. exists-tree\ P\ (ts\ x))$

lemma *exists-map*:
 $(\bigwedge x. P x \implies Q (f x)) \implies$
 $\text{exists-tree } P \text{ } ts \implies \text{exists-tree } Q \text{ } (\text{map-tree } f \text{ } ts)$
 $\langle \text{proof} \rangle$

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

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

Multiplication of ordinals

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

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

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

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* \times *brouwer*) *set*
where *brouwer-pred* = $(\bigcup i. \{(m, n). n = Succ \ m \vee (\exists f. n = Lim \ f \wedge m = f \ i)\})$

definition *brouwer-order* :: (*brouwer* \times *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]: $(j, \text{Succ } j) \in \text{brouwer-order}$
<proof>

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

Example of a general function

function *add2* :: *brouwer* \Rightarrow *brouwer* \Rightarrow *brouwer*
where

add2 *i* *Zero* = *i*
| *add2* *i* (*Succ* *j*) = *Succ* (*add2* *i* *j*)
| *add2* *i* (*Lim* *f*) = *Lim* ($\lambda n. \text{add2 } i (f \ n)$)
<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* \Rightarrow *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* :: $('a \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow ('a \Rightarrow 'a)$
where *iter* *f* *n* $\equiv f \ \widehat{\sim} \ n$

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

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

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*: $\{\} \in \sigma\text{-algebra } A$
<proof>

theorem *sigma-algebra-Inter*:
 $(\bigwedge i::\text{nat. } a \ i \in \sigma\text{-algebra } A) \implies (\bigcap i. a \ i) \in \sigma\text{-algebra } A$
<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 [1].

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 $\exists z. r^{**} y' z \wedge r y z$
 <*proof*>

proposition *diamond-rtrancl*:

assumes *diamond* *r*
shows *diamond*(*r***)
 <*proof*>

11.3 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases

K-contractE [*elim!*]: *K* \rightarrow^1 *r*
and *S-contractE* [*elim!*]: *S* \rightarrow^1 *r*
and *Ap-contractE* [*elim!*]: *p* · *q* \rightarrow^1 *r*

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

lemma *I-contract-E* [*iff*]: $\neg I \rightarrow^1 z$
 <*proof*>

lemma *K1-contractD* [*elim!*]: $K \cdot x \rightarrow^1 z \implies (\exists x'. z = K \cdot x' \wedge x \rightarrow^1 x')$
 ⟨*proof*⟩

lemma *Ap-reduce1* [*intro*]: $x \rightarrow y \implies x \cdot z \rightarrow y \cdot z$
 ⟨*proof*⟩

lemma *Ap-reduce2* [*intro*]: $x \rightarrow y \implies z \cdot x \rightarrow z \cdot y$
 ⟨*proof*⟩

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

lemma *not-diamond-contract*: $\neg \text{diamond}(\text{contract1})$
 ⟨*proof*⟩

11.4 Results about Parallel Contraction

Derive a case for each combinator constructor.

inductive-cases

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

declare *parcontract1.intros* [*intro*]

11.5 Basic properties of parallel contraction

The rules below are not essential but make proofs much faster

lemma *K1-parcontractD* [*dest!*]: $K \cdot x \Rightarrow^1 z \implies (\exists x'. z = K \cdot x' \wedge x \Rightarrow^1 x')$
 ⟨*proof*⟩

lemma *S1-parcontractD* [*dest!*]: $S \cdot x \Rightarrow^1 z \implies (\exists x'. z = S \cdot x' \wedge x \Rightarrow^1 x')$
 ⟨*proof*⟩

lemma *S2-parcontractD* [*dest!*]: $S \cdot x \cdot y \Rightarrow^1 z \implies (\exists x' y'. z = S \cdot x' \cdot y' \wedge x \Rightarrow^1 x' \wedge y \Rightarrow^1 y')$
 ⟨*proof*⟩

Church-Rosser property for parallel contraction

proposition *diamond-parcontract*: *diamond parcontract1*
 ⟨*proof*⟩

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

lemma *contract-imp-parcontract*: $x \rightarrow^1 y \implies x \Rightarrow^1 y$
 ⟨*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*: $x \Rightarrow^1 y \implies x \rightarrow y$
<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 \rightarrow 90)*

12.2 The proof system

inductive *thms* :: [*'a pl set, 'a pl*] \Rightarrow *bool* (**infixl** \vdash 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*: $\llbracket H \vdash p \rightarrow q; H \vdash p \rrbracket \implies H \vdash q$

