Examples of Inductive and Coinductive Definitions in HOL

Stefan Berghofer
Tobias Nipkow
Lawrence C Paulson
Markus Wenzel

June 21, 2010

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 Defining an Initial Algebra by Quotienting a Free Algebra **7**
2.1 Defining the Free Algebra 8
2.2 Some Functions on the Free Algebra 8
2.2.1 The Set of Nonces 8
2.2.2 The Left Projection 9
2.2.3 The Right Projection 9
2.2.4 The Discriminator for Constructors 9
2.3 The Initial Algebra: A Quotiented Message Type 10
2.3.1 Characteristic Equations for the Abstract Constructors 11
2.4 The Abstract Function to Return the Set of Nonces 11
2.5 The Abstract Function to Return the Left Part 12
2.6 The Abstract Function to Return the Right Part 12
2.7 Injectivity Properties of Some Constructors 13

2.8 The Abstract Discriminator 14

3 Quotienting a Free Algebra Involving Nested Recursion 14
3.1 Defining the Free Algebra 14
3.2 Some Functions on the Free Algebra 15
3.2.1 The Set of Variables 15
3.2.2 Functions for Freeness 16
3.3 The Initial Algebra: A Quotiented Message Type 17
3.4 Every list of abstract expressions can be expressed in terms of a list of concrete expressions 18
3.4.1 Characteristic Equations for the Abstract Constructors 18
3.5 The Abstract Function to Return the Set of Variables 19
3.6 Injectivity Properties of Some Constructors 19
3.7 Injectivity of *FnCall* 20
3.8 The Abstract Discriminator 20

4 Terms over a given alphabet 21

5 Extended List Theory (old) 25

6 Arithmetic and boolean expressions 32

7 Infinitely branching trees 33
7.1 The Brouwer ordinals, as in ZF/Induct/Brouwer.thy. 34
7.2 A WF Ordering for The Brouwer ordinals (Michael Compton) 34

8 Ordinals 35

9 Sigma algebras 37

10 Combinatory Logic example: the Church-Rosser Theorem 37
10.1 Definitions 37
10.2 Reflexive/Transitive closure preserves Church-Rosser property 39
10.3 Non-contraction results 39
10.4 Results about Parallel Contraction 39
10.5 Basic properties of parallel contraction 40

11 Meta-theory of propositional logic 41
11.1 The datatype of propositions 41
11.2 The proof system 41
11.3 The semantics 41
11.3.1 Semantics of propositional logic. 41
11.3.2 Logical consequence 42
11.4 Proof theory of propositional logic 42
11.4.1 Weakening, left and right 42

11.4.2 The deduction theorem 42
11.4.3 The cut rule 42
11.4.4 Soundness of the rules wrt truth-table semantics . . . 42
11.5 Completeness 43
11.5.1 Towards the completeness proof 43
11.6 Completeness – lemmas for reducing the set of assumptions . 43
11.6.1 Completeness theorem 44

12 Mutual Induction via Iteratived Inductive Definitions 44
12.1 Commands 45
12.2 Expressions 46
12.3 Equivalence of IF e THEN c;;(WHILE e DO c) ELSE SKIP and WHILE e DO c 48
12.4 Equivalence of (IF e THEN c1 ELSE c2);;c and IF e THEN (c1;;c) ELSE (c2;;c) 48
12.5 Equivalence of VALOF c1 RESULTIS (VALOF c2 RESULTIS e) and VALOF c1;;c2 RESULTIS e 49
12.6 Equivalence of VALOF SKIP RESULTIS e and e 49
12.7 Equivalence of VALOF x:=e RESULTIS x and e 49

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: $[\![P\ 0;\ \bigwedge nat.\ P\ nat \Longrightarrow P\ (Suc\ nat)]\!] \Longrightarrow P\ nat$, 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 $A\ n \Longrightarrow P\ n$
    and $B\ n \Longrightarrow Q\ n$
⟨*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 $\Longrightarrow$ of the Pure framework. In contrast, local parameters do not require separate $\bigwedge$ 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 $A\ n\ x \Longrightarrow P\ n\ x$
    and $B\ n\ y \Longrightarrow Q\ n\ y$
⟨*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:

⟦⋀*nat*. *P1* (*Foo1 nat*); ⋀*bar*. *P2 bar* ⟹ *P1* (*Foo2 bar*); ⋀*bool*. *P2* (*Bar1 bool*);
 ⋀*bazar*. *P3 bazar* ⟹ *P2* (*Bar2 bazar*); ⋀*foo*. *P1 foo* ⟹ *P3* (*Bazar foo*)⟧
⟹ *P1 foo*
⟦⋀*nat*. *P1* (*Foo1 nat*); ⋀*bar*. *P2 bar* ⟹ *P1* (*Foo2 bar*); ⋀*bool*. *P2* (*Bar1 bool*);
 ⋀*bazar*. *P3 bazar* ⟹ *P2* (*Bar2 bazar*); ⋀*foo*. *P1 foo* ⟹ *P3* (*Bazar foo*)⟧
⟹ *P2 bar*
⟦⋀*nat*. *P1* (*Foo1 nat*); ⋀*bar*. *P2 bar* ⟹ *P1* (*Foo2 bar*); ⋀*bool*. *P2* (*Bar1 bool*);
 ⋀*bazar*. *P3 bazar* ⟹ *P2* (*Bar2 bazar*); ⋀*foo*. *P1 foo* ⟹ *P3* (*Bazar foo*)⟧
⟹ *P3 bazar*

This corresponds to the following basic proof pattern:

lemma
  fixes *foo* :: *foo*
    and *bar* :: *bar*
    and *bazar* :: *bazar*
  shows *P foo*
    and *Q bar*
    and *R bazar*
⟨*proof*⟩

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 ⟹ *P x foo*
  and
    B1 y bar ⟹ *Q1 y bar*
    B2 y bar ⟹ *Q2 y bar*
  and
    C1 z bazar ⟹ *R1 z bazar*
    C2 z bazar ⟹ *R2 z bazar*
    C3 z bazar ⟹ *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 n* ⟹ *Even* (*2* ∗ *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*: *Evn n* ⟹ *Odd* (*Suc n*)
| *succ-Odd*: *Odd n* ⟹ *Evn* (*Suc n*)

lemma
    Evn n ⟹ *P1 n*
    Evn n ⟹ *P2 n*
    Evn n ⟹ *P3 n*
  and
    Odd n ⟹ *Q1 n*
    Odd n ⟹ *Q2 n*
⟨*proof*⟩

end

2 Defining an Initial Algebra by Quotienting a Free Algebra

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

2.1 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*⟩

2.2 Some Functions on the Free Algebra

2.2.1 The Set of Nonces

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

consts
freenonces :: *freemsg* ⇒ *nat set*

primrec
freenonces (*NONCE N*) = {*N*}
freenonces (*MPAIR X Y*) = *freenonces X* ∪ *freenonces Y*

freenonces (*CRYPT K X*) = *freenonces X*
freenonces (*DECRYPT K X*) = *freenonces X*

