Examples of Inductive and Coinductive Definitions in HOL

Stefan Berghofer
Tobias Nipkow
Lawrence C Paulson
Markus Wenzel

October 1, 2005

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 The Mutilated Chess Board Problem

theory *Mutil* **imports** *Main* **begin**

The Mutilated Chess Board Problem, formalized inductively.

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

consts $tiling :: 'a\ set\ set => 'a\ set\ set$
inductive *tiling A*
  intros
    $empty\ [simp, intro]: \{\} \in tiling\ A$
    $Un\ [simp, intro]: \quad [|\ a \in A;\ t \in tiling\ A;\ a \cap t = \{\}\ |]$
      $==> a \cup t \in tiling\ A$

consts $domino :: (nat \times nat)\ set\ set$
inductive *domino*
  intros
    $horiz\ [simp]: \{(i, j), (i, Suc\ j)\} \in domino$
    $vertl\ [simp]: \{(i, j), (Suc\ i, j)\} \in domino$

Sets of squares of the given colour

constdefs
  $coloured :: nat => (nat \times nat)\ set$
  $coloured\ b == \{(i, j).\ (i + j)\ mod\ 2 = b\}$

syntax $whites :: (nat \times nat)\ set$
  $blacks :: (nat \times nat)\ set$

translations
  $whites == coloured\ 0$
  $blacks == coloured\ (Suc\ 0)$

The union of two disjoint tilings is a tiling

lemma *tiling-UnI* [*intro*]:
  $[|t \in tiling\ A;\ u \in tiling\ A;\ t \cap u = \{\}\ |] ==> t \cup u \in tiling\ A$
  ⟨*proof*⟩

Chess boards

lemma *Sigma-Suc1* [*simp*]:
  $lessThan\ (Suc\ n) \times B = (\{n\} \times B) \cup ((lessThan\ n) \times B)$
  ⟨*proof*⟩

lemma *Sigma-Suc2* [*simp*]:
  $A \times lessThan\ (Suc\ n) = (A \times \{n\}) \cup (A \times (lessThan\ n))$
  ⟨*proof*⟩

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

$\langle proof \rangle$

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

lemma *dominoes-tile-matrix*: $(lessThan\ m) \times lessThan\ (2 * n) \in tiling\ domino$
$\langle proof \rangle$

coloured and Dominoes

lemma *coloured-insert* [*simp*]:
$coloured\ b \cap (insert\ (i, j)\ t) =$
$(if\ (i + j)\ mod\ 2 = b\ then\ insert\ (i, j)\ (coloured\ b \cap t)$
$else\ coloured\ b \cap t)$
$\langle proof \rangle$

lemma *domino-singletons*:
$d \in domino ==>$
$(\exists\, i\ j.\ whites \cap d = \{(i, j)\}) \wedge$
$(\exists\, m\ n.\ blacks \cap d = \{(m, n)\})$
$\langle proof \rangle$

lemma *domino-finite* [*simp*]: $d \in domino ==> finite\ d$
$\langle proof \rangle$

Tilings of dominoes

lemma *tiling-domino-finite* [*simp*]: $t \in tiling\ domino ==> finite\ t$
$\langle proof \rangle$

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

lemma *tiling-domino-0-1*:
$t \in tiling\ domino ==> card(whites \cap t) = card(blacks \cap t)$
$\langle proof \rangle$

Final argument is surprisingly complex

theorem *gen-mutil-not-tiling*:
$t \in tiling\ domino ==>$
$(i + j)\ mod\ 2 = 0 ==> (m + n)\ mod\ 2 = 0 ==>$
$\{(i, j), (m, n)\} \subseteq t$
$==> (t - \{(i, j)\} - \{(m, n)\}) \notin tiling\ domino$
$\langle proof \rangle$

Apply the general theorem to the well-known case

theorem *mutil-not-tiling*:
$t = lessThan\ (2 * Suc\ m) \times lessThan\ (2 * Suc\ n)$

$==> t - \{(0,\ 0)\} - \{(Suc\ (2 * m),\ Suc\ (2 * n))\} \notin tiling\ domino$
⟨*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.

consts *msgrel* :: (*freemsg* ∗ *freemsg*) *set*

syntax
-msgrel :: [*freemsg*, *freemsg*] => *bool* (**infixl** ~~ *50*)
syntax (*xsymbols*)
-msgrel :: [*freemsg*, *freemsg*] => *bool* (**infixl** ∼ *50*)
syntax (*HTML* **output**)
-msgrel :: [*freemsg*, *freemsg*] => *bool* (**infixl** ∼ *50*)
translations
$X \sim Y == (X,Y) \in msgrel$

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 *msgrel*
intros
CD: $CRYPT\ K\ (DECRYPT\ K\ X) \sim X$
DC: $DECRYPT\ K\ (CRYPT\ K\ X) \sim X$
NONCE: $NONCE\ N \sim NONCE\ N$
MPAIR: $[\![X \sim X';\ Y \sim Y']\!] \Longrightarrow MPAIR\ X\ Y \sim MPAIR\ X'\ Y'$
CRYPT: $X \sim X' \Longrightarrow CRYPT\ K\ X \sim CRYPT\ K\ X'$
DECRYPT: $X \sim X' \Longrightarrow DECRYPT\ K\ X \sim DECRYPT\ K\ X'$
SYM: $X \sim Y \Longrightarrow Y \sim X$
TRANS: $[\![X \sim Y;\ Y \sim Z]\!] \Longrightarrow X \sim Z$

Proving that it is an equivalence relation

lemma *msgrel-refl*: $X \sim X$
$\langle proof \rangle$

theorem *equiv-msgrel*: *equiv UNIV msgrel*
$\langle proof \rangle$

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* $\Rightarrow$ *nat set*

primrec
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*: $U \sim V \Longrightarrow$ *freenonces U* = *freenonces V*
$\langle proof \rangle$

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

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

theorem *msgrel-imp-eqv-freeleft*:
$U \sim V \Longrightarrow$ *freeleft U* $\sim$ *freeleft V*
$\langle proof \rangle$

2.2.3 The Right Projection

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

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

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*:
$U \sim V \Longrightarrow \mathit{freeright}\ U \sim \mathit{freeright}\ V$
$\langle \mathit{proof} \rangle$

2.2.4 The Discriminator for Constructors

A function to distinguish nonces, mpairs and encryptions

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

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

theorem *msgrel-imp-eq-freediscrim*:
$U \sim V \Longrightarrow \mathit{freediscrim}\ U = \mathit{freediscrim}\ V$
$\langle \mathit{proof} \rangle$

2.3 The Initial Algebra: A Quotiented Message Type

typedef $(\mathit{Msg})\ \ \mathit{msg} = \mathit{UNIV}//\mathit{msgrel}$
$\langle \mathit{proof} \rangle$

The abstract message constructors

constdefs
$\mathit{Nonce} :: \mathit{nat} \Rightarrow \mathit{msg}$
$\mathit{Nonce}\ N == \mathit{Abs\text{-}Msg}(\mathit{msgrel}``\{\mathit{NONCE}\ N\})$

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

$\mathit{Crypt} :: [\mathit{nat},\mathit{msg}] \Rightarrow \mathit{msg}$
$\mathit{Crypt}\ K\ X ==$

$Abs\text{-}Msg\ (\bigcup U \in Rep\text{-}Msg\ X.\ msgrel``\{CRYPT\ K\ U\})$

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

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

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*]: $msgrel``\{U\} \in Msg$
⟨*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*: $MPair\ (Abs\text{-}Msg(msgrel``\{U\}))\ (Abs\text{-}Msg(msgrel``\{V\})) =$
$Abs\text{-}Msg\ (msgrel``\{MPAIR\ U\ V\})$
⟨*proof*⟩

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

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

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

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

constdefs
nonces :: *msg* ⇒ *nat set*
nonces X == $\bigcup U \in$ *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

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

lemma *left-congruent*: (λU. *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

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

lemma *right-congruent*: $(\lambda U.\ msgrel \text{ `` } \{freeright\ U\})$ *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.