12.3 The semantics

12.3.1 Semantics of propositional logic.

primrec *eval* :: [*'a set, 'a pl*] \Rightarrow *bool* (**-**[**-**] [*100,0*] 100)
 where

$tt[[false]] = False$
 $| tt[[\#v]] = (v \in tt)$
 $| eval-imp: tt[[p \rightarrow q]] = (tt[[p]] \longrightarrow tt[[q]])$

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

primrec $hyps :: ['a\ pl, 'a\ set] \Rightarrow 'a\ pl\ set$
where
 $hyps\ false\ tt = \{\}$
 $| hyps\ (\#v)\ tt = \{if\ v \in tt\ then\ \#v\ else\ \#v \rightarrow false\}$
 $| hyps\ (p \rightarrow q)\ tt = hyps\ p\ tt\ Un\ hyps\ q\ tt$

12.3.2 Logical consequence

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

definition $sat :: ['a\ pl\ set, 'a\ pl] \Rightarrow bool$ (**infixl** \models 50)
where $H \models p = (\forall tt. (\forall q \in H. tt[[q]]) \longrightarrow tt[[p]])$

12.4 Proof theory of propositional logic

lemma $thms-mono$:

assumes $G \subseteq H$ **shows** $thms(G) \leq thms(H)$
 $\langle proof \rangle$

lemma $thms-I$: $H \vdash p \rightarrow p$

— Called I for Identity Combinator, not for Introduction.
 $\langle proof \rangle$

12.4.1 Weakening, left and right

lemma $weaken-left$: $\llbracket G \subseteq H; G \vdash p \rrbracket \Longrightarrow H \vdash p$
— Order of premises is convenient with *THEN*
 $\langle proof \rangle$

lemma $weaken-left-insert$: $G \vdash p \Longrightarrow insert\ a\ G \vdash p$
 $\langle proof \rangle$

lemma $weaken-left-Un1$: $G \vdash p \Longrightarrow G \cup B \vdash p$
 $\langle proof \rangle$

lemma $weaken-left-Un2$: $G \vdash p \Longrightarrow A \cup G \vdash p$
 $\langle proof \rangle$

lemma $weaken-right$: $H \vdash q \Longrightarrow H \vdash p \rightarrow q$
 $\langle proof \rangle$

12.4.2 The deduction theorem

theorem $deduction$: $insert\ p\ H \vdash q \Longrightarrow H \vdash p \rightarrow q$
 $\langle proof \rangle$

12.4.3 The cut rule

lemma *cut*: $insert\ p\ H \vdash q \implies H \vdash p \implies H \vdash q$
<proof>

lemma *thms-falseE*: $H \vdash false \implies H \vdash q$
<proof>

lemma *thms-notE*: $H \vdash p \rightarrow false \implies H \vdash p \implies H \vdash q$
<proof>

12.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: $H \vdash p \implies H \models p$
<proof>

12.5 Completeness

12.5.1 Towards the completeness proof

lemma *false-imp*: $H \vdash p \rightarrow false \implies H \vdash p \rightarrow q$
<proof>

lemma *imp-false*:
 $\llbracket H \vdash p; H \vdash q \rightarrow false \rrbracket \implies H \vdash (p \rightarrow q) \rightarrow false$
<proof>

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

lemma *sat-thms-p*: $\{\} \models p \implies hyps\ p\ tt \vdash p$
— 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*: $H \vdash (p \rightarrow q) \rightarrow ((p \rightarrow false) \rightarrow q) \rightarrow q$
<proof>

lemma *thms-excluded-middle-rule*:
 $\llbracket insert\ p\ H \vdash q; insert\ (p \rightarrow false)\ H \vdash q \rrbracket \implies H \vdash q$
— Hard to prove directly because it requires cuts
<proof>

12.6 Completeness – lemmas for reducing the set of assumptions

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

lemma *hyps-Diff*: $\text{hyps } p \ (t - \{v\}) \subseteq \text{insert } (\#v \rightarrow \text{false}) \ ((\text{hyps } p \ t) - \{\#v\})$
<proof>

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

lemma *hyps-insert*: $\text{hyps } p \ (\text{insert } v \ t) \subseteq \text{insert } (\#v) \ (\text{hyps } p \ t - \{\#v \rightarrow \text{false}\})$
<proof>