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

theorem *msgrel-imp-eq-freenonces*: $U \sim V \Longrightarrow$ *freenonces U* = *freenonces V*
⟨*proof*⟩

2.2.2 The Left Projection

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

consts *freeleft* :: *freemsg* ⇒ *freemsg*
primrec
freeleft (*NONCE N*) = *NONCE N*
freeleft (*MPAIR X Y*) = *X*
freeleft (*CRYPT K X*) = *freeleft X*
freeleft (*DECRYPT K X*) = *freeleft X*

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

theorem *msgrel-imp-eqv-freeleft*:
$U \sim V \Longrightarrow$ *freeleft U* $\sim$ *freeleft V*
⟨*proof*⟩

2.2.3 The Right Projection

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

consts *freeright* :: *freemsg* ⇒ *freemsg*
primrec
freeright (*NONCE N*) = *NONCE N*
freeright (*MPAIR X Y*) = *Y*
freeright (*CRYPT K X*) = *freeright X*
freeright (*DECRYPT K X*) = *freeright X*

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

theorem *msgrel-imp-eqv-freeright*:
$U \sim V \Longrightarrow$ *freeright U* $\sim$ *freeright V*
⟨*proof*⟩

2.2.4 The Discriminator for Constructors

A function to distinguish nonces, mpairs and encryptions

consts *freediscrim* :: *freemsg* ⇒ *int*
primrec
freediscrim (*NONCE N*) = *0*
freediscrim (*MPAIR X Y*) = *1*
freediscrim (*CRYPT K X*) = *freediscrim X + 2*
freediscrim (*DECRYPT K X*) = *freediscrim X − 2*

This theorem helps us prove *Nonce N* ≠ *MPair X Y*

theorem *msgrel-imp-eq-freediscrim*:
U ∼ *V* ⟹ *freediscrim U* = *freediscrim V*
⟨*proof*⟩

2.3 The Initial Algebra: A Quotiented Message Type

typedef (*Msg*) *msg* = *UNIV*//*msgrel*
⟨*proof*⟩

The abstract message constructors

definition
Nonce :: *nat* ⇒ *msg* **where**
Nonce N = *Abs-Msg*(*msgrel*``{*NONCE N*})

definition
MPair :: [*msg*,*msg*] ⇒ *msg* **where**
MPair X Y =
Abs-Msg (⋃ *U* ∈ *Rep-Msg X*. ⋃ *V* ∈ *Rep-Msg Y*. *msgrel*``{*MPAIR U V*})

definition
Crypt :: [*nat*,*msg*] ⇒ *msg* **where**
Crypt K X =
Abs-Msg (⋃ *U* ∈ *Rep-Msg X*. *msgrel*``{*CRYPT K U*})

definition
Decrypt :: [*nat*,*msg*] ⇒ *msg* **where**
Decrypt K X =
Abs-Msg (⋃ *U* ∈ *Rep-Msg X*. *msgrel*``{*DECRYPT K U*})

Reduces equality of equivalence classes to the *msgrel* relation: (*msgrel* `` {*x*} = *msgrel* `` {*y*}) = (*x* ∼ *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*]: *msgrel*``{*U*} ∈ *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-iff*, *simp*]

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

2.3.1 Characteristic Equations for the Abstract Constructors

lemma *MPair*: *MPair* (*Abs-Msg*(*msgrel*``{*U*})) (*Abs-Msg*(*msgrel*``{*V*})) =
Abs-Msg (*msgrel*``{*MPAIR U V*})
⟨*proof*⟩

lemma *Crypt*: *Crypt K* (*Abs-Msg*(*msgrel*``{*U*})) = *Abs-Msg* (*msgrel*``{*CRYPT K U*})
⟨*proof*⟩

lemma *Decrypt*:
Decrypt K (*Abs-Msg*(*msgrel*``{*U*})) = *Abs-Msg* (*msgrel*``{*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* = *Abs-Msg*(*msgrel*``{*U*}) ==> *P*) ==> *P*
⟨*proof*⟩

Establishing these two equations is the point of the whole exercise

theorem *CD-eq* [*simp*]: *Crypt K* (*Decrypt K X*) = *X*
⟨*proof*⟩

theorem *DC-eq* [*simp*]: *Decrypt K* (*Crypt K X*) = *X*
⟨*proof*⟩

2.4 The Abstract Function to Return the Set of Nonces

definition
nonces :: *msg* ⇒ *nat set* **where**
nonces X = (⋃ *U* ∈ *Rep-Msg X*. *freenonces U*)

lemma *nonces-congruent*: *freenonces respects msgrel*
⟨*proof*⟩

Now prove the four equations for *nonces*

lemma *nonces-Nonce* [*simp*]: *nonces* (*Nonce N*) = {*N*}
⟨*proof*⟩

lemma *nonces-MPair* [*simp*]: *nonces* (*MPair X Y*) = *nonces X* ∪ *nonces Y*
⟨*proof*⟩

lemma *nonces-Crypt* [*simp*]: *nonces* (*Crypt K X*) = *nonces X*
⟨*proof*⟩

lemma *nonces-Decrypt* [*simp*]: *nonces* (*Decrypt K X*) = *nonces X*
⟨*proof*⟩

2.5 The Abstract Function to Return the Left Part

definition
left :: *msg* ⇒ *msg* **where**
left X = *Abs-Msg* (⋃ *U* ∈ *Rep-Msg X*. *msgrel*``{*freeleft U*})

lemma *left-congruent*: (λ*U*. *msgrel* `` {*freeleft U*}) *respects msgrel*
⟨*proof*⟩

Now prove the four equations for *left*

lemma *left-Nonce* [*simp*]: *left* (*Nonce N*) = *Nonce N*
⟨*proof*⟩

lemma *left-MPair* [*simp*]: *left* (*MPair X Y*) = *X*
⟨*proof*⟩

lemma *left-Crypt* [*simp*]: *left* (*Crypt K X*) = *left X*
⟨*proof*⟩

lemma *left-Decrypt* [*simp*]: *left* (*Decrypt K X*) = *left X*
⟨*proof*⟩

2.6 The Abstract Function to Return the Right Part

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

lemma *right-congruent*: (λ*U*. *msgrel* `` {*freeright U*}) *respects msgrel*
⟨*proof*⟩

Now prove the four equations for *right*

lemma *right-Nonce* [*simp*]: *right* (*Nonce N*) = *Nonce N*
⟨*proof*⟩

lemma *right-MPair* [*simp*]: *right* (*MPair X Y*) = *Y*
⟨*proof*⟩

lemma *right-Crypt* [*simp*]: *right* (*Crypt K X*) = *right X*
⟨*proof*⟩

lemma *right-Decrypt* [*simp*]: *right* (*Decrypt K X*) = *right X*
⟨*proof*⟩

2.7 Injectivity Properties of Some Constructors

lemma *NONCE-imp-eq*: $NONCE\ m \sim NONCE\ n \Longrightarrow m = n$
⟨*proof*⟩

Can also be proved using the function *nonces*

lemma *Nonce-Nonce-eq* [*iff*]: $(Nonce\ m = Nonce\ n) = (m = n)$
⟨*proof*⟩

lemma *MPAIR-imp-eqv-left*: $MPAIR\ X\ Y \sim MPAIR\ X'\ Y' \Longrightarrow X \sim X'$
⟨*proof*⟩

lemma *MPair-imp-eq-left*:
assumes *eq*: $MPair\ X\ Y = MPair\ X'\ Y'$ **shows** $X = X'$
⟨*proof*⟩

lemma *MPAIR-imp-eqv-right*: $MPAIR\ X\ Y \sim MPAIR\ X'\ Y' \Longrightarrow Y \sim Y'$
⟨*proof*⟩

lemma *MPair-imp-eq-right*: $MPair\ X\ Y = MPair\ X'\ Y' \Longrightarrow Y = Y'$
⟨*proof*⟩