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

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*
    | *PLUS freeExp freeExp*
    | *FNCALL nat freeExp list*

The equivalence relation, which makes PLUS associative.

consts *exprel* :: (*freeExp* ∗ *freeExp*) *set*

syntax
  -exprel :: [*freeExp*, *freeExp*] => *bool* (**infixl** ~~ *50*)
syntax (*xsymbols*)
  -exprel :: [*freeExp*, *freeExp*] => *bool* (**infixl** ∼ *50*)
syntax (*HTML* **output**)
  -exprel :: [*freeExp*, *freeExp*] => *bool* (**infixl** ∼ *50*)
translations
  $X \sim Y == (X,Y) \in$ *exprel*

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 *exprel*
  intros
    ASSOC: $PLUS\ X\ (PLUS\ Y\ Z) \sim PLUS\ (PLUS\ X\ Y)\ Z$
    VAR: $VAR\ N \sim VAR\ N$
    PLUS: $[\![X \sim X'; Y \sim Y']\!] \Longrightarrow PLUS\ X\ Y \sim PLUS\ X'\ Y'$
    FNCALL: $(Xs,Xs') \in listrel\ exprel \Longrightarrow FNCALL\ F\ Xs \sim FNCALL\ F\ Xs'$
    SYM: $X \sim Y \Longrightarrow Y \sim X$
    TRANS: $[\![X \sim Y; Y \sim Z]\!] \Longrightarrow X \sim Z$
  monos *listrel-mono*

Proving that it is an equivalence relation

lemma *exprel-refl-conj*: $X \sim X\ \&\ (Xs,Xs) \in listrel(exprel)$
⟨*proof*⟩

lemmas *exprel-refl* = *exprel-refl-conj* [*THEN conjunct1*]
lemmas *list-exprel-refl* = *exprel-refl-conj* [*THEN conjunct2*]

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

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

lemma *FNCALL-Nil*: *FNCALL F* [] $\sim$ *FNCALL F* []
⟨*proof*⟩

lemma *FNCALL-Cons*:
  $[\![X \sim X'; (Xs,Xs') \in listrel(exprel)]\!]$
  $\Longrightarrow$ *FNCALL F* $(X\#Xs) \sim$ *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* $\Rightarrow$ *nat set*
freevars-list :: *freeExp list* $\Rightarrow$ *nat set*

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

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

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

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

3.2.2 Functions for Freeness

A discriminator function to distinguish vars, sums and function calls

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

theorem *exprel-imp-eq-freediscrim*:
$U \sim V \Longrightarrow \mathit{freediscrim}\ U = \mathit{freediscrim}\ V$
⟨*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*:
$U \sim V \Longrightarrow \mathit{freefun}\ U = \mathit{freefun}\ V$
⟨*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*:
$U \sim V \Longrightarrow (\mathit{freeargs}\ U, \mathit{freeargs}\ V) \in \mathit{listrel\ exprel}$
⟨*proof*⟩

3.3 The Initial Algebra: A Quotiented Message Type

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

The abstract message constructors

constdefs
Var :: *nat* ⇒ *exp*
$\mathit{Var}\ N == \mathit{Abs\text{-}Exp}(\mathit{exprel}``\{\mathit{VAR}\ N\})$

Plus :: [*exp*,*exp*] ⇒ *exp*
Plus X Y ==
$\mathit{Abs\text{-}Exp}\ (\bigcup U \in \mathit{Rep\text{-}Exp}\ X.\ \bigcup V \in \mathit{Rep\text{-}Exp}\ Y.\ \mathit{exprel}``\{\mathit{PLUS}\ U\ V\})$

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

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

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

declare *equiv-exprel-iff* [*simp*]

All equivalence classes belong to set of representatives

lemma [*simp*]: *exprel*``{*U*} ∈ *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

constdefs *Abs-ExpList* :: *freeExp list* => *exp list*
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*: $FnCall\ F\ [] = Abs\text{-}Exp\ (exprel``\{FNCALL\ F\ []\})$
⟨*proof*⟩

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

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

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

lemma *FnCall*:
$FnCall\ F\ (Abs\text{-}ExpList\ Us) = Abs\text{-}Exp\ (exprel``\{FNCALL\ F\ Us\})$
⟨*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

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

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

The extension of the function *vars* to lists

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

Now prove the three equations for *vars*

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

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

lemma *vars-FnCall* [*simp*]: $vars\ (FnCall\ F\ Xs) = vars\text{-}list\ Xs$

⟨*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*

constdefs
fun :: *exp* ⇒ *nat*
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*⟩

constdefs
args :: *exp* ⇒ *exp list*
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.

constdefs
  discrim :: *exp* ⇒ *int*
  discrim X == *contents* ($\bigcup$ *U* ∈ *Rep-Exp X*. {*freediscrim U*})

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

Now prove the four equations for *discrim*

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

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

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

The structural induction rule for the abstract type

theorem *exp-induct*:
  assumes *V*: ⋀*nat*. *P1* (*Var nat*)
    and *P*: ⋀*exp1 exp2*. ⟦*P1 exp1*; *P1 exp2*⟧ ⟹ *P1* (*Plus exp1 exp2*)
    and *F*: ⋀*nat list*. *P2 list* ⟹ *P1* (*FnCall nat list*)
    and *Nil*: *P2* []
    and *Cons*: ⋀*exp list*. ⟦*P1 exp*; *P2 list*⟧ ⟹ *P2* (*exp* # *list*)
  shows *P1 exp* & *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 :: $('a$ => $('a, 'b)$ *term*) => $('a, 'b)$ *term* => $('a, 'b)$ *term*
  subst-term-list ::
    $('a$ => $('a, 'b)$ *term*) => $('a, 'b)$ *term list* => $('a, 'b)$ *term list*

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

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

A simple theorem about composition of substitutions

lemma *subst-comp*:
  subst-term (*subst-term f1* $\circ$ *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*: !!*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*
  $\langle$*proof*$\rangle$

end

5 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

$evala\ env\ (IF\ b\ a1\ a2) = (if\ evalb\ env\ b\ then\ evala\ env\ a1\ else\ evala\ env\ a2)$

$evala\ env\ (Sum\ a1\ a2) = evala\ env\ a1 + evala\ env\ a2$

$evala\ env\ (Diff\ a1\ a2) = evala\ env\ a1 - evala\ env\ a2$

$evala\ env\ (Var\ v) = env\ v$

$evala\ env\ (Num\ n) = n$

$evalb\ env\ (Less\ a1\ a2) = (evala\ env\ a1 < evala\ env\ a2)$

$evalb\ env\ (And\ b1\ b2) = (evalb\ env\ b1 \wedge evalb\ env\ b2)$

$evalb\ env\ (Neg\ b) = (\neg\ evalb\ env\ b)$

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*:

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

and *subst-all-bexp*:

$evalb\ env\ (substb\ s\ b) = evalb\ (\lambda x.\ evala\ env\ (s\ x))\ b$
⟨*proof*⟩

end

6 Infinitely branching trees

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

datatype *'a tree* =
  Atom 'a
| *Branch nat => 'a tree*

consts
  map-tree :: (*'a => 'b*) => *'a tree => 'b tree*
primrec
  map-tree f (*Atom a*) = *Atom* (*f a*)
  map-tree f (*Branch ts*) = *Branch* ($\lambda x.$ *map-tree f* (*ts x*))

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

consts
  exists-tree :: (*'a => bool*) => *'a tree => bool*
primrec
  exists-tree P (*Atom a*) = *P a*
  exists-tree P (*Branch ts*) = ($\exists\, x.$ *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*⟩

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

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

Addition of ordinals

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

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

Multiplication of ordinals

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

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

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

constdefs
  brouwer-pred :: (*brouwer* ∗ *brouwer*) *set*
  brouwer-pred == $\bigcup$ *i*. {(*m*,*n*). *n* = *Succ m* ∨ (*EX f*. *n* = *Lim f* & *m* = *f i*)}

  brouwer-order :: (*brouwer* ∗ *brouwer*) *set*
  brouwer-order == *brouwer-pred*^+

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

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

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

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

Example of a recdef

consts
  add2 :: (*brouwer*∗*brouwer*) => *brouwer*
recdef *add2 inv-image brouwer-order* (λ (*x*,*y*). *y*)
  add2 (*i*, *Zero*) = *i*
  add2 (*i*, (*Succ j*)) = *Succ* (*add2* (*i*, *j*))

add2 (*i*, (*Lim f*)) = *Lim* (λ *n*. *add2* (*i*, (*f n*)))
(**hints** *recdef-wf*: *wf-brouwer-order*)

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

end

7 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*)

constdefs
  OpLim :: (*nat* => (*ordinal* => *ordinal*)) => (*ordinal* => *ordinal*)
  OpLim F a == *Limit* (λ*n*. *F n a*)
  OpItw :: (*ordinal* => *ordinal*) => (*ordinal* => *ordinal*)   ($\bigsqcup$)
  $\bigsqcup$*f* == *OpLim* (*iter f*)

consts
  cantor :: *ordinal* => *ordinal* => *ordinal*
primrec
  cantor a Zero = *Succ a*
  cantor a (*Succ b*) = $\bigsqcup$(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* (λn. ∇f (*h n*))

constdefs
  deriv :: (*ordinal* => *ordinal*) => (*ordinal* => *ordinal*)
  deriv f == $\nabla(\bigsqcup f)$

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

constdefs
  veb a == *veblen a Zero*
  ε_0 == *veb Zero*
  Γ_0 == *Limit* (λn. *iter veb n Zero*)

end

8 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.

consts
  σ*-algebra* :: *'a set set* => *'a set set*

inductive σ*-algebra A*
  intros
    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*
$\langle$*proof*$\rangle$

theorem *sigma-algebra-Inter*:

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

end

9 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

9.1 Definitions

Datatype definition of combinators S and K.

datatype *comb* = *K*
  | *S*
  | ## *comb comb* (**infixl** *90*)

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

consts
  contract :: (*comb*∗*comb*) *set*
  −*1*−> :: [*comb*,*comb*] => *bool* (**infixl** *50*)
  −−−> :: [*comb*,*comb*] => *bool* (**infixl** *50*)

translations
  x −*1*−> *y* == (*x*,*y*) ∈ *contract*
  x −−−> *y* == (*x*,*y*) ∈ *contract*ˆ∗

syntax (*xsymbols*)
  op ## :: [*comb*,*comb*] => *comb* (**infixl** · *90*)

inductive *contract*
  intros
    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*