Two lemmas for use with *weaken-left*

lemma *insert-Diff-same*: $B - C \subseteq \text{insert } a \ (B - \text{insert } a \ C)$
<proof>

lemma *insert-Diff-subset2*: $\text{insert } a \ (B - \{c\}) - D \subseteq \text{insert } a \ (B - \text{insert } c \ D)$
<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*: $\text{hyps } p \ t \subseteq (\text{UN } v. \{\#v, \#v \rightarrow \text{false}\})$
<proof>

lemma *Diff-weaken-left*: $A \subseteq C \implies A - B \vdash p \implies C - B \vdash p$
<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*:
assumes $\{\} \models p$
shows $\{\} \vdash p$
<proof>

A semantic analogue of the Deduction Theorem

lemma *sat-imp*: $\text{insert } p \ H \models q \implies H \models p \rightarrow q$
<proof>

theorem *completeness*: $\text{finite } H \implies H \models p \implies H \vdash p$
<proof>

theorem *syntax-iff-semantics*: $\text{finite } H \implies (H \vdash p) = (H \models p)$

<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*) -|[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]-> $s1$ |]
 \implies (*IF* e *THEN* $c0$ *ELSE* $c1, s$) -|[eval]-> $s1$

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

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

| *WhileTrue*: [| (e, s) -|[eval]-> ($0, s1$);
 $(c, s1)$ -|[eval]-> $s2$; (*WHILE* e *DO* $c, s2$) -|[eval]-> $s3$ |]
 \implies (*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*: $A \leq B \implies \text{exec}(A) \leq \text{exec}(B)$
<*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*: *single-valued ev* \implies *single-valued(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*] \implies *bool* (**infixl** -|-> 50)
where
esig -|-> $ns == (esig, ns) \in \text{eval}$

| N [intro!]: $(N(n),s) \dashv\vdash (n,s)$

| X [intro!]: $(X(x),s) \dashv\vdash (s(x),s)$

| Op [intro]: $[(e0,s) \dashv\vdash (n0,s0); (e1,s0) \dashv\vdash (n1,s1)]$
 $\implies (Op\ f\ e0\ e1, s) \dashv\vdash (f\ n0\ n1, s1)$

| $valOf$ [intro]: $[(c,s) \dashv\vdash [eval] \rightarrow s0; (e,s0) \dashv\vdash (n,s1)]$
 $\implies (VALOF\ c\ RESULTIS\ e, s) \dashv\vdash (n, s1)$

monos *exec-mono*

inductive-cases

[elim!]: $(N(n),sigma) \dashv\vdash (n',s')$
and [elim!]: $(X(x),sigma) \dashv\vdash (n,s')$
and [elim!]: $(Op\ f\ a1\ a2,sigma) \dashv\vdash (n,s')$
and [elim!]: $(VALOF\ c\ RESULTIS\ e, s) \dashv\vdash (n, s1)$

lemma *var-assign-eval* [intro!]: $(X\ x, s(x:=n)) \dashv\vdash (n, s(x:=n))$
 $\langle proof \rangle$

Make the induction rule look nicer – though *eta-contract* makes the new version look worse than it is...

lemma *split-lemma*: $\{((e,s),(n,s')). P\ e\ s\ n\ s'\} = Collect\ (case-prod\ (\%v.\ case-prod\ (case-prod\ P\ v)))$
 $\langle proof \rangle$

New induction rule. Note the form of the VALOF induction hypothesis

lemma *eval-induct*

[case-names $N\ Op\ valOf$, consumes 1, induct set: *eval*]:
 $[(e,s) \dashv\vdash (n,s');$
 $!!n\ s.\ P\ (N\ n)\ s\ n\ s;$
 $!!s\ x.\ P\ (X\ x)\ s\ (s\ x)\ s;$
 $!!e0\ e1\ f\ n0\ n1\ s\ s0\ s1.$
 $[(e0,s) \dashv\vdash (n0,s0); P\ e0\ s\ n0\ s0;$
 $(e1,s0) \dashv\vdash (n1,s1); P\ e1\ s0\ n1\ s1$
 $]] \implies P\ (Op\ f\ e0\ e1)\ s\ (f\ n0\ n1)\ s1;$
 $!!c\ e\ n\ s\ s0\ s1.$
 $[(c,s) \dashv\vdash [eval\ Int\ \{((e,s),(n,s')). P\ e\ s\ n\ s'\}] \rightarrow s0;$
 $(c,s) \dashv\vdash [eval] \rightarrow s0;$
 $(e,s0) \dashv\vdash (n,s1); P\ e\ s0\ n\ s1\]]$
 $\implies P\ (VALOF\ c\ RESULTIS\ e)\ s\ n\ s1$
 $]] \implies P\ e\ s\ n\ s'$
 $\langle proof \rangle$