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

lemma *NONCE-neqv-MPAIR*: $NONCE\ m \sim MPAIR\ X\ Y \Longrightarrow False$
⟨*proof*⟩

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

Example suggested by a referee

theorem *Crypt-Nonce-neq-Nonce*: $Crypt\ K\ (Nonce\ M) \neq Nonce\ N$
⟨*proof*⟩

...and many similar results

theorem *Crypt2-Nonce-neq-Nonce*: $Crypt\ K\ (Crypt\ K'\ (Nonce\ M)) \neq Nonce\ N$
⟨*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: $\bigwedge X\ Y.\ [\![P\ X;\ P\ Y]\!] \Longrightarrow P\ (MPair\ X\ Y)$

and C: $\bigwedge K\ X.\ P\ X \Longrightarrow P\ (Crypt\ K\ X)$
and D: $\bigwedge K\ X.\ P\ X \Longrightarrow P\ (Decrypt\ K\ X)$
shows $P\ msg$
⟨*proof*⟩

2.8 The Abstract Discriminator

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

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

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

3 Quotienting a Free Algebra Involving Nested Recursion

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

3.1 Defining the Free Algebra

Messages with encryption and decryption as free constructors.

datatype
$freeExp = VAR\ \ nat$
$\quad | \ PLUS\ \ freeExp\ freeExp$
$\quad | \ FNCALL\ \ nat\ freeExp\ list$

The equivalence relation, which makes PLUS associative.

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

3.2 Some Functions on the Free Algebra

3.2.1 The Set of Variables

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

consts
  freevars :: *freeExp* ⇒ *nat set*

freevars-list :: *freeExp list* ⇒ *nat set*

primrec
freevars (*VAR N*) = {*N*}
freevars (*PLUS X Y*) = *freevars X* ∪ *freevars Y*
freevars (*FNCALL F Xs*) = *freevars-list Xs*

freevars-list [] = {}
freevars-list (*X* # *Xs*) = *freevars X* ∪ *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* ∼ *V* ⟹ *freevars U* = *freevars V*
⟨*proof*⟩

3.2.2 Functions for Freeness

A discriminator function to distinguish vars, sums and function calls

consts *freediscrim* :: *freeExp* ⇒ *int*
primrec
freediscrim (*VAR N*) = *0*
freediscrim (*PLUS X Y*) = *1*
freediscrim (*FNCALL F Xs*) = *2*

theorem *exprel-imp-eq-freediscrim*:
U ∼ *V* ⟹ *freediscrim U* = *freediscrim V*
⟨*proof*⟩

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

consts *freefun* :: *freeExp* ⇒ *nat*

primrec
freefun (*VAR N*) = *0*
freefun (*PLUS X Y*) = *0*
freefun (*FNCALL F Xs*) = *F*

theorem *exprel-imp-eq-freefun*:
U ∼ *V* ⟹ *freefun U* = *freefun V*
⟨*proof*⟩

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

consts *freeargs* :: *freeExp* ⇒ *freeExp list*
primrec
freeargs (*VAR N*) = []

freeargs (*PLUS X Y*) = []
freeargs (*FNCALL F Xs*) = *Xs*

theorem *exprel-imp-eqv-freeargs*:
U ∼ *V* ⟹ (*freeargs U*, *freeargs V*) ∈ *listrel exprel*
⟨*proof*⟩

3.3 The Initial Algebra: A Quotiented Message Type

typedef (*Exp*) *exp* = *UNIV*//*exprel*
⟨*proof*⟩

The abstract message constructors

definition
Var :: *nat* ⇒ *exp* **where**
Var N = *Abs-Exp*(*exprel*``{*VAR N*})

definition
Plus :: [*exp*,*exp*] ⇒ *exp* **where**
Plus X Y =
Abs-Exp (⋃ *U* ∈ *Rep-Exp X*. ⋃ *V* ∈ *Rep-Exp Y*. *exprel*``{*PLUS U V*})

definition
FnCall :: [*nat*, *exp list*] ⇒ *exp* **where**
FnCall F Xs =
Abs-Exp (⋃ *Us* ∈ *listset* (*map Rep-Exp Xs*). *exprel* `` {*FNCALL F Us*})

Reduces equality of equivalence classes to the *exprel* relation: (*exprel* `` {*x*} = *exprel* `` {*y*}) = (*x* ∼ *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*} ∈ *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-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* = *Abs-Exp*(*exprel*``{*U*}) ==> *P*) ==> *P*
⟨*proof*⟩

3.4 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*]:
Abs-ExpList (*X*#*Xs*) == *Abs-Exp* (*exprel*``{*X*}) # *Abs-ExpList Xs*
⟨*proof*⟩

lemma *ExpList-rep*: ∃ *Us*. *z* = *Abs-ExpList Us*
⟨*proof*⟩

lemma *eq-Abs-ExpList* [*case-names Abs-ExpList*]:
(!!*Us*. *z* = *Abs-ExpList Us* ==> *P*) ==> *P*
⟨*proof*⟩

3.4.1 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*:
(λ*Us*. *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*}
⟨*proof*⟩

lemma *FnCall*:

FnCall F (Abs-ExpList Us) = Abs-Exp (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*⟩

3.5 The Abstract Function to Return the Set of Variables

definition
vars :: *exp* ⇒ *nat set* **where**
vars X = (⋃ *U* ∈ *Rep-Exp X*. *freevars U*)

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

The extension of the function *vars* to lists

consts *vars-list* :: *exp list* ⇒ *nat set*
primrec
vars-list [] = {}
vars-list(*E*#*Es*) = *vars E* ∪ *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* ∪ *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* ∪ *vars-list Xs*
⟨*proof*⟩

3.6 Injectivity Properties of Some Constructors

lemma *VAR-imp-eq*: *VAR m* ∼ *VAR n* ⟹ *m* = *n*
⟨*proof*⟩

Can also be proved using the function *vars*

lemma *Var-Var-eq* [*iff*]: (*Var m* = *Var n*) = (*m* = *n*)
⟨*proof*⟩

lemma *VAR-neqv-PLUS*: *VAR m* ∼ *PLUS X Y* ⟹ *False*
⟨*proof*⟩

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

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

3.7 Injectivity of *FnCall*

definition
fun :: *exp* ⇒ *nat* **where**
fun X = *contents* (⋃ *U* ∈ *Rep-Exp X*. {*freefun U*})