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

consts
parcontract :: (*comb*∗*comb*) *set*
=*1*=> :: [*comb*,*comb*] => *bool* (**infixl** *50*)
===> :: [*comb*,*comb*] => *bool* (**infixl** *50*)

translations
x =*1*=> *y* == (*x*,*y*) ∈ *parcontract*
x ===> *y* == (*x*,*y*) ∈ *parcontract*^∗

inductive *parcontract*
intros
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*

Misc definitions.

constdefs
I :: *comb*
I == *S*##*K*##*K*

diamond :: ('*a* ∗ '*a*)*set* => *bool*
— 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*))

9.2 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*⟩

9.3 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*!]: $K\#\#x\ -1->\ z\ ==>\ (\exists x'.\ z = K\#\#x'\ \&\ x\ -1->\ x')$
⟨*proof*⟩

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

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

lemma *KIII-contract1*: $K\#\#I\#\#(I\#\#I)\ -1->\ I$
⟨*proof*⟩

lemma *KIII-contract2*: $K\#\#I\#\#(I\#\#I)\ -1->\ K\#\#I\#\#((K\#\#I)\#\#(K\#\#I))$
⟨*proof*⟩

lemma *KIII-contract3*: $K\#\#I\#\#((K\#\#I)\#\#(K\#\#I))\ -1->\ I$
⟨*proof*⟩

lemma *not-diamond-contract*: $\sim\ diamond(contract)$
⟨*proof*⟩

9.4 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*]

9.5 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 $=1=>$ $x')$
$\langle proof \rangle$

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

The rules above are not essential but make proofs much faster

Church-Rosser property for parallel contraction

lemma *diamond-parcontract*: *diamond parcontract*
$\langle proof \rangle$

Equivalence of p $--->$ q and p $===>$ q.

lemma *contract-subset-parcontract*: *contract* $<=$ *parcontract*
$\langle proof \rangle$

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
$\langle proof \rangle$

lemma *parcontract-subset-reduce*: *parcontract* $<=$ *contract*^*
$\langle proof \rangle$

lemma *reduce-eq-parreduce*: *contract*^* = *parcontract*^*
$\langle proof \rangle$

lemma *diamond-reduce*: *diamond*(*contract*^*)
$\langle proof \rangle$

end

10 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 $G \in Fin(H)$

10.1 The datatype of propositions

datatype
$'a\ pl = false \mid var\ 'a\ (\#\text{-}\ [1000]) \mid -\!> \ 'a\ pl\ 'a\ pl$ (**infixr** *90*)

10.2 The proof system

consts
$thms\ ::\ 'a\ pl\ set => 'a\ pl\ set$
$|-\ ::\ ['a\ pl\ set,\ 'a\ pl] => bool$ (**infixl** *50*)

translations
$H\ |-\ p == p \in thms(H)$

inductive $thms(H)$
intros
$H\ [intro]$: $p{\in}H ==> H\ |-\ p$
K: $H\ |-\ p->q->p$
S: $H\ |-\ (p->q->r) -> (p->q) -> p->r$
DN: $H\ |-\ ((p->false) -> false) -> p$
MP: $[|\ H\ |-\ p->q;\ H\ |-\ p\ |] ==> H\ |-\ q$

10.3 The semantics

10.3.1 Semantics of propositional logic.

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

primrec $tt[[false]] = False$
$tt[[\#v]] = (v \in tt)$
$eval\text{-}imp$: $tt[[p->q]] = (tt[[p]] --> tt[[q]])$

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

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

primrec
$hyps\ false\ \ tt = \{\}$
$hyps\ (\#v)\ \ tt = \{if\ v \in tt\ then\ \#v\ else\ \#v->false\}$
$hyps\ (p->q)\ tt = hyps\ p\ tt\ Un\ hyps\ q\ tt$

10.3.2 Logical consequence

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

constdefs
$sat\ ::\ ['a\ pl\ set,\ 'a\ pl] => bool$ (**infixl** $|=$ *50*)

$H \mid= p \;==\; (\forall\, tt.\ (\forall\, q{\in}H.\ tt[[q]]) \dashrightarrow tt[[p]])$

10.4 Proof theory of propositional logic

lemma *thms-mono*: $G{<=}H \implies \mathit{thms}(G) <= \mathit{thms}(H)$
⟨*proof*⟩

lemma *thms-I*: $H \mid- p{-}{>}p$
— Called I for Identity Combinator, not for Introduction.
⟨*proof*⟩

10.4.1 Weakening, left and right

lemma *weaken-left*: $[\mid G \subseteq H;\ \ G\mid-p \mid] \implies H\mid-p$
— 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*: $H \mid- q \implies H \mid- p{-}{>}q$
⟨*proof*⟩

10.4.2 The deduction theorem

theorem *deduction*: $\mathit{insert}\ p\ H \mid- q \implies H \mid- p{-}{>}q$
⟨*proof*⟩

10.4.3 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*]

10.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: $H \mid- p \implies H \mid= p$
⟨*proof*⟩

10.5 Completeness

10.5.1 Towards the completeness proof

lemma *false-imp*: $H \mid- p{-}{>}\mathit{false} \implies H \mid- p{-}{>}q$
⟨*proof*⟩

lemma *imp-false*:

$[| \ H \ |- \ p; \ \ H \ |- \ q->false \ |] \ ==> \ H \ |- \ (p->q)->false$
⟨*proof*⟩

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

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

For proving certain theorems in our new propositional logic.

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

The excluded middle in the form of an elimination rule.

lemma *thms-excluded-middle*: $H\ |- \ (p->q)\ ->\ ((p->false)->q)\ ->\ q$
⟨*proof*⟩

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

10.6 Completeness – lemmas for reducing the set of assumptions

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

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

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

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*]

10.6.1 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* [*rule-format*]: *finite H* ==> ∀ *p*. *H* |= *p* −−> *H* |− *p*
⟨*proof*⟩

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

end

theory *Sexp* **imports** *Datatype-Universe Inductive* **begin**
consts
 sexp :: *'a item set*

inductive *sexp*
 intros
  LeafI: *Leaf*(*a*) ∈ *sexp*
  NumbI: *Numb*(*i*) ∈ *sexp*
  SconsI: [| *M* ∈ *sexp*; *N* ∈ *sexp* |] ==> *Scons M N* ∈ *sexp*

constdefs

 sexp-case :: [*'a*=>*'b*, *nat*=>*'b*, [*'a item*, *'a item*]=>*'b*,
   'a item] => *'b*
  sexp-case c d e M == *THE z*. (*EX x*. *M*=*Leaf*(*x*) & *z*=*c*(*x*))

$| (EX\ k.\quad M{=}Numb(k)\ \&\ z{=}d(k))$
$| (EX\ N1\ N2.\ M = Scons\ N1\ N2\ \ \&\ z{=}e\ N1\ N2)$

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

sexp-rec :: ['a item, 'a=>'b, nat=>'b,
['a item, 'a item, 'b, 'b]=>'b] => 'b
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*: $M \in sexp$ ==> *In0*(*M*) $\in sexp$
⟨*proof*⟩

lemma *sexp-In1I*: $M \in sexp$ ==> *In1*(*M*) $\in sexp$
⟨*proof*⟩

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

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

lemma *Scons-D*: *Scons M N* $\in sexp$ ==> $M \in sexp$ & $N \in sexp$
⟨*proof*⟩

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

lemmas *trancl-pred-sexpD1* =
pred-sexp-subset-Sigma

$[THEN\ \mathit{trancl\text{-}subset\text{-}Sigma},\ THEN\ \mathit{subsetD},\ THEN\ \mathit{SigmaD1}]$
and *trancl-pred-sexpD2* =
pred-sexp-subset-Sigma
$[THEN\ \mathit{trancl\text{-}subset\text{-}Sigma},\ THEN\ \mathit{subsetD},\ THEN\ \mathit{SigmaD2}]$

lemma *pred-sexpI1*:
$[|\ M \in sexp;\ \ N \in sexp\ |] ==> (M,\ Scons\ M\ N) \in pred\text{-}sexp$
⟨*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*:
$(\%M.\ sexp\text{-}rec\ M\ c\ d\ e) ==$
$wfrec\ pred\text{-}sexp\ (\%g.\ sexp\text{-}case\ c\ d\ (\%N1\ N2.\ e\ N1\ N2\ (g\ N1)\ (g\ N2)))$
⟨*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-rec* (*Numb* *k*) *c* *d* *h* = *d*(*k*)
⟨*proof*⟩

lemma *sexp-rec-Scons*: [| *M* ∈ *sexp*; *N* ∈ *sexp* |] ==>
sexp-rec (*Scons* *M* *N*) *c* *d* *h* = *h* *M* *N* (*sexp-rec* *M* *c* *d* *h*) (*sexp-rec* *N* *c* *d* *h*)
⟨*proof*⟩

end

theory *SList* **imports** *NatArith* *Sexp* *Hilbert-Choice* **begin**

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

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

consts
list :: '*a* *item* *set* => '*a* *item* *set*
inductive *list*(*A*)
intros
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*⟩

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

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

constdefs
Nil :: *'a list*
Nil == *Abs-List*(*NIL*)

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

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

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

consts
[] :: *'a list* ([])
syntax
@*list* :: *args* => *'a list* ([(-)])