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 $(c,s) \text{--}[eval Int \{...\}]\text{--}> s1$ assures us that execution is functional on the argument (c,s) .

lemma *com-Unique*:

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

Expression evaluation is functional, or deterministic

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

lemma *while-true-E*:

$$(c', s) \text{--}[eval]\text{--}> t \implies c' = \text{WHILE } (N \ 0) \ \text{DO } c \implies \text{False}$$

$\langle proof \rangle$

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

lemma *while-if1*:

$$(c',s) \text{--}[eval]\text{--}> t \\ \implies c' = \text{WHILE } e \ \text{DO } c \implies \\ (\text{IF } e \ \text{THEN } c;;c' \ \text{ELSE } \text{SKIP}, s) \text{--}[eval]\text{--}> t$$

$\langle proof \rangle$

lemma *while-if2*:

$$(c',s) \text{--}[eval]\text{--}> t \\ \implies c' = \text{IF } e \ \text{THEN } c;;(\text{WHILE } e \ \text{DO } c) \ \text{ELSE } \text{SKIP} \implies \\ (\text{WHILE } e \ \text{DO } c, s) \text{--}[eval]\text{--}> t$$

$\langle proof \rangle$

theorem *while-if*:

$$((\text{IF } e \ \text{THEN } c;;(\text{WHILE } e \ \text{DO } c) \ \text{ELSE } \text{SKIP}, s) \text{--}[eval]\text{--}> t) = \\ ((\text{WHILE } e \ \text{DO } c, s) \text{--}[eval]\text{--}> t)$$

$\langle proof \rangle$

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

lemma *if-semi1*:

$$(c',s) \text{--}[eval]\text{--}> t \\ \implies c' = (\text{IF } e \ \text{THEN } c1 \ \text{ELSE } c2);;c \implies$$

$(IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c),\ s) -[eval]-> t$
 ⟨proof⟩

lemma *if-semi2*:

$(c',s) -[eval]-> t$
 $==> c' = IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c) ==>$
 $((IF\ e\ THEN\ c1\ ELSE\ c2);;c,\ s) -[eval]-> t$
 ⟨proof⟩

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

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

lemma *valof-valof1*:

$(e',s) -|-> (v,s')$
 $==> e' = VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e) ==>$
 $(VALOF\ c1;;c2\ RESULTIS\ e,\ s) -|-> (v,s')$
 ⟨proof⟩

lemma *valof-valof2*:

$(e',s) -|-> (v,s')$
 $==> e' = VALOF\ c1;;c2\ RESULTIS\ e ==>$
 $(VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e),\ s) -|-> (v,s')$
 ⟨proof⟩

theorem *valof-valof*:

$((VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e),\ s) -|-> (v,s')) =$
 $((VALOF\ c1;;c2\ RESULTIS\ e,\ s) -|-> (v,s'))$
 ⟨proof⟩

13.6 Equivalence of VALOF SKIP RESULTIS e and e

lemma *valof-skip1*:

$(e',s) -|-> (v,s')$
 $==> e' = VALOF\ SKIP\ RESULTIS\ e ==>$
 $(e,\ s) -|-> (v,s')$
 ⟨proof⟩

lemma *valof-skip2*:

$(e,s) -|-> (v,s') ==> (VALOF\ SKIP\ RESULTIS\ e,\ s) -|-> (v,s')$
 ⟨proof⟩

theorem *valof-skip*:

$((VALOF\ SKIP\ RESULTIS\ e,\ s) -|-> (v,s')) = ((e,\ s) -|-> (v,s'))$
 ⟨proof⟩

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

lemma *valof-assign1*:

$$\begin{aligned} & (e',s) \dashv\vdash (v,s'') \\ \implies & e' = \text{VALOF } x:=e \text{ RESULTIS } X x \implies \\ & (\exists s'. (e, s) \dashv\vdash (v,s') \ \& \ (s'' = s'(x:=v))) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *valof-assign2*:

$$\begin{aligned} & (e,s) \dashv\vdash (v,s') \implies (\text{VALOF } x:=e \text{ RESULTIS } X x, s) \dashv\vdash (v,s'(x:=v)) \\ & \langle \text{proof} \rangle \end{aligned}$$

end

References

- [1] J. Camilleri and T. F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, Aug. 1992.