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

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

definition
args :: *exp* ⇒ *exp list* **where**
args X = *contents* (⋃ *U* ∈ *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*) ∈ *listrel exprel* ⟹ *Abs-ExpList* (*y*) = *Abs-ExpList* (*z*)
⟨*proof*⟩

lemma *args-respects*: (%*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′*) = (*F*=*F′* & *Xs*=*Xs′*)
⟨*proof*⟩

3.8 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 = *contents* (⋃ *U* ∈ *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*: $\bigwedge exp1\ exp2.\ [\![P1\ exp1;\ P1\ exp2]\!] \Longrightarrow P1\ (Plus\ exp1\ exp2)$
    and *F*: $\bigwedge nat\ list.\ P2\ list \Longrightarrow P1\ (FnCall\ nat\ list)$
    and *Nil*: *P2* []
    and *Cons*: $\bigwedge exp\ list.\ [\![P1\ exp;\ P2\ list]\!] \Longrightarrow P2\ (exp\ \#\ list)$
  shows *P1 exp* **and** *P2 list*
⟨*proof*⟩

end

4 Terms over a given alphabet

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

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

Substitution function on terms

consts
  subst-term :: $('a \Rightarrow ('a, 'b)\ term) \Rightarrow ('a, 'b)\ term \Rightarrow ('a, 'b)\ term$
  subst-term-list ::
    $('a \Rightarrow ('a, 'b)\ term) \Rightarrow ('a, 'b)\ term\ list \Rightarrow ('a, 'b)\ term\ list$

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

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

A simple theorem about composition of substitutions

lemma *subst-comp*:
  subst-term (*subst-term f1* ∘ *f2*) *t* =
    subst-term f1 (*subst-term f2 t*)
and *subst-term-list* (*subst-term f1* ∘ *f2*) *ts* =
    subst-term-list f1 (*subst-term-list f2 ts*)
  ⟨*proof*⟩

Alternative induction rule

lemma
  assumes *var*: !!*v*. *P* (*Var v*)
    and *app*: !!*f ts*. *list-all P ts* ==> *P* (*App f ts*)
  shows *term-induct2*: *P t*
    and *list-all P ts*
  ⟨*proof*⟩

end

theory *Sexp* **imports** *Main* **begin**

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

inductive-set
  sexp :: *'a item set*
  where
    LeafI: *Leaf*(*a*) ∈ *sexp*
  | *NumbI*: *Numb*(*i*) ∈ *sexp*
  | *SconsI*: [| *M* ∈ *sexp*; *N* ∈ *sexp* |] ==> *Scons M N* ∈ *sexp*

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

definition
  pred-sexp :: ('*a item* * '*a item*)*set* **where**
    pred-sexp = (⋃*M* ∈ *sexp*. ⋃*N* ∈ *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* **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*:
[| $M \in sexp$; $N \in sexp$ |] ==> $(N, Scons\ M\ N) \in pred\text{-}sexp$
⟨*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*:
[| $p \in pred\text{-}sexp$;
!!$M\ N$. [| $p = (M, Scons\ M\ N)$; $M \in sexp$; $N \in sexp$ |] ==> R;
!!$M\ N$. [| $p = (N, Scons\ M\ N)$; $M \in sexp$; $N \in sexp$ |] ==> R
|] ==> R
⟨*proof*⟩

lemma *wf-pred-sexp*: $wf(pred\text{-}sexp)$
⟨*proof*⟩

lemma *sexp-rec-unfold-lemma*:
$(\%M.\ sexp\text{-}rec\ M\ c\ d\ e) ==$
wfrec pred-sexp $(\%g.\ sexp\text{-}case\ c\ d\ (\%N1\ N2.\ e\ N1\ N2\ (g\ N1)\ (g\ N2)))$
⟨*proof*⟩

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

lemma *sexp-rec-Leaf*: $sexp\text{-}rec\ (Leaf\ a)\ c\ d\ h = c(a)$
⟨*proof*⟩

lemma *sexp-rec-Numb*: $sexp\text{-}rec\ (Numb\ k)\ c\ d\ h = d(k)$
⟨*proof*⟩

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

end

5 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*

typedef (*List*)
'a list = *list*(*range Leaf*) :: *'a item set*
⟨*proof*⟩

abbreviation *Case* == *Datatype.Case*
abbreviation *Split* == *Datatype.Split*

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

definition
List-rec :: [*'a item*, *'b*, [*'a item*, *'a item*, *'b*]=>*'b*] => *'b* **where**
List-rec M c d = *wfrec* (*pred-sexp*^+)

$(\%g.\ List\text{-}case\ c\ (\%x\ y.\ d\ x\ y\ (g\ y)))\ M$

no-translations
$[x,\ xs] == x\#[xs]$
$[x] == x\#[]$
no-notation
$Nil\ \ ([])$ **and**
$Cons$ (**infixr** $\#\ 65$)

definition
$Nil \quad :: 'a\ list \qquad ([])$ **where**
$Nil\ =\ Abs\text{-}List(NIL)$

definition
$Cons \quad :: ['a,\ 'a\ list] => 'a\ list \qquad$ (**infixr** $\#\ 65$) **where**
$x\#xs\ =\ Abs\text{-}List(CONS\ (Leaf\ x)(Rep\text{-}List\ xs))$

definition

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

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

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

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

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)

consts *take* :: ['a list,nat] => 'a list
primrec
take-0: *take xs 0 = []*
take-Suc: *take xs (Suc n) = list-case [] (%x l. x # take l n) xs*

lemma *ListI*: *x : list (range Leaf) ==> x : List*
⟨*proof*⟩

lemma *ListD*: *x : List ==> x : list (range Leaf)*
⟨*proof*⟩

lemma *list-unfold*: *list(A) = usum {Numb(0)} (uprod A (list(A)))*
⟨*proof*⟩

lemma *list-mono*: *A<=B ==> list(A) <= list(B)*
⟨*proof*⟩

lemma *list-sexp*: *list(sexp) <= 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* ~= *NIL*
⟨*proof*⟩