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

case xs of Nil => *a* | *y#ys* => *b* == *list-case*(*a*, %*y ys*. *b*, *xs*)

```
constdefs
  Rep-map   :: ('b => 'a item) => ('b list => 'a item)
   Rep-map f xs == list-rec xs  NIL(%x l r. CONS(f x) r)

  Abs-map   :: ('a item => 'b) => 'a item => 'b list
   Abs-map g M  == List-rec M Nil (%N L r. g(N)#r)
```

```
constdefs

  null      :: 'a list => bool
  null xs  == list-rec xs True (%x xs r. False)

  hd        :: 'a list => 'a
  hd xs     == list-rec xs (@x. True) (%x xs r. x)

  tl        :: 'a list => 'a list
  tl xs     == list-rec xs (@xs. True) (%x xs r. xs)


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

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

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

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

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

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

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

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

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

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

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

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

nth :: [nat, 'a list] => 'a
nth == nat-rec hd (%m r xs. r(tl xs))

rev :: 'a list => 'a list
rev xs == list-rec xs [] (%x xs xsa. xsa @ [x])

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

zip :: 'a list * 'b list => ('a*'b) list
zip == zipWith (%s. s)

unzip :: ('a*'b) list => ('a list * 'b list)
unzip == foldr(% (a,b)(c,d).(a#c,b#d))([],[])

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

consts enum :: [nat,nat] => nat list
primrec
enum-0: enum i 0 = []

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

syntax

```
@Alls     :: [idt, 'a list, bool] => bool      ((2Alls -:-./ -) 10)
@filter   :: [idt, 'a list, bool] => 'a list   ((1[-:- ./ -]))
```

translations

```
[x:xs. P]    == filter(%x. P) xs
Alls x:xs. P== list-all(%x. P)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*: ∀ *x*. *l* ~= *x*#*l*
⟨*proof*⟩

lemma *neq-Nil-conv2*: (*xs* ~= []) = (∃ *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 (*trancl 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 null-def*, *simp*]
declare *def-list-rec-NilCons* [*OF hd-def*, *simp*]
declare *def-list-rec-NilCons* [*OF tl-def*, *simp*]
declare *def-list-rec-NilCons* [*OF ttl-def*, *simp*]
declare *def-list-rec-NilCons* [*OF append-def*, *simp*]
declare *def-list-rec-NilCons* [*OF member-def*, *simp*]
declare *def-list-rec-NilCons* [*OF map-def*, *simp*]
declare *def-list-rec-NilCons* [*OF filter-def*, *simp*]
declare *def-list-rec-NilCons* [*OF list-all-def*, *simp*]

lemma *def-nat-rec-0-eta*:
  [| !!*n*. *f* == *nat-rec c h* |] ==> *f*(*0*) = *c*
⟨*proof*⟩

lemma *def-nat-rec-Suc-eta*:
  [| !!*n*. *f* == *nat-rec c h* |] ==> *f*(*Suc*(*n*)) = *h n* (*f n*)
⟨*proof*⟩

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

lemma *length-Nil* [*simp*]: *length*([]) = *0*

⟨*proof*⟩

lemma *length-Cons* [*simp*]: *length*(*a#xs*) = *Suc*(*length*(*xs*))
⟨*proof*⟩

lemma *append-assoc* [*simp*]: (*xs*@*ys*)@*zs* = *xs*@(*ys*@*zs*)
⟨*proof*⟩

lemma *append-Nil2* [*simp*]: *xs* @ [] = *xs*
⟨*proof*⟩

lemma *mem-append* [*simp*]: *x mem* (*xs*@*ys*) = (*x mem xs* | *x mem ys*)
⟨*proof*⟩

lemma *mem-filter* [*simp*]: *x mem* [*x*:*xs*. *P x*] = (*x mem xs* & *P*(*x*))
⟨*proof*⟩

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

lemma *list-all-conj* [*simp*]:
list-all p (*xs*@*ys*) = ((*list-all p xs*) & (*list-all p ys*))
⟨*proof*⟩

lemma *list-all-mem-conv*: (*Alls x*:*xs*. *P*(*x*)) = (!*x*. *x mem xs* --> *P*(*x*))
⟨*proof*⟩

lemma *nat-case-dist* : (! *n*. *P n*) = (*P 0* & (! *n*. *P* (*Suc n*)))
⟨*proof*⟩

lemma *alls-P-eq-P-nth*: (*Alls u*:*A*. *P u*) = (!*n*. *n* < *length A* --> *P*(*nth n A*))
⟨*proof*⟩

lemma *list-all-imp*:
[| !*x*. *P x* --> *Q x*; (*Alls x*:*xs*. *P*(*x*)) |] ==> (*Alls x*:*xs*. *Q*(*x*))
⟨*proof*⟩

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

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

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

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

lemma *mem-map-aux1* [*rule-format*]:
  $x\ mem\ (map\ f\ q) --> (\exists\, y.\ y\ mem\ q\ \&\ x = f\ y)$
⟨*proof*⟩

lemma *mem-map-aux2* [*rule-format*]:
  $(\exists\, y.\ y\ mem\ q\ \&\ x = f\ y) --> x\ mem\ (map\ f\ q)$
⟨*proof*⟩

lemma *mem-map*: $x\ mem\ (map\ f\ q) = (\exists\, y.\ y\ mem\ q\ \&\ x = f\ y)$
⟨*proof*⟩

lemma *hd-append* [*rule-format*]: $A\ \tilde{}= []\ --> hd(A\ @\ B) = hd(A)$
⟨*proof*⟩

lemma *tl-append* [*rule-format*]: $A\ \tilde{}= []\ --> tl(A\ @\ B) = tl(A)\ @\ B$
⟨*proof*⟩

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

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

lemma *drop-0* [*simp*]: $drop\ xs\ 0 = xs$
⟨*proof*⟩

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

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

lemma *copy-0* [*simp*]: *copy x 0* = []
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *zipWith-Cons-Cons* [*simp*]:
  $zipWith\ f\ (a\#as,b\#bs) \ = f(a,b)\ \#\ zipWith\ f\ (as,bs)$
⟨*proof*⟩

lemma *zipWith-Nil-Nil* [*simp*]: $zipWith\ f\ ([],[]) \quad = []$
⟨*proof*⟩

lemma *zipWith-Cons-Nil* [*simp*]: $zipWith\ f\ (x,[]) \ = []$
⟨*proof*⟩

lemma *zipWith-Nil-Cons* [*simp*]: $zipWith\ f\ ([],x) = []$
⟨*proof*⟩

lemma *unzip-Nil* [*simp*]: $unzip\ [] = ([],[])$
⟨*proof*⟩

lemma *map-compose-ext*: $map(f\ o\ g) = ((map\ f)\ o\ (map\ g))$
⟨*proof*⟩

lemma *map-flat*: $map\ f\ (flat\ S) = flat(map\ (map\ f)\ S)$
⟨*proof*⟩

lemma *list-all-map-eq*: $(Alls\ u{:}xs.\ f(u) = g(u))\ {-}{-}{>}\ map\ f\ xs = map\ g\ xs$
⟨*proof*⟩

lemma *filter-map-d*: $filter\ p\ (map\ f\ xs) = map\ f\ (filter(p\ o\ f)(xs))$
⟨*proof*⟩

lemma *filter-compose*: $filter\ p\ (filter\ q\ xs) = filter(\%x.\ p\ x\ \&\ q\ x)\ xs$
⟨*proof*⟩

lemma *filter-append* [*rule-format*, *simp*]:
  $\forall B.\ filter\ p\ (A\ @\ B) = (filter\ p\ A\ @\ filter\ p\ B)$
⟨*proof*⟩

lemma *length-append*: $length(xs@ys) = length(xs)+length(ys)$
⟨*proof*⟩

lemma *length-map*: $length(map\ f\ xs) = length(xs)$
⟨*proof*⟩

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

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

lemma *take-take-Suc-eq1* [*rule-format*]:
$\forall n.\ take\ (take\ xs(Suc(n+m)))\ n = take\ xs\ n$
⟨*proof*⟩

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

lemma *take-take-1*: $take\ (take\ xs\ (n+m))\ n = take\ xs\ n$
⟨*proof*⟩

lemma *take-take-Suc-eq2* [*rule-format*]:
$\forall n.\ take\ (take\ xs\ n)(Suc(n+m)) = take\ xs\ n$
⟨*proof*⟩

lemma *take-take-2*: $take(take\ xs\ n)(n+m) = take\ xs\ n$
⟨*proof*⟩

lemma *drop-Nil* [*simp*]: $drop\ \ []\ n\ = []$
⟨*proof*⟩

lemma *drop-drop* [*rule-format*]: $\forall xs.\ drop\ (drop\ xs\ m)\ n = drop\ xs(m+n)$
⟨*proof*⟩

lemma *take-drop* [*rule-format*]: $\forall xs.\ (take\ xs\ n)\ @\ (drop\ xs\ n) = xs$
⟨*proof*⟩

lemma *copy-copy*: $copy\ x\ n\ @\ copy\ x\ m = copy\ x\ (n+m)$
⟨*proof*⟩

lemma *length-copy*: $length(copy\ x\ n)\ = n$
⟨*proof*⟩

lemma *length-take* [*rule-format*, *simp*]:

