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*]: [| $a \in A$; $t \in$ *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*
  apply (*induct set*: *tiling*)
  apply (*auto simp add*: *Un-assoc*)
  done

Chess boards

lemma *Sigma-Suc1* [*simp*]:
  lessThan (*Suc n*) $\times$ $B = (\{n\} \times B) \cup ((lessThan\ n) \times B)$
  by (*auto simp add*: *lessThan-def*)

lemma *Sigma-Suc2* [*simp*]:
  $A \times$ *lessThan* (*Suc n*) $= (A \times \{n\}) \cup (A \times (lessThan\ n))$
  by (*auto simp add*: *lessThan-def*)

```
lemma sing-Times-lemma: ({i} × {n}) ∪ ({i} × {m}) = {(i, m), (i, n)}
  by auto

lemma dominoes-tile-row [intro!]: {i} × lessThan (2 * n) ∈ tiling domino
  apply (induct n)
   apply (simp-all add: Un-assoc [symmetric])
  apply (rule tiling.Un)
    apply (auto simp add: sing-Times-lemma)
  done

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

coloured and Dominoes

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

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

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

Tilings of dominoes

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

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

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

```
  apply blast
  done
```

Final argument is surprisingly complex

```
theorem gen-mutil-not-tiling:
       t ∈ tiling domino ==>
         (i + j) mod 2 = 0 ==> (m + n) mod 2 = 0 ==>
         {(i, j), (m, n)} ⊆ t
       ==> (t − {(i, j)} − {(m, n)}) ∉ tiling domino
  apply (rule notI)
  apply (subgoal-tac
          card (whites ∩ (t − {(i, j)} − {(m, n)})) <
           card (blacks ∩ (t − {(i, j)} − {(m, n)})))
   apply (force simp only: tiling-domino-0-1)
  apply (simp add: tiling-domino-0-1 [symmetric])
  apply (simp add: coloured-def card-Diff2-less)
  done
```

Apply the general theorem to the well-known case

```
theorem mutil-not-tiling:
       t = lessThan (2 ∗ Suc m) × lessThan (2 ∗ Suc n)
         ==> t − {(0, 0)} − {(Suc (2 ∗ m), Suc (2 ∗ n))} ∉ tiling domino
  apply (rule gen-mutil-not-tiling)
     apply (blast intro!: dominoes-tile-matrix)
    apply auto
  done

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*) ∼ *X*
    DC: *DECRYPT K* (*CRYPT K X*) ∼ *X*
    NONCE: *NONCE N* ∼ *NONCE N*
    MPAIR: ⟦$X \sim X'$; $Y \sim Y'$⟧ ⟹ *MPAIR X Y* ∼ *MPAIR X′ Y′*
    CRYPT: $X \sim X'$ ⟹ *CRYPT K X* ∼ *CRYPT K X′*
    DECRYPT: $X \sim X'$ ⟹ *DECRYPT K X* ∼ *DECRYPT K X′*
    SYM: $X \sim Y \Longrightarrow Y \sim X$
    TRANS: ⟦$X \sim Y$; $Y \sim Z$⟧ ⟹ $X \sim Z$

Proving that it is an equivalence relation

lemma *msgrel-refl*: $X \sim X$
by (*induct X*, (*blast intro*: *msgrel.intros*)+)

theorem *equiv-msgrel*: *equiv UNIV msgrel*
proof (*simp add*: *equiv-def*, *intro conjI*)
  show *reflexive msgrel* **by** (*simp add*: *refl-def msgrel-refl*)
  show *sym msgrel* **by** (*simp add*: *sym-def*, *blast intro*: *msgrel.SYM*)
  show *trans msgrel* **by** (*simp add*: *trans-def*, *blast intro*: *msgrel.TRANS*)
qed

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*: $U \sim V \Longrightarrow freenonces\ U = freenonces\ V$
by (*erule msgrel.induct*, *auto*)

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*:
$U \sim V \Longrightarrow freeleft\ U \sim freeleft\ V$
by (*erule msgrel.induct*, *auto intro*: *msgrel.intros*)

2.2.3 The Right Projection

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

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

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

theorem *msgrel-imp-eqv-freeright*:
$U \sim V \Longrightarrow freeright\ U \sim freeright\ V$
by (*erule msgrel.induct*, *auto intro*: *msgrel.intros*)

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 $Nonce\ N \neq MPair\ X\ Y$

theorem *msgrel-imp-eq-freediscrim*:
  $U \sim V \Longrightarrow freediscrim\ U = freediscrim\ V$
by (*erule msgrel.induct*, *auto*)