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

lemma *Cons-not-Nil* [*iff*]: *x # xs* ~= *Nil*
⟨*proof*⟩

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

lemma *CONS-CONS-eq* [*iff*]: (*CONS K M*)=(*CONS L N*) = (*K*=*L* & *M*=*N*)
⟨*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*, *standard*]

lemma *CONS-D*: *CONS M N*: *list*(*A*) ==> *M*: *A* & *N*: *list*(*A*)
⟨*proof*⟩

lemma *sexp-CONS-D*: *CONS M N*: *sexp* ==> *M*: *sexp* & *N*: *sexp*
⟨*proof*⟩

lemma *not-CONS-self*: *N*: *list*(*A*) ==> !*M*. *N* ~= *CONS M N*
⟨*proof*⟩

lemma *not-Cons-self2*: $\forall x.\ l \mathrel{\sim=} x\#l$
⟨*proof*⟩

lemma *neq-Nil-conv2*: $(xs \mathrel{\sim=} []) = (\exists y\ ys.\ xs = y\#ys)$
⟨*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*:
(%*M*. *List-rec M c d*) ==
wfrec (*pred-sexp*^+) (%*g*. *List-case c* (%*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*],
standard]

lemma *pred-sexp-CONS-I1*:
[| *M*: *sexp*; *N*: *sexp* |] ==> (*M*, *CONS M N*) : *pred-sexp*^+
⟨*proof*⟩

lemma *pred-sexp-CONS-I2*:
[| *M*: *sexp*; *N*: *sexp* |] ==> (*N*, *CONS M N*) : *pred-sexp*^+
⟨*proof*⟩

lemma *pred-sexp-CONS-D*:
(*CONS M1 M2*, *N*) : *pred-sexp*^+ ==>
(*M1*,*N*) : *pred-sexp*^+ & (*M2*,*N*) : *pred-sexp*^+
⟨*proof*⟩

lemma *List-rec-NIL* [*simp*]: *List-rec NIL c h = c*
⟨*proof*⟩

lemma *List-rec-CONS* [*simp*]:
[| *M*: *sexp*; *N*: *sexp* |]
==> *List-rec* (*CONS M N*) *c h* = *h M N* (*List-rec N c h*)
⟨*proof*⟩

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

lemma *list-rec-Nil* [*simp*]: *list-rec Nil c h = c*
⟨*proof*⟩

lemma *list-rec-Cons* [*simp*]: *list-rec* (*a*#*l*) *c h* = *h a l* (*list-rec l c h*)
⟨*proof*⟩

lemma *List-rec-type*:
[| *M*: *list*(*A*);
A<=*sexp*;
c: *C*(*NIL*);
!!*x y r*. [| *x*: *A*; *y*: *list*(*A*); *r*: *C*(*y*) |] ==> *h x y r*: *C*(*CONS x y*)
|] ==> *List-rec M c h* : *C*(*M* :: *'a item*)
⟨*proof*⟩

lemma *Rep-map-Nil* [*simp*]: *Rep-map f Nil = NIL*
⟨*proof*⟩

lemma *Rep-map-Cons* [*simp*]:
Rep-map f(*x*#*xs*) = *CONS*(*f x*)(*Rep-map f xs*)
⟨*proof*⟩

lemma *Rep-map-type*: (!!*x*. *f*(*x*): *A*) ==> *Rep-map f xs*: *list*(*A*)
⟨*proof*⟩

lemma *Abs-map-NIL* [*simp*]: *Abs-map g NIL = Nil*
⟨*proof*⟩

lemma *Abs-map-CONS* [*simp*]:
[| M: *sexp*; N: *sexp* |] ==> *Abs-map g* (*CONS M N*) = $g(M)$ # *Abs-map g N*
⟨*proof*⟩

lemma *def-list-rec-NilCons*:
[| !!xs. $f(xs)$ = *list-rec xs c h* |]
==> f [] = c & $f(x\#xs)$ = h x xs (f xs)
⟨*proof*⟩

lemma *Abs-map-inverse*:
[| M: *list*(A); A<=*sexp*; !!z. z: A ==> $f(g(z))$ = z |]
==> *Rep-map f* (*Abs-map g 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(*list-case a f xs*) = ((xs=[] --> P a) & (!y ys. xs=y#ys --> P(f y ys)))
⟨*proof*⟩

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

lemma *Abs-Rep-map*:
(!!x. $f(x)$: *sexp*) ==>
Abs-map g (*Rep-map f xs*) = *map* (%t. $g(f(t))$) xs
⟨*proof*⟩

lemma *map-ident* [*simp*]: *map*(%x. x)(xs) = xs
⟨*proof*⟩

lemma *map-compose*: *map*(f *o* g)(xs) = *map* f (*map* g xs)
⟨*proof*⟩

lemma *take-Suc1* [*simp*]: *take* [] (*Suc x*) = []
⟨*proof*⟩

lemma *take-Suc2* [*simp*]: *take*(*a*#*xs*)(*Suc x*) = *a*#*take xs x*
⟨*proof*⟩

lemma *take-Nil* [*simp*]: *take* [] *n* = []
⟨*proof*⟩

lemma *take-take-eq* [*simp*]: ∀ *n*. *take* (*take xs n*) *n* = *take xs n*
⟨*proof*⟩

end

6 Arithmetic and boolean expressions

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

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

Evaluation of arithmetic and boolean expressions

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

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

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

Substitution on arithmetic and boolean expressions

consts

substa :: (*'a* => *'b aexp*) => *'a aexp* => *'b aexp*
substb :: (*'a* => *'b aexp*) => *'a bexp* => *'b bexp*

primrec
substa f (*IF b a1 a2*) = *IF* (*substb f b*) (*substa f a1*) (*substa f a2*)
substa f (*Sum a1 a2*) = *Sum* (*substa f a1*) (*substa f a2*)
substa f (*Diff a1 a2*) = *Diff* (*substa f a1*) (*substa f a2*)
substa f (*Var v*) = *f v*
substa f (*Num n*) = *Num n*

substb f (*Less a1 a2*) = *Less* (*substa f a1*) (*substa f a2*)
substb f (*And b1 b2*) = *And* (*substb f b1*) (*substb f b2*)
substb f (*Neg b*) = *Neg* (*substb f b*)

lemma *subst1-aexp*:
evala env (*substa* (*Var* (*v* := *a'*)) *a*) = *evala* (*env* (*v* := *evala env a'*)) *a*
and *subst1-bexp*:
evalb env (*substb* (*Var* (*v* := *a'*)) *b*) = *evalb* (*env* (*v* := *evala env a'*)) *b*
— one variable
⟨*proof*⟩

lemma *subst-all-aexp*:
evala env (*substa s a*) = *evala* (λ*x*. *evala env* (*s x*)) *a*
and *subst-all-bexp*:
evalb env (*substb s b*) = *evalb* (λ*x*. *evala env* (*s x*)) *b*
⟨*proof*⟩

end

7 Infinitely branching trees

theory *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* (λ*x*. *map-tree f* (*ts x*))

lemma *tree-map-compose*: *map-tree g* (*map-tree f t*) = *map-tree* (*g* ∘ *f*) *t*
⟨*proof*⟩

primrec
exists-tree :: ('*a* => *bool*) => '*a tree* => *bool*
where
exists-tree P (*Atom a*) = *P a*
| *exists-tree P* (*Branch ts*) = (∃ *x*. *exists-tree P* (*ts x*))

lemma *exists-map*:
(!!*x*. *P x* ==> *Q* (*f x*)) ==>
exists-tree P ts ==> *exists-tree Q* (*map-tree f ts*)
⟨*proof*⟩

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

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

Addition of ordinals

primrec
add :: [*brouwer*,*brouwer*] => *brouwer*
where
add i Zero = *i*
| *add i* (*Succ j*) = *Succ* (*add i j*)
| *add i* (*Lim f*) = *Lim* (%*n*. *add i* (*f n*))