$\forall xs.\ length(take\ xs\ n) = min\ (length\ xs)\ n$
$\langle proof \rangle$

lemma *length-take-drop*: $length(take\ A\ k) + length(drop\ A\ k) = length(A)$
$\langle proof \rangle$

lemma *take-append* [*rule-format*]: $\forall A.\ length(A) = n\ {-}{-}{>}\ take(A@B)\ n = A$
$\langle proof \rangle$

lemma *take-append2* [*rule-format*]:
$\forall A.\ length(A) = n\ {-}{-}{>}\ take(A@B)\ (n+k) = A\ @\ take\ B\ k$
$\langle proof \rangle$

lemma *take-map* [*rule-format*]: $\forall n.\ take\ (map\ f\ A)\ n = map\ f\ (take\ A\ n)$
$\langle proof \rangle$

lemma *drop-append* [*rule-format*]: $\forall A.\ length(A) = n\ {-}{-}{>}\ drop(A@B)n = B$
$\langle proof \rangle$

lemma *drop-append2* [*rule-format*]:
$\forall A.\ length(A) = n\ {-}{-}{>}\ drop(A@B)(n+k) = drop\ B\ k$
$\langle proof \rangle$

lemma *drop-all* [*rule-format*]: $\forall A.\ length(A) = n\ {-}{-}{>}\ drop\ A\ n = []$
$\langle proof \rangle$

lemma *drop-map* [*rule-format*]: $\forall n.\ drop\ (map\ f\ A)\ n = map\ f\ (drop\ A\ n)$
$\langle proof \rangle$

lemma *take-all* [*rule-format*]: $\forall A.\ length(A) = n\ {-}{-}{>}\ take\ A\ n = A$
$\langle proof \rangle$

lemma *foldl-single*: $foldl\ f\ a\ [b] = f\ a\ b$
$\langle proof \rangle$

lemma *foldl-append* [*rule-format, simp*]:
$\forall a.\ foldl\ f\ a\ (A\ @\ B) = foldl\ f\ (foldl\ f\ a\ A)\ B$
$\langle proof \rangle$

lemma *foldl-map* [*rule-format*]:
$\forall e.\ foldl\ f\ e\ (map\ g\ S) = foldl\ (\%x\ y.\ f\ x\ (g\ y))\ e\ S$
$\langle proof \rangle$

lemma *foldl-neutr-distr* [*rule-format*]:
assumes *r-neutr*: $\forall a.\ f\ a\ e = a$
and *r-neutl*: $\forall a.\ f\ e\ a = a$
and *assoc*: $\forall a\ b\ c.\ f\ a\ (f\ b\ c) = f(f\ a\ b)\ c$
shows $\forall y.\ f\ y\ (foldl\ f\ e\ A) = foldl\ f\ y\ A$

⟨*proof*⟩

lemma *foldl-append-sym*:
$[| \; !a. \; f \; a \; e = a; \; !a. \; f \; e \; a = a;$
$\quad !a \; b \; c. \; f \; a \; (f \; b \; c) = f(f \; a \; b) \; c \; |]$
$\; ==> foldl \; f \; e \; (A \; @ \; B) = f(foldl \; f \; e \; A)(foldl \; f \; e \; B)$
⟨*proof*⟩

lemma *foldr-append* [*rule-format*, *simp*]:
$\quad \forall \, a. \; foldr \; f \; a \; (A \; @ \; B) = foldr \; f \; (foldr \; f \; a \; B) \; A$
⟨*proof*⟩

lemma *foldr-map* [*rule-format*]: $\forall \, e. \; foldr \; f \; e \; (map \; g \; S) = foldr \; (f \; o \; g) \; e \; S$
⟨*proof*⟩

lemma *foldr-Un-eq-UN*: $foldr \; op \; Un \; \{\} \; S = (UN \; X{:} \; \{t. \; t \; mem \; S\}.X)$
⟨*proof*⟩

lemma *foldr-neutr-distr*:
$\quad [| \; !a. \; f \; e \; a = a; \; !a \; b \; c. \; f \; a \; (f \; b \; c) = f(f \; a \; b) \; c \; |]$
$\quad ==> foldr \; f \; y \; S = f \; (foldr \; f \; e \; S) \; y$
⟨*proof*⟩

lemma *foldr-append2*:
$\quad [| \; !a. \; f \; e \; a = a; \; !a \; b \; c. \; f \; a \; (f \; b \; c) = f(f \; a \; b) \; c \; |]$
$\quad ==> foldr \; f \; e \; (A \; @ \; B) = f \; (foldr \; f \; e \; A) \; (foldr \; f \; e \; B)$
⟨*proof*⟩

lemma *foldr-flat*:
$\quad [| \; !a. \; f \; e \; a = a; \; !a \; b \; c. \; f \; a \; (f \; b \; c) = f(f \; a \; b) \; c \; |] ==>$
$\quad foldr \; f \; e \; (flat \; S) = (foldr \; f \; e)(map \; (foldr \; f \; e) \; S)$
⟨*proof*⟩

lemma *list-all-map*: $(Alls \; x{:}map \; f \; xs \; .P(x)) = (Alls \; x{:}xs.(P \; o \; f)(x))$
⟨*proof*⟩

lemma *list-all-and*:
$\quad (Alls \; x{:}xs. \; P(x) \& Q(x)) = ((Alls \; x{:}xs. \; P(x)) \& (Alls \; x{:}xs. \; Q(x)))$
⟨*proof*⟩

lemma *nth-map* [*rule-format*]:
$\quad \forall \, i. \; i < length(A) \; --> nth \; i \; (map \; f \; A) = f(nth \; i \; A)$
⟨*proof*⟩

lemma *nth-app-cancel-right* [*rule-format*]:
$\quad \forall \, i. \; i < length(A) \; --> nth \; i(A@B) = nth \; i \; A$

⟨*proof*⟩

lemma *nth-app-cancel-left* [*rule-format*]:
  $\forall n.\ n = length(A)\ {-}{-}{>}\ nth(n{+}i)(A@B) = nth\ i\ B$
⟨*proof*⟩

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

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

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

lemma *rev-rev-ident* [*simp*]: $rev(rev\ l) = l$
⟨*proof*⟩

lemma *rev-flat*: $rev(flat\ ls) = flat\ (map\ rev\ (rev\ ls))$
⟨*proof*⟩

lemma *rev-map-distrib*: $rev(map\ f\ l) = map\ f\ (rev\ l)$
⟨*proof*⟩

lemma *foldl-rev*: $foldl\ f\ b\ (rev\ l) = foldr\ (\%x\ y.\ f\ y\ x)\ b\ l$
⟨*proof*⟩

lemma *foldr-rev*: $foldr\ f\ b\ (rev\ l) = foldl\ (\%x\ y.\ f\ y\ x)\ b\ l$
⟨*proof*⟩

end

11 Definition of type llist by a greatest fixed point

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

consts

  $llist\ ::\ 'a\ item\ set\ =>\ 'a\ item\ set$
  $LListD\ ::\ ('a\ item * 'a\ item)set\ =>\ ('a\ item * 'a\ item)set$

coinductive *llist*(A)
intros
NIL-I: *NIL* $\in$ *llist*(A)
CONS-I: [| $a \in A$; $M \in$ *llist*(A) |] ==> *CONS* a $M \in$ *llist*(A)

coinductive *LListD*(r)
intros
NIL-I: (*NIL*, *NIL*) $\in$ *LListD*(r)
CONS-I: [| (a,b) $\in r$; (M,N) $\in$ *LListD*(r) |]
==> (*CONS* a M, *CONS* b N) $\in$ *LListD*(r)

typedef (*LList*)
'a llist = *llist*(*range Leaf*) :: *'a item set*
⟨*proof*⟩

constdefs
list-Fun :: [*'a item set*, *'a item set*] => *'a item set*
— Now used exclusively for abbreviating the coinduction rule
list-Fun A X == {z. z = *NIL* | ($\exists$ M a. z = *CONS a M* & $a \in A$ & $M \in X$)}

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

LNil :: *'a llist*
— abstract constructor
LNil == *Abs-LList NIL*

LCons :: [*'a*, *'a llist*] => *'a llist*
— abstract constructor
LCons x xs == *Abs-LList*(*CONS* (*Leaf x*) (*Rep-LList xs*))

llist-case :: [*'b*, [*'a*, *'a llist*]=>*'b*, *'a llist*] => *'b*
llist-case c d l ==
List-case c (%*x y*. *d* (*inv Leaf x*) (*Abs-LList y*)) (*Rep-LList l*)

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

LList-corec :: [*'a*, *'a* => (*'b item* $*$ *'a*) *option*] => *'b item*
LList-corec a f == $\bigcup$ *k*. *LList-corec-fun k f a*

llist-corec :: ['a, 'a => ('b * 'a) option] => 'b llist
  llist-corec a f ==
    Abs-LList(*LList-corec a*
      (%z. case f z of None => None
        | Some(v,w) => Some(Leaf(v), w)))

llistD-Fun :: ('a llist * 'a llist)set => ('a llist * 'a llist)set
  llistD-Fun(r) ==
    prod-fun Abs-LList Abs-LList ‘
      LListD-Fun (diag(range Leaf))
        (prod-fun Rep-LList Rep-LList ‘ r)

The case syntax for type *'a llist*

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

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

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

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

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

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

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

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

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

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

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

11.0.3 Simplification

declare *option.split* [*split*]