2.3 The Initial Algebra: A Quotiented Message Type

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

The abstract message constructors

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

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

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

  Decrypt :: [*nat*,*msg*] ⇒ *msg*
  Decrypt K X ==
    Abs-Msg ($\bigcup$ *U* ∈ *Rep-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*} ∈ *Msg*
by (*auto simp add*: *Msg-def quotient-def intro*: *msgrel-refl*)

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

Reduces equality on abstractions to equality on representatives

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

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

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 −
have (λ*U V*. *msgrel* `` {*MPAIR U V*}) *respects2 msgrel*
by (*simp add*: *congruent2-def msgrel.MPAIR*)
thus *?thesis*
by (*simp add*: *MPair-def UN-equiv-class2* [*OF equiv-msgrel equiv-msgrel*])
qed

lemma *Crypt*: *Crypt K* (*Abs-Msg*(*msgrel*``{*U*})) = *Abs-Msg* (*msgrel*``{*CRYPT K U*})
proof −
have (λ*U*. *msgrel* `` {*CRYPT K U*}) *respects msgrel*
by (*simp add*: *congruent-def msgrel.CRYPT*)
thus *?thesis*
by (*simp add*: *Crypt-def UN-equiv-class* [*OF equiv-msgrel*])
qed

lemma *Decrypt*:
Decrypt K (*Abs-Msg*(*msgrel*``{*U*})) = *Abs-Msg* (*msgrel*``{*DECRYPT K U*})
proof −
have (λ*U*. *msgrel* `` {*DECRYPT K U*}) *respects msgrel*
by (*simp add*: *congruent-def msgrel.DECRYPT*)
thus *?thesis*
by (*simp add*: *Decrypt-def UN-equiv-class* [*OF equiv-msgrel*])
qed

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*
apply (*rule Rep-Msg* [*of z*, *unfolded Msg-def*, *THEN quotientE*])
apply (*drule arg-cong* [**where** *f*=*Abs-Msg*])
apply (*auto simp add*: *Rep-Msg-inverse intro*: *msgrel-refl*)
done

Establishing these two equations is the point of the whole exercise

theorem *CD-eq* [*simp*]: *Crypt K* (*Decrypt K X*) = *X*
by (*cases X*, *simp add*: *Crypt Decrypt CD*)

theorem *DC-eq* [*simp*]: *Decrypt K* (*Crypt K X*) = *X*
by (*cases X*, *simp add*: *Crypt Decrypt DC*)

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

Now prove the four equations for *nonces*

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

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

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

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

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*
by (*simp add*: *congruent-def msgrel-imp-eqv-freeleft*)

Now prove the four equations for *left*

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

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

done

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

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

2.6 The Abstract Function to Return the Right Part

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

lemma *right-congruent*: (λ*U*. *msgrel* ‘‘ {*freeright U*}) *respects msgrel*
by (*simp add*: *congruent-def msgrel-imp-eqv-freeright*)

Now prove the four equations for *right*

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

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

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

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

2.7 Injectivity Properties of Some Constructors

lemma *NONCE-imp-eq*: *NONCE m* ∼ *NONCE n* ⟹ *m* = *n*

by (*drule msgrel-imp-eq-freenonces*, *simp*)

Can also be proved using the function *nonces*

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

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

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

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

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

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

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

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

Example suggested by a referee

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

...and many similar results

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

theorem *Crypt-Crypt-eq* [*iff*]: (*Crypt* K X = *Crypt* K X') = (X=X')
proof

```
  assume Crypt K X = Crypt K X′
  hence Decrypt K (Crypt K X) = Decrypt K (Crypt K X′) by simp
  thus X = X′ by simp
next
  assume X = X′
  thus Crypt K X = Crypt K X′ by simp
qed

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

lemma msg-induct [case-names Nonce MPair Crypt Decrypt, cases type: msg]:
  assumes N: ⋀N. P (Nonce N)
    and M: ⋀X Y. ⟦P X; P Y⟧ ⟹ P (MPair X Y)
    and C: ⋀K X. P X ⟹ P (Crypt K X)
    and D: ⋀K X. P X ⟹ P (Decrypt K X)
  shows P msg
proof (cases msg, erule ssubst)
  fix U::freemsg
  show P (Abs-Msg (msgrel `` {U}))
  proof (induct U)
    case (NONCE N)
    with N show ?case by (simp add: Nonce-def)
  next
    case (MPAIR X Y)
    with M [of Abs-Msg (msgrel `` {X}) Abs-Msg (msgrel `` {Y})]
    show ?case by (simp add: MPair)
  next
    case (CRYPT K X)
    with C [of Abs-Msg (msgrel `` {X})]
    show ?case by (simp add: Crypt)
  next
    case (DECRYPT K X)
    with D [of Abs-Msg (msgrel `` {X})]
    show ?case by (simp add: Decrypt)
  qed
qed
```