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

Multiplication of ordinals

primrec
mult :: [*brouwer*,*brouwer*] => *brouwer*
where
mult i Zero = *Zero*
| *mult i* (*Succ j*) = *add* (*mult i j*) *i*
| *mult i* (*Lim f*) = *Lim* (%*n*. *mult i* (*f n*))

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

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

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

7.2 A WF Ordering for The Brouwer ordinals (Michael Compton)

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 = (⋃ *i*. {(*m*,*n*). *n* = *Succ m* ∨ (*EX f*. *n* = *Lim f* & *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* (λ *n*. *add2* (*i*, (*f n*)))
⟨*proof*⟩
termination ⟨*proof*⟩

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

end

8 Ordinals

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

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

datatype *ordinal* =
Zero
| *Succ ordinal*
| *Limit nat* => *ordinal*

consts
pred :: *ordinal* => *nat* => *ordinal option*
primrec
pred Zero n = *None*
pred (*Succ a*) *n* = *Some a*
pred (*Limit f*) *n* = *Some* (*f n*)

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

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

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

consts
cantor :: *ordinal* => *ordinal* => *ordinal*
primrec
cantor a Zero = *Succ a*
cantor a (*Succ b*) = ⨆(λ*x*. *cantor x b*) *a*
cantor a (*Limit f*) = *Limit* (λ*n*. *cantor a* (*f n*))

consts
Nabla :: (*ordinal* => *ordinal*) => (*ordinal* => *ordinal*) (∇)
primrec
∇*f Zero* = *f Zero*
∇*f* (*Succ a*) = *f* (*Succ* (∇*f a*))
∇*f* (*Limit h*) = *Limit* (λ*n*. ∇*f* (*h n*))

definition
deriv :: (*ordinal* => *ordinal*) => (*ordinal* => *ordinal*) **where**
deriv f = ∇(⨆*f*)

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

definition *veb a* = *veblen a Zero*
definition ε_0 = *veb Zero*
definition Γ_0 = *Limit* (λ*n*. *iter veb n Zero*)

end

9 Sigma algebras

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

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

inductive-set
  σ-*algebra* :: $'a$ *set set* => $'a$ *set set*
  for A :: $'a$ *set set*
  where
    basic: $a \in A$ ==> $a \in \sigma$-*algebra* A
  | *UNIV*: *UNIV* $\in \sigma$-*algebra* A
  | *complement*: $a \in \sigma$-*algebra* A ==> $-a \in \sigma$-*algebra* A
  | *Union*: (!!i::*nat*. $a\ i \in \sigma$-*algebra* A) ==> $(\bigcup i.\ a\ i) \in \sigma$-*algebra* A

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

theorem *sigma-algebra-empty*: $\{\} \in \sigma$-*algebra* A
⟨*proof*⟩

theorem *sigma-algebra-Inter*:
  (!!i::*nat*. $a\ i \in \sigma$-*algebra* A) ==> $(\bigcap i.\ a\ i) \in \sigma$-*algebra* A
⟨*proof*⟩

end

10 Combinatory Logic example: the Church-Rosser Theorem

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

Curiously, combinators do not include free variables.

Example taken from [**?**].

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

10.1 Definitions

Datatype definition of combinators S and K.

datatype *comb* = *K*

| *S*
| *Ap comb comb* (**infixl** *##* *90*)

notation (*xsymbols*)
Ap (**infixl** · *90*)

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

inductive-set
contract :: (*comb*∗*comb*) *set*
and *contract-rel1* :: [*comb*,*comb*] => *bool* (**infixl** *−1−>* *50*)
where
x −1−> y == (*x*,*y*) ∈ *contract*
| *K*: *K##x##y −1−> x*
| *S*: *S##x##y##z −1−> (x##z)##(y##z)*
| *Ap1*: *x−1−>y* ==> *x##z −1−> y##z*
| *Ap2*: *x−1−>y* ==> *z##x −1−> z##y*

abbreviation
contract-rel :: [*comb*,*comb*] => *bool* (**infixl** *−−−>* *50*) **where**
x −−−> y == (*x*,*y*) ∈ *contract*^∗

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

inductive-set
parcontract :: (*comb*∗*comb*) *set*
and *parcontract-rel1* :: [*comb*,*comb*] => *bool* (**infixl** *=1=>* *50*)
where
x =1=> y == (*x*,*y*) ∈ *parcontract*
| *refl*: *x =1=> x*
| *K*: *K##x##y =1=> x*
| *S*: *S##x##y##z =1=> (x##z)##(y##z)*
| *Ap*: [| *x=1=>y*; *z=1=>w* |] ==> *x##z =1=> y##w*

abbreviation
parcontract-rel :: [*comb*,*comb*] => *bool* (**infixl** *===>* *50*) **where**
x ===> y == (*x*,*y*) ∈ *parcontract*^∗

Misc definitions.

definition
I :: *comb* **where**
I = *S##K##K*

definition
diamond :: (*'a* ∗ *'a*)*set* => *bool* **where**
— confluence; Lambda/Commutation treats this more abstractly
diamond(*r*) = (∀ *x y*. (*x*,*y*) ∈ *r* −−>
(∀ *y'*. (*x*,*y'*) ∈ *r* −−>
(∃ *z*. (*y*,*z*) ∈ *r* & (*y'*,*z*) ∈ *r*)))

10.2 Reflexive/Transitive closure preserves Church-Rosser property

So does the Transitive closure, with a similar proof

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

lemma *diamond-strip-lemmaE* [*rule-format*]:
[| *diamond*(*r*); (*x*,*y*) ∈ *r*^* |] ==>
∀ *y'*. (*x*,*y'*) ∈ *r* −−> (∃ *z*. (*y'*,*z*) ∈ *r*^* & (*y*,*z*) ∈ *r*)
⟨*proof*⟩

lemma *diamond-rtrancl*: *diamond*(*r*) ==> *diamond*(*r*^*)
⟨*proof*⟩

10.3 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases
K-contractE [*elim*!]: *K* −1−> *r*
and *S-contractE* [*elim*!]: *S* −1−> *r*
and *Ap-contractE* [*elim*!]: *p*##*q* −1−> *r*

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

lemma *I-contract-E* [*elim*!]: *I* −1−> *z* ==> *P*
⟨*proof*⟩

lemma *K1-contractD* [*elim*!]: *K*##*x* −1−> *z* ==> (∃ *x'*. *z* = *K*##*x'* & *x* −1−> *x'*)
⟨*proof*⟩

lemma *Ap-reduce1* [*intro*]: *x* −−−> *y* ==> *x*##*z* −−−> *y*##*z*
⟨*proof*⟩

lemma *Ap-reduce2* [*intro*]: *x* −−−> *y* ==> *z*##*x* −−−> *z*##*y*
⟨*proof*⟩

Counterexample to the diamond property for *x* −1−> *y*

lemma *not-diamond-contract*: ~ *diamond*(*contract*)
⟨*proof*⟩

10.4 Results about Parallel Contraction

Derive a case for each combinator constructor.

inductive-cases

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

declare *parcontract.intros* [*intro*]