This justifies using llist in other recursive type definitions

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

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

11.1 Type checking by coinduction

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

lemma *llist-coinduct*:
$[|\ M \in X;\ \ X <= list\text{-}Fun\ A\ (X\ Un\ llist(A))\ |]$ ==> $M \in llist(A)$
⟨*proof*⟩

lemma *list-Fun-NIL-I* [*iff*]: $NIL \in list\text{-}Fun\ A\ X$
⟨*proof*⟩

lemma *list-Fun-CONS-I* [*intro*!,*simp*]:
$[|\ M \in A;\ \ N \in X\ |]$ ==> $CONS\ M\ N \in list\text{-}Fun\ A\ X$
⟨*proof*⟩

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

lemma *list-Fun-llist-I*: $M \in llist(A)$ ==> $M \in list\text{-}Fun\ A\ (X\ Un\ llist(A))$
⟨*proof*⟩

11.2 *LList-corec* satisfies the desired recurion equation

A continuity result?

lemma *CONS-UN1*: $CONS\ M\ (\bigcup x.\ f(x)) = (\bigcup x.\ CONS\ M\ (f\ x))$
⟨*proof*⟩

lemma *CONS-mono*: $[|\ M<=M';\ \ N<=N'\ |]$ ==> $CONS\ M\ N <= CONS\ M'\ N'$
⟨*proof*⟩

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

11.2.1 The directions of the equality are proved separately

lemma *LList-corec-subset1*:
LList-corec a f <=

(case f a of None => NIL | Some(z,w) => CONS z (LList-corec w f))
⟨*proof*⟩

lemma *LList-corec-subset2*:
(case f a of None => NIL | Some(z,w) => CONS z (LList-corec w f)) <=
LList-corec a f
⟨*proof*⟩

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

lemma *LList-corec*:
LList-corec a f =
(case f a of None => NIL | Some(z,w) => CONS z (LList-corec w f))
⟨*proof*⟩

definitional version of same

lemma *def-LList-corec*:
[| !!x. h(x) == LList-corec x f |]
==> h(a) = (case f a of None => NIL | Some(z,w) => CONS z (h w))
⟨*proof*⟩

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

lemma *LList-corec-type*: *LList-corec a f ∈ llist UNIV*
⟨*proof*⟩

11.3 *llist* equality as a *gfp*; the bisimulation principle

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

lemma *LListD-unfold*: *LListD r = dsum (diag {Numb 0}) (dprod r (LListD r))*
⟨*proof*⟩

lemma *LListD-implies-ntrunc-equality [rule-format]*:
∀ M N. (M,N) ∈ LListD(diag A) --> ntrunc k M = ntrunc k N
⟨*proof*⟩

The domain of the *LListD* relation

lemma *Domain-LListD*:
Domain (LListD(diag A)) <= llist(A)
⟨*proof*⟩

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

lemma *LListD-subset-diag*: *LListD(diag A) <= diag(llist(A))*
⟨*proof*⟩

11.3.1 Coinduction, using *LListD-Fun*

THE COINDUCTIVE DEFINITION PACKAGE COULD DO THIS!

lemma *LListD-Fun-mono*: *A<=B ==> LListD-Fun r A <= LListD-Fun r B*
⟨*proof*⟩

lemma *LListD-coinduct*:
[| $M \in X$; X <= *LListD-Fun r* (X *Un LListD*(r)) |] ==> $M \in$ *LListD*(r)
⟨*proof*⟩

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

lemma *LListD-Fun-CONS-I*:
[| $x \in A$; (M,N):s |] ==> (*CONS x M*, *CONS x N*) $\in$ *LListD-Fun* (*diag A*) *s*
⟨*proof*⟩

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

lemma *LListD-Fun-LListD-I*:
$M \in$ *LListD*(r) ==> $M \in$ *LListD-Fun r* (X *Un LListD*(r))
⟨*proof*⟩

This converse inclusion helps to strengthen *LList-equalityI*

lemma *diag-subset-LListD*: *diag*(*llist*(A)) <= *LListD*(*diag A*)
⟨*proof*⟩

lemma *LListD-eq-diag*: *LListD*(*diag A*) = *diag*(*llist*(A))
⟨*proof*⟩

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

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

lemma *LList-equalityI*:
[| (M,N) $\in r$; r <= *LListD-Fun* (*diag A*) (r *Un diag*(*llist*(A))) |]
==> M=N
⟨*proof*⟩

11.4 Finality of *llist*(A): Uniqueness of functions defined by corecursion

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

declare *Pair-eq* [*simp del*]

abstract proof using a bisimulation

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

lemma *equals-LList-corec*:
[| !!x. *h*(x) = (*case f x of None* => *NIL* | *Some*(z,w) => *CONS z* (*h w*)) |]
==> *h* = (%x. *LList-corec x f*)
⟨*proof*⟩

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

lemma *ntrunc-one-CONS* [*simp*]: *ntrunc* (*Suc 0*) (*CONS M N*) = {}
⟨*proof*⟩

lemma *ntrunc-CONS* [*simp*]:
ntrunc (*Suc*(*Suc*(k))) (*CONS M N*) = *CONS* (*ntrunc k M*) (*ntrunc k N*)
⟨*proof*⟩

lemma
assumes *prem1*:
!!x. *h1 x* = (*case f x of None* => *NIL* | *Some*(z,w) => *CONS z* (*h1 w*))
and *prem2*:
!!x. *h2 x* = (*case f x of None* => *NIL* | *Some*(z,w) => *CONS z* (*h2 w*))
shows *h1*=*h2*
⟨*proof*⟩

11.5 Lconst: defined directly by *lfp*

But it could be defined by corecursion.

lemma *Lconst-fun-mono*: *mono*(*CONS*(M))
⟨*proof*⟩

Lconst(M) = *CONS M* (*Lconst M*)

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

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

lemma *Lconst-type*: M∈A ==> *Lconst*(M): *llist*(A)
⟨*proof*⟩

lemma *Lconst-eq-LList-corec*: *Lconst*(M) = *LList-corec M* (%x. *Some*(x,x))
⟨*proof*⟩

Thus we could have used gfp in the definition of Lconst

lemma *gfp-Lconst-eq-LList-corec*: $gfp(\%N.\ CONS\ M\ N) = LList\text{-}corec\ M\ (\%x.\ Some(x,x))$
⟨*proof*⟩

11.6 Isomorphisms

lemma *LListI*: $x \in llist\ (range\ Leaf) ==> x \in LList$
⟨*proof*⟩

lemma *LListD*: $x \in LList ==> x \in llist\ (range\ Leaf)$
⟨*proof*⟩

11.6.1 Distinctness of constructors

lemma *LCons-not-LNil* [*iff*]: $\tilde{}\ LCons\ x\ xs = LNil$
⟨*proof*⟩

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

11.6.2 llist constructors

lemma *Rep-LList-LNil*: $Rep\text{-}LList\ LNil = NIL$
⟨*proof*⟩

lemma *Rep-LList-LCons*: $Rep\text{-}LList(LCons\ x\ l) = CONS\ (Leaf\ x)\ (Rep\text{-}LList\ l)$
⟨*proof*⟩

11.6.3 Injectiveness of *CONS* and *LCons*

lemma *CONS-CONS-eq2*: $(CONS\ M\ N{=}CONS\ M'\ N') = (M{=}M'\ \&\ N{=}N')$
⟨*proof*⟩

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

For reasoning about abstract llist constructors

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

lemma *LCons-LCons-eq* [*iff*]: $(LCons\ x\ xs{=}LCons\ y\ ys) = (x{=}y\ \&\ xs{=}ys)$
⟨*proof*⟩

lemma *CONS-D2*: $CONS\ M\ N \in llist(A) ==> M \in A\ \&\ N \in llist(A)$
⟨*proof*⟩

11.7 Reasoning about *llist*(*A*)

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

lemma *LList-fun-equalityI*:

[| $M \in llist(A)$; $g(NIL)$: $llist(A)$;
$f(NIL)=g(NIL)$;
!!x l. [| $x\in A$; $l \in llist(A)$ |] ==>
$(f(CONS\ x\ l),g(CONS\ x\ l)) \in$
$LListD\text{-}Fun\ (diag\ A)\ ((\%u.(f(u),g(u)))`llist(A)\ Un$
$diag(llist(A)))$
|] ==> $f(M) = g(M)$
⟨proof⟩

11.8 The functional *Lmap*

lemma *Lmap-NIL* [*simp*]: *Lmap f NIL = NIL*
⟨proof⟩

lemma *Lmap-CONS* [*simp*]: *Lmap f (CONS M N) = CONS (f M) (Lmap f N)*
⟨proof⟩

Another type-checking proof by coinduction

lemma *Lmap-type*:
[| $M \in llist(A)$; !!x. $x\in A$ ==> $f(x)$:B |] ==> *Lmap f M* $\in llist(B)$
⟨proof⟩

This type checking rule synthesises a sufficiently large set for *f*

lemma *Lmap-type2*: $M \in llist(A)$ ==> *Lmap f M* $\in llist(f`A)$
⟨proof⟩

11.8.1 Two easy results about *Lmap*

lemma *Lmap-compose*: $M \in llist(A)$ ==> *Lmap (f o g) M = Lmap f (Lmap g M)*
⟨proof⟩

lemma *Lmap-ident*: $M \in llist(A)$ ==> *Lmap (%x. x) M = M*
⟨proof⟩

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

lemma *Lappend-NIL-NIL* [*simp*]: *Lappend NIL NIL = NIL*
⟨proof⟩

lemma *Lappend-NIL-CONS* [*simp*]:
Lappend NIL (CONS N N') = CONS N (Lappend NIL N')
⟨proof⟩

lemma *Lappend-CONS* [*simp*]:
Lappend (CONS M M') N = CONS M (Lappend M' N)
⟨proof⟩