10.5 Basic properties of parallel contraction

lemma *K1-parcontractD* [*dest*!]: $K\#\#x$ =*1*=> z ==> $(\exists x'.\ z = K\#\#x' \ \&\ x$ =*1*=> $x')$
⟨*proof*⟩

lemma *S1-parcontractD* [*dest*!]: $S\#\#x$ =*1*=> z ==> $(\exists x'.\ z = S\#\#x' \ \&\ x$ =*1*=> $x')$
⟨*proof*⟩

lemma *S2-parcontractD* [*dest*!]:
$S\#\#x\#\#y$ =*1*=> z ==> $(\exists x'\ y'.\ z = S\#\#x'\#\#y' \ \&\ x$ =*1*=> $x' \ \&\ y$ =*1*=> $y')$
⟨*proof*⟩

The rules above are not essential but make proofs much faster

Church-Rosser property for parallel contraction

lemma *diamond-parcontract*: *diamond parcontract*
⟨*proof*⟩

Equivalence of p −−−> q and p ===> q.

lemma *contract-subset-parcontract*: *contract* <= *parcontract*
⟨*proof*⟩

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

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

lemma *reduce-I*: $I\#\#x$ −−−> x
⟨*proof*⟩

lemma *parcontract-subset-reduce*: *parcontract* <= *contract*^*
⟨*proof*⟩

lemma *reduce-eq-parreduce*: *contract*^* = *parcontract*^*
⟨*proof*⟩

theorem *diamond-reduce*: *diamond*(*contract*^*)
⟨*proof*⟩

end

11 Meta-theory of propositional logic

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

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

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

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

11.1 The datatype of propositions

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

11.2 The proof system

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

11.3 The semantics

11.3.1 Semantics of propositional logic.

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

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

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

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

primrec

hyps false tt = {}
hyps (#*v*) *tt* = {*if v* ∈ *tt then* #*v else* #*v*−>*false*}
hyps (*p*−>*q*) *tt* = *hyps p tt Un hyps q tt*

11.3.2 Logical consequence

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

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

11.4 Proof theory of propositional logic

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

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

11.4.1 Weakening, left and right

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

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

lemmas *weaken-left-Un1* = *Un-upper1* [*THEN weaken-left*]
lemmas *weaken-left-Un2* = *Un-upper2* [*THEN weaken-left*]

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

11.4.2 The deduction theorem

theorem *deduction*: *insert p H* |− *q* ==> *H* |− *p*−>*q*
⟨*proof*⟩

11.4.3 The cut rule

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

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

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

11.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: *H* |− *p* ==> *H* |= *p*

⟨*proof*⟩

11.5 Completeness

11.5.1 Towards the completeness proof

lemma *false-imp*: $H \mid\!- \; p->false \Longrightarrow H \mid\!- \; p->q$
⟨*proof*⟩

lemma *imp-false*:
$[\!| \; H \mid\!- \; p; \;\; H \mid\!- \; q->false \; |\!] \Longrightarrow H \mid\!- \; (p->q)->false$
⟨*proof*⟩

lemma *hyps-thms-if*: *hyps p tt* $\mid\!-$ (*if tt*[[*p*]] *then p else p*−>*false*)
— Typical example of strengthening the induction statement.
⟨*proof*⟩

lemma *sat-thms-p*: $\{\} \models p \Longrightarrow$ *hyps p tt* $\mid\!-$ *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*: $H \mid\!- \; (p->q) \; -> \; ((p->false)->q) \; -> \; q$
⟨*proof*⟩

lemma *thms-excluded-middle-rule*:
$[\!| \; insert \; p \; H \mid\!- \; q; \;\; insert \; (p->false) \; H \mid\!- \; q \; |\!] \Longrightarrow H \mid\!- \; q$
— Hard to prove directly because it requires cuts
⟨*proof*⟩

11.6 Completeness – lemmas for reducing the set of assumptions

For the case *hyps p t* − *insert* #*v Y* $\mid\!-$ *p* we also have *hyps p t* − {#*v*} ⊆ *hyps p* (*t* − {*v*}).

lemma *hyps-Diff*: *hyps p* (*t*−{*v*}) <= *insert* (#*v*−>*false*) ((*hyps p t*)−{#*v*})
⟨*proof*⟩

For the case *hyps p t* − *insert* (#*v* −> *Fls*) *Y* $\mid\!-$ *p* we also have *hyps p t* − {#*v* −> *Fls*} ⊆ *hyps p* (*insert v t*).

lemma *hyps-insert*: *hyps p* (*insert v t*) <= *insert* (#*v*) (*hyps p t*−{#*v*−>*false*})
⟨*proof*⟩

Two lemmas for use with *weaken-left*

lemma *insert-Diff-same*: *B−C <= insert a (B−insert a C)*
⟨*proof*⟩

lemma *insert-Diff-subset2*: *insert a (B−{c}) − D <= insert a (B−insert c D)*
⟨*proof*⟩

The set *hyps p t* is finite, and elements have the form *#v* or *#v −> Fls*.

lemma *hyps-finite*: *finite*(*hyps p t*)
⟨*proof*⟩

lemma *hyps-subset*: *hyps p t <= (UN v. {#v, #v−>false})*
⟨*proof*⟩

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

11.6.1 Completeness theorem

Induction on the finite set of assumptions *hyps p t0*. We may repeatedly subtract assumptions until none are left!

lemma *completeness-0-lemma*:
{} |= p ==> ∀ t. hyps p t − hyps p t0 |− p
⟨*proof*⟩

The base case for completeness

lemma *completeness-0*: *{} |= p ==> {} |− p*
⟨*proof*⟩

A semantic analogue of the Deduction Theorem

lemma *sat-imp*: *insert p H |= q ==> H |= p−>q*
⟨*proof*⟩

theorem *completeness*: *finite H ==> H |= p ==> H |− p*
⟨*proof*⟩

theorem *syntax-iff-semantics*: *finite H ==> (H |− p) = (H |= p)*
⟨*proof*⟩

end

12 Mutual Induction via Iteratived Inductive Definitions

theory *Com* **imports** *Main* **begin**

typedecl *loc*
types *state = loc => nat*

datatype
exp = *N nat*
| *X loc*
| *Op nat => nat => nat exp exp*
| *valOf com exp* (*VALOF - RESULTIS - 60*)
and
com = *SKIP*
| *Assign loc exp* (**infixl** := *60*)
| *Semi com com* (-;;- [*60*, *60*] *60*)
| *Cond exp com com* (*IF - THEN - ELSE - 60*)
| *While exp com* (*WHILE - DO - 60*)

12.1 Commands

Execution of commands

abbreviation (*input*)
generic-rel (-/ −|[-]−> - [*50*,*0*,*50*] *50*) **where**
esig −|[*eval*]−> *ns* == (*esig*,*ns*) ∈ *eval*

Command execution. Natural numbers represent Booleans: 0=True, 1=False

inductive-set
exec :: ((*exp*∗*state*) ∗ (*nat*∗*state*)) *set* => ((*com*∗*state*)∗*state*)*set*
and *exec-rel* :: *com* ∗ *state* => ((*exp*∗*state*) ∗ (*nat*∗*state*)) *set* => *state* => *bool*
(-/ −[-]−> - [*50*,*0*,*50*] *50*)
for *eval* :: ((*exp*∗*state*) ∗ (*nat*∗*state*)) *set*
where
csig −[*eval*]−> *s* == (*csig*,*s*) ∈ *exec eval*

| *Skip*: (*SKIP*,*s*) −[*eval*]−> *s*

| *Assign*: (*e*,*s*) −|[*eval*]−> (*v*,*s′*) ==> (*x* := *e*, *s*) −[*eval*]−> *s′*(*x*:=*v*)

| *Semi*: [| (*c0*,*s*) −[*eval*]−> *s2*; (*c1*,*s2*) −[*eval*]−> *s1* |]
==> (*c0* ;; *c1*, *s*) −[*eval*]−> *s1*

| *IfTrue*: [| (*e*,*s*) −|[*eval*]−> (*0*,*s′*); (*c0*,*s′*) −[*eval*]−> *s1* |]
==> (*IF e THEN c0 ELSE c1*, *s*) −[*eval*]−> *s1*

| *IfFalse*: [| (*e*,*s*) −|[*eval*]−> (*Suc 0*, *s′*); (*c1*,*s′*) −[*eval*]−> *s1* |]
==> (*IF e THEN c0 ELSE c1*, *s*) −[*eval*]−> *s1*

| *WhileFalse*: (*e*,*s*) −|[*eval*]−> (*Suc 0*, *s1*)
==> (*WHILE e DO c*, *s*) −[*eval*]−> *s1*

| *WhileTrue*: [| (*e*,*s*) −|[*eval*]−> (*0*,*s1*);
(*c*,*s1*) −[*eval*]−> *s2*; (*WHILE e DO c*, *s2*) −[*eval*]−> *s3* |]
==> (*WHILE e DO c*, *s*) −[*eval*]−> *s3*

declare *exec.intros* [*intro*]

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

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

lemma *exec-mono*: *A*<=*B* ==> *exec*(*A*) <= *exec*(*B*)
⟨*proof*⟩

lemma [*pred-set-conv*]:
((λ*x x′ y y′*. ((*x*, *x′*), (*y*, *y′*)) ∈ *R*) <= (λ*x x′ y y′*. ((*x*, *x′*), (*y*, *y′*)) ∈ *S*)) = (*R* <= *S*)
⟨*proof*⟩

lemma [*pred-set-conv*]:
((λ*x x′ y*. ((*x*, *x′*), *y*) ∈ *R*) <= (λ*x x′ y*. ((*x*, *x′*), *y*) ∈ *S*)) = (*R* <= *S*)
⟨*proof*⟩

Command execution is functional (deterministic) provided evaluation is

theorem *single-valued-exec*: *single-valued ev* ==> *single-valued*(*exec ev*)
⟨*proof*⟩

12.2 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*:
$\{((e,s),(n,s')).\ P\ e\ s\ n\ s'\} = Collect\ (split\ (\%v.\ split\ (split\ P\ v)))$
⟨*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*]:
$[|\ (e,s) -|-> (n,s');$
$!!n\ s.\ P\ (N\ n)\ s\ n\ s;$
$!!s\ x.\ P\ (X\ x)\ s\ (s\ x)\ s;$
$!!e0\ e1\ f\ n0\ n1\ s\ s0\ s1.$
$[|\ (e0,s) -|-> (n0,s0);\ P\ e0\ s\ n0\ s0;$
$(e1,s0) -|-> (n1,s1);\ P\ e1\ s0\ n1\ s1$
$|] ==> P\ (Op\ f\ e0\ e1)\ s\ (f\ n0\ n1)\ s1;$
$!!c\ e\ n\ s\ s0\ s1.$
$[|\ (c,s) -[eval\ Int\ \{((e,s),(n,s'))\}]-> s0;$
$(c,s) -[eval]-> s0;$
$(e,s0) -|-> (n,s1);\ P\ e\ s0\ n\ s1\ |]$
$==> P\ (VALOF\ c\ RESULTIS\ e)\ s\ n\ s1$
$|] ==> P\ e\ s\ n\ s'$
⟨*proof*⟩

Lemma for *Function-eval*. The major premise is that (c,s) executes to *s1* using eval restricted to its functional part. Note that the execution (c,s) $-[eval]->$ *s2* can use unrestricted *eval*! The reason is that the execution (c,s) $-[eval\ Int\ \{...\}]->$ *s1* assures us that execution is functional on the argument (c,s).

lemma *com-Unique*:
$(c,s) -[eval\ Int\ \{((e,s),(n,t)).\ \forall nt'.\ (e,s) -|-> nt' --> (n,t)=nt'\}]-> s1$
$==> \forall s2.\ (c,s) -[eval]-> s2 --> s2=s1$
⟨*proof*⟩

Expression evaluation is functional, or deterministic

theorem *single-valued-eval*: *single-valued eval*
⟨*proof*⟩

lemma *eval-N-E* [*dest*!]: $(N\ n,\ s)\ -|->\ (v,\ s')\ ==>\ (v = n\ \&\ s' = s)$
⟨*proof*⟩

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

lemma *while-true-E*:
$(c',\ s)\ -[eval]->\ t\ ==>\ c' = WHILE\ (N\ 0)\ DO\ c\ ==>\ False$
⟨*proof*⟩

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

lemma *while-if1*:
$(c',s)\ -[eval]->\ t$
$==>\ c' = WHILE\ e\ DO\ c\ ==>$
$(IF\ e\ THEN\ c;;c'\ ELSE\ SKIP,\ s)\ -[eval]->\ t$
⟨*proof*⟩

lemma *while-if2*:
$(c',s)\ -[eval]->\ t$
$==>\ c' = IF\ e\ THEN\ c;;(WHILE\ e\ DO\ c)\ ELSE\ SKIP\ ==>$
$(WHILE\ e\ DO\ c,\ s)\ -[eval]->\ t$
⟨*proof*⟩

theorem *while-if*:
$((IF\ e\ THEN\ c;;(WHILE\ e\ DO\ c)\ ELSE\ SKIP,\ s)\ -[eval]->\ t)\ =$
$((WHILE\ e\ DO\ c,\ s)\ -[eval]->\ t)$
⟨*proof*⟩

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

lemma *if-semi1*:
$(c',s)\ -[eval]->\ t$
$==>\ c' = (IF\ e\ THEN\ c1\ ELSE\ c2);;c\ ==>$
$(IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c),\ s)\ -[eval]->\ t$
⟨*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*⟩

12.5 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*⟩

12.6 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*⟩

12.7 Equivalence of VALOF x:=e RESULTIS x and e

lemma *valof-assign1*:
(e',s) −|−> (v,s'')
==> $e' =$ *VALOF* x:=e *RESULTIS* X x ==>
($\exists\, s'$. (e, s) −|−> (v,s') & ($s'' = s'(x:=v)$))
⟨*proof*⟩

lemma *valof-assign2*:
(e,s) −|−> (v,s') ==> (*VALOF* x:=e *RESULTIS* X x, s) −|−> $(v,s'(x:=v))$
⟨*proof*⟩

end