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

lemma *Lappend-NIL* [*simp*]: $M \in llist(A)$ ==> *Lappend NIL M* = *M*
⟨*proof*⟩

lemma *Lappend-NIL2*: $M \in llist(A)$ ==> *Lappend M NIL* = *M*
⟨*proof*⟩

11.9.1 Alternative type-checking proofs for *Lappend*

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

lemma *Lappend-type*: [| $M \in llist(A)$; $N \in llist(A)$ |] ==> *Lappend M N* $\in llist(A)$
⟨*proof*⟩

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

lemma *Lappend-type'*: [| $M \in llist(A)$; $N \in llist(A)$ |] ==> *Lappend M N* $\in$ $llist(A)$
⟨*proof*⟩

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

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

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

lemma *llist-case-LNil* [*simp*]: *llist-case c d LNil* = *c*
⟨*proof*⟩

lemma *llist-case-LCons* [*simp*]:
llist-case c d (*LCons M N*) = *d M N*
⟨*proof*⟩

Elimination is case analysis, not induction.

lemma *llistE*: [| *l*=*LNil* ==> *P*; !!*x l'*. *l*=*LCons x l'* ==> *P* |] ==> *P*
⟨*proof*⟩

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

Lemma for the proof of *llist-corec*

lemma *LList-corec-type2*:
LList-corec a
(%*z*. *case f z of None* => *None* | *Some*(*v*,*w*) => *Some*(*Leaf*(*v*),*w*))
$\in$ *llist*(*range Leaf*)

⟨*proof*⟩

lemma *llist-corec*:
llist-corec a f =
(*case f a of None* => *LNil* | *Some*(*z*,*w*) => *LCons z* (*llist-corec w f*))
⟨*proof*⟩

definitional version of same

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

11.11 Proofs about type *'a llist* functions

11.12 Deriving *llist-equalityI* − *llist* equality is a bisimulation

lemma *LListD-Fun-subset-Times-llist*:
r <= (*llist A*) <*> (*llist A*)
==> *LListD-Fun* (*diag A*) *r* <= (*llist A*) <*> (*llist A*)
⟨*proof*⟩

lemma *subset-Times-llist*:
prod-fun Rep-LList Rep-LList ' *r* <=
(*llist*(*range Leaf*)) <*> (*llist*(*range Leaf*))
⟨*proof*⟩

lemma *prod-fun-lemma*:
r <= (*llist*(*range Leaf*)) <*> (*llist*(*range Leaf*))
==> *prod-fun* (*Rep-LList o Abs-LList*) (*Rep-LList o Abs-LList*) ' *r* <= *r*
⟨*proof*⟩

lemma *prod-fun-range-eq-diag*:
prod-fun Rep-LList Rep-LList ' *range*(%*x*. (*x*, *x*)) =
diag(*llist*(*range Leaf*))
⟨*proof*⟩

Used with *lfilter*

lemma *llistD-Fun-mono*:
A<=*B* ==> *llistD-Fun A* <= *llistD-Fun B*
⟨*proof*⟩

11.12.1 To show two llists are equal, exhibit a bisimulation! [also admits true equality]

lemma *llist-equalityI*:
[| (*l1*,*l2*) ∈ *r*; *r* <= *llistD-Fun*(*r Un range*(%*x*.(*x*,*x*))) |] ==> *l1*=*l2*
⟨*proof*⟩

11.12.2 Rules to prove the 2nd premise of *llist-equalityI*

lemma *llistD-Fun-LNil-I* [*simp*]: (*LNil*,*LNil*) $\in$ *llistD-Fun*(*r*)
⟨*proof*⟩

lemma *llistD-Fun-LCons-I* [*simp*]:
  (*l1*,*l2*):*r* ==> (*LCons x l1*, *LCons x l2*) $\in$ *llistD-Fun*(*r*)
⟨*proof*⟩

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

lemma *llistD-Fun-range-I*: (*l*,*l*) $\in$ *llistD-Fun*(*r Un range*(%*x*.(*x*,*x*)))
⟨*proof*⟩

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

lemma *llist-fun-equalityI*:
  [| *f*(*LNil*)=*g*(*LNil*);
    !!*x l*. (*f*(*LCons x l*),*g*(*LCons x l*))
      $\in$ *llistD-Fun*(*range*(%*u*. (*f*(*u*),*g*(*u*))) *Un range*(%*v*. (*v*,*v*)))
  |] ==> *f*(*l*) = (*g*(*l* :: '*a llist*) :: '*b llist*)
⟨*proof*⟩

11.13 The functional *lmap*

lemma *lmap-LNil* [*simp*]: *lmap f LNil* = *LNil*
⟨*proof*⟩

lemma *lmap-LCons* [*simp*]: *lmap f* (*LCons M N*) = *LCons* (*f M*) (*lmap f N*)
⟨*proof*⟩

11.13.1 Two easy results about *lmap*

lemma *lmap-compose* [*simp*]: *lmap* (*f o g*) *l* = *lmap f* (*lmap g l*)
⟨*proof*⟩

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

11.14 iterates – *llist-fun-equalityI* cannot be used!

lemma *iterates*: *iterates f x* = *LCons x* (*iterates f* (*f x*))
⟨*proof*⟩

lemma *lmap-iterates* [*simp*]: *lmap f* (*iterates f x*) = *iterates f* (*f x*)
⟨*proof*⟩

lemma *iterates-lmap*: *iterates f x* = *LCons x* (*lmap f* (*iterates f x*))
⟨*proof*⟩

11.15 A rather complex proof about iterates – cf Andy Pitts

11.15.1 Two lemmas about *natrec n x* (%*m*. *g*), which is essentially (*g*ˆ*n*)(*x*)

lemma *fun-power-lmap*: *nat-rec* (*LCons b l*) (%*m*. *lmap*(*f*)) *n* =
  LCons (*nat-rec b* (%*m*. *f*) *n*) (*nat-rec l* (%*m*. *lmap*(*f*)) *n*)
⟨*proof*⟩

lemma *fun-power-Suc*: *nat-rec* (*g x*) (%*m*. *g*) *n* = *nat-rec x* (%*m*. *g*) (*Suc n*)
⟨*proof*⟩

lemmas *Pair-cong* = *refl* [*THEN cong*, *THEN cong*, *of* **concl**: *Pair*]

The bisimulation consists of {(*lmap*(*f*) ˆ*n* (*h*(*u*)), *lmap*(*f*) ˆ*n* (*iterates*(*f*,*u*)))} for all *u* and all *n*::*nat*.

lemma *iterates-equality*:
  (!!*x*. *h*(*x*) = *LCons x* (*lmap f* (*h x*))) ==> *h* = *iterates*(*f*)
⟨*proof*⟩

11.16 *lappend* – its two arguments cause some complications!

lemma *lappend-LNil-LNil* [*simp*]: *lappend LNil LNil* = *LNil*
⟨*proof*⟩

lemma *lappend-LNil-LCons* [*simp*]:
  lappend LNil (*LCons l l′*) = *LCons l* (*lappend LNil l′*)
⟨*proof*⟩

lemma *lappend-LCons* [*simp*]:
  lappend (*LCons l l′*) *N* = *LCons l* (*lappend l′ N*)
⟨*proof*⟩

lemma *lappend-LNil* [*simp*]: *lappend LNil l* = *l*
⟨*proof*⟩

lemma *lappend-LNil2* [*simp*]: *lappend l LNil* = *l*
⟨*proof*⟩

The infinite first argument blocks the second

lemma *lappend-iterates* [*simp*]: *lappend* (*iterates f x*) *N* = *iterates f x*
⟨*proof*⟩

11.16.1 Two proofs that *lmap* distributes over lappend

Long proof requiring case analysis on both both arguments

lemma *lmap-lappend-distrib*:
  lmap f (*lappend l n*) = *lappend* (*lmap f l*) (*lmap f n*)
⟨*proof*⟩

Shorter proof of theorem above using *llist-equalityI* as strong coinduction

lemma *lmap-lappend-distrib'*:
  lmap f (lappend l n) = lappend (lmap f l) (lmap f n)
⟨*proof*⟩

Without strong coinduction, three case analyses might be needed

lemma *lappend-assoc'*: *lappend (lappend l1 l2) l3 = lappend l1 (lappend l2 l3)*
⟨*proof*⟩

end

12 The "filter" functional for coinductive lists – defined by a combination of induction and coinduction

theory *LFilter* **imports** *LList* **begin**

consts
 findRel :: *('a => bool) => ('a llist * 'a llist)set*

inductive *findRel p*
 intros
  found: *p x ==> (LCons x l, LCons x l) ∈ findRel p*
  seek: *[| ~p x; (l,l') ∈ findRel p |] ==> (LCons x l, l') ∈ findRel p*

declare *findRel.intros [intro]*

constdefs
 find :: *['a => bool, 'a llist] => 'a llist*
  find p l == @l'. (l,l'): findRel p | (l' = LNil & l ~: Domain(findRel p))

 lfilter :: *['a => bool, 'a llist] => 'a llist*
  lfilter p l == llist-corec l (%l. case find p l of
            LNil => None
           | LCons y z => Some(y,z))

12.1 *findRel*: basic laws

inductive-cases
 findRel-LConsE [elim!]: *(LCons x l, l'') ∈ findRel p*

lemma *findRel-functional [rule-format]*:
  (l,l'): findRel p ==> (l,l''): findRel p --> l'' = l'
⟨*proof*⟩

lemma *findRel-imp-LCons* [*rule-format*]:
(l,l'): *findRel* p ==> $\exists\, x\ l''.\ l' = LCons\ x\ l''$ & $p\ x$
⟨*proof*⟩

lemma *findRel-LNil* [*elim*!]: (*LNil*,*l*): *findRel* p ==> R
⟨*proof*⟩

12.2 Properties of *Domain* (*findRel p*)

lemma *LCons-Domain-findRel* [*simp*]:
LCons $x\ l \in$ *Domain*(*findRel* p) = ($p\ x$ | $l \in$ *Domain*(*findRel* p))
⟨*proof*⟩

lemma *Domain-findRel-iff*:
($l \in$ *Domain* (*findRel* p)) = ($\exists\, x\ l'$. (l, *LCons* $x\ l'$) $\in$ *findRel* p & $p\ x$)
⟨*proof*⟩

lemma *Domain-findRel-mono*:
[| !!x. $p\ x$ ==> $q\ x$ |] ==> *Domain* (*findRel* p) <= *Domain* (*findRel* q)
⟨*proof*⟩

12.3 *find*: basic equations

lemma *find-LNil* [*simp*]: *find* p *LNil* = *LNil*
⟨*proof*⟩

lemma *findRel-imp-find* [*simp*]: $(l,l') \in$ *findRel* p ==> *find* $p\ l = l'$
⟨*proof*⟩

lemma *find-LCons-found*: $p\ x$ ==> *find* p (*LCons* $x\ l$) = *LCons* $x\ l$
⟨*proof*⟩

lemma *diverge-find-LNil* [*simp*]: l ~: *Domain*(*findRel* p) ==> *find* $p\ l$ = *LNil*
⟨*proof*⟩

lemma *find-LCons-seek*: ~ ($p\ x$) ==> *find* p (*LCons* $x\ l$) = *find* $p\ l$
⟨*proof*⟩

lemma *find-LCons* [*simp*]:
find p (*LCons* $x\ l$) = (*if* $p\ x$ *then LCons* $x\ l$ *else find* $p\ l$)
⟨*proof*⟩

12.4 *lfilter*: basic equations

lemma *lfilter-LNil* [*simp*]: *lfilter* p *LNil* = *LNil*
⟨*proof*⟩

lemma *diverge-lfilter-LNil* [*simp*]:
l ~: *Domain*(*findRel* p) ==> *lfilter* $p\ l$ = *LNil*
⟨*proof*⟩

lemma *lfilter-LCons-found*:
  $p\ x$ ==> *lfilter* p (*LCons* x l) = *LCons* x (*lfilter* p l)
⟨*proof*⟩

lemma *findRel-imp-lfilter* [*simp*]:
  (l, *LCons* x l') ∈ *findRel* p ==> *lfilter* p l = *LCons* x (*lfilter* p l')
⟨*proof*⟩

lemma *lfilter-LCons-seek*: ~ (p x) ==> *lfilter* p (*LCons* x l) = *lfilter* p l
⟨*proof*⟩

lemma *lfilter-LCons* [*simp*]:
  lfilter p (*LCons* x l) =
    (*if* p x *then* *LCons* x (*lfilter* p l) *else* *lfilter* p l)
⟨*proof*⟩

declare *llistD-Fun-LNil-I* [*intro*!] *llistD-Fun-LCons-I* [*intro*!]

lemma *lfilter-eq-LNil*: *lfilter* p l = *LNil* ==> l ~: *Domain*(*findRel* p)
⟨*proof*⟩

lemma *lfilter-eq-LCons* [*rule-format*]:
  lfilter p l = *LCons* x l' -->
      (∃ l''. l' = *lfilter* p l'' & (l, *LCons* x l'') ∈ *findRel* p)
⟨*proof*⟩

lemma *lfilter-cases*: *lfilter* p l = *LNil* |
    (∃ y l'. *lfilter* p l = *LCons* y (*lfilter* p l') & p y)
⟨*proof*⟩

12.5 *lfilter*: simple facts by coinduction

lemma *lfilter-K-True*: *lfilter* (%x. *True*) l = l
⟨*proof*⟩

lemma *lfilter-idem*: *lfilter* p (*lfilter* p l) = *lfilter* p l
⟨*proof*⟩

12.6 Numerous lemmas required to prove *lfilter-conj*

lemma *findRel-conj-lemma* [*rule-format*]:
  (l,l') ∈ *findRel* q
   ==> l' = *LCons* x l'' --> p x --> (l,l') ∈ *findRel* (%x. p x & q x)
⟨*proof*⟩

lemmas *findRel-conj* = *findRel-conj-lemma* [*OF* - *refl*]

lemma *findRel-not-conj-Domain* [*rule-format*]:
$(l,l'') \in \mathit{findRel}\ (\%x.\ p\ x\ \&\ q\ x)$
$==> (l,\ \mathit{LCons}\ x\ l') \in \mathit{findRel}\ q\ -->\ \tilde{}\ p\ x\ -->$
$l' \in \mathit{Domain}\ (\mathit{findRel}\ (\%x.\ p\ x\ \&\ q\ x))$
⟨*proof*⟩

lemma *findRel-conj2* [*rule-format*]:
$(l,\mathit{lxx}) \in \mathit{findRel}\ q$
$==> \mathit{lxx} = \mathit{LCons}\ x\ \mathit{lx}\ --> (\mathit{lx},\mathit{lz}) \in \mathit{findRel}(\%x.\ p\ x\ \&\ q\ x)\ -->\ \tilde{}\ p\ x$
$--> (l,\mathit{lz}) \in \mathit{findRel}\ (\%x.\ p\ x\ \&\ q\ x)$
⟨*proof*⟩

lemma *findRel-lfilter-Domain-conj* [*rule-format*]:
$(\mathit{lx},\mathit{ly}) \in \mathit{findRel}\ p$
$==> \forall\, l.\ \mathit{lx} = \mathit{lfilter}\ q\ l\ --> l \in \mathit{Domain}\ (\mathit{findRel}(\%x.\ p\ x\ \&\ q\ x))$
⟨*proof*⟩

lemma *findRel-conj-lfilter* [*rule-format*]:
$(l,l'') \in \mathit{findRel}(\%x.\ p\ x\ \&\ q\ x)$
$==> l'' = \mathit{LCons}\ y\ l'\ -->$
$(\mathit{lfilter}\ q\ l,\ \mathit{LCons}\ y\ (\mathit{lfilter}\ q\ l')) \in \mathit{findRel}\ p$
⟨*proof*⟩

lemma *lfilter-conj-lemma*:
$(\mathit{lfilter}\ p\ (\mathit{lfilter}\ q\ l),\ \mathit{lfilter}\ (\%x.\ p\ x\ \&\ q\ x)\ l)$
$\in \mathit{llistD\text{-}Fun}\ (\mathit{range}\ (\%u.\ (\mathit{lfilter}\ p\ (\mathit{lfilter}\ q\ u),$
$\mathit{lfilter}\ (\%x.\ p\ x\ \&\ q\ x)\ u)))$
⟨*proof*⟩

lemma *lfilter-conj*: $\mathit{lfilter}\ p\ (\mathit{lfilter}\ q\ l) = \mathit{lfilter}\ (\%x.\ p\ x\ \&\ q\ x)\ l$
⟨*proof*⟩

12.7 Numerous lemmas required to prove ??: $\mathit{lfilter}\ p\ (\mathit{lmap}\ f\ l) = \mathit{lmap}\ f\ (\mathit{lfilter}\ (\%x.\ p(f\ x))\ l)$

lemma *findRel-lmap-Domain*:
$(l,l') \in \mathit{findRel}(\%x.\ p\ (f\ x)) ==> \mathit{lmap}\ f\ l \in \mathit{Domain}(\mathit{findRel}\ p)$
⟨*proof*⟩

lemma *lmap-eq-LCons* [*rule-format*]: $\mathit{lmap}\ f\ l = \mathit{LCons}\ x\ l'\ -->$
$(\exists\, y\ l''.\ x = f\ y\ \&\ l' = \mathit{lmap}\ f\ l''\ \&\ l = \mathit{LCons}\ y\ l'')$
⟨*proof*⟩

lemma *lmap-LCons-findRel-lemma* [*rule-format*]:
$(\mathit{lx},\mathit{ly}) \in \mathit{findRel}\ p$
$==> \forall\, l.\ \mathit{lmap}\ f\ l = \mathit{lx}\ --> \mathit{ly} = \mathit{LCons}\ x\ l'\ -->$

$(\exists\, y\ l''.\ x = f\,y\ \&\ l' = lmap\ f\ l''\ \&$
$(l,\ LCons\ y\ l'') \in findRel(\%x.\ p(f\,x)))$
$\langle proof \rangle$

lemmas *lmap-LCons-findRel* = *lmap-LCons-findRel-lemma* [*OF* - *refl refl*]

lemma *lfilter-lmap*: *lfilter p* (*lmap f l*) = *lmap f* (*lfilter* (*p o f*) *l*)
$\langle proof \rangle$

end