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

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

Now prove the four equations for *discrim*

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

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

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

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

end

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** $\sim$ *50*)
syntax (*HTML* **output**)
-*exprel* :: [*freeExp*, *freeExp*] => *bool* (**infixl** $\sim$ *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*)
apply (*induct X* **and** *Xs*)
apply (*blast intro*: *exprel.intros listrel.intros*)+
done

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

theorem *equiv-exprel*: *equiv UNIV exprel*
proof (*simp add*: *equiv-def*, *intro conjI*)
show *reflexive exprel* **by** (*simp add*: *refl-def exprel-refl*)
show *sym exprel* **by** (*simp add*: *sym-def*, *blast intro*: *exprel.SYM*)
show *trans exprel* **by** (*simp add*: *trans-def*, *blast intro*: *exprel.TRANS*)
qed

theorem *equiv-list-exprel*: *equiv UNIV* (*listrel exprel*)
by (*insert equiv-listrel* [*OF equiv-exprel*], *simp*)

lemma *FNCALL-Nil*: *FNCALL F* [] $\sim$ *FNCALL F* []
apply (*rule exprel.intros*)
apply (*rule listrel.intros*)

done

lemma *FNCALL-Cons*:
$[\![X \sim X'; (Xs,Xs') \in listrel(exprel)]\!]$
$\Longrightarrow FNCALL\ F\ (X\#Xs) \sim FNCALL\ F\ (X'\#Xs')$
by (*blast intro*: *exprel.intros listrel.intros*)

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*
apply (*erule exprel.induct*)
apply (*erule-tac* [*4*] *listrel.induct*)
apply (*simp-all add*: *Un-assoc*)
done

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*:
$U \sim V \Longrightarrow$ *freediscrim U* = *freediscrim V*
by (*erule exprel.induct*, *auto*)

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*:
    $U \sim V \Longrightarrow$ *freefun U* = *freefun V*
by (*erule exprel.induct*, *simp-all add*: *listrel.intros*)

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

consts *freeargs* :: *freeExp* ⇒ *freeExp list*
primrec
  freeargs (*VAR N*) = []
  freeargs (*PLUS X Y*) = []
  freeargs (*FNCALL F Xs*) = *Xs*

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

3.3 The Initial Algebra: A Quotiented Message Type

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

The abstract message constructors

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

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

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

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

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

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

All equivalence classes belong to set of representatives

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

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

Reduces equality on abstractions to equality on representatives

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

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

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

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

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

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

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

lemma *eq-Abs-ExpList* [*case-names Abs-ExpList*]:

(!! *Us*. *z* = *Abs-ExpList Us* ==> *P*) ==> *P*
by (*rule exE* [*OF ExpList-rep*], *blast*)

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 −
have (λ*U V*. *exprel* `` {*PLUS U V*}) *respects2 exprel*
by (*simp add*: *congruent2-def exprel.PLUS*)
thus *?thesis*
by (*simp add*: *Plus-def UN-equiv-class2* [*OF equiv-exprel equiv-exprel*])
qed

It is not clear what to do with FnCall: it's argument is an abstraction of an *exp list*. Is it just Nil or Cons? What seems to work best is to regard an *exp list* as a *listrel exprel* equivalence class

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

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

lemma *FnCall-respects*:
(λ*Us*. *exprel* `` {*FNCALL F Us*}) *respects* (*listrel exprel*)
by (*simp add*: *congruent-def exprel.FNCALL*)

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

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

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

Establishing this equation is the point of the whole exercise

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

3.5 The Abstract Function to Return the Set of Variables

constdefs
vars :: *exp* ⇒ *nat set*
vars X == $\bigcup$ *U* ∈ *Rep-Exp X*. *freevars U*

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

The extension of the function *vars* to lists

consts *vars-list* :: *exp list* ⇒ *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*}
by (*simp add*: *vars-def Var-def*
UN-equiv-class [*OF equiv-exprel vars-respects*])

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

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

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

lemma *vars-FnCall-Cons*: *vars* (*FnCall F* (*X*#*Xs*)) = *vars X* ∪ *vars-list Xs*
by *simp*

3.6 Injectivity Properties of Some Constructors

lemma *VAR-imp-eq*: *VAR m* ∼ *VAR n* ⟹ *m* = *n*
by (*drule exprel-imp-eq-freevars*, *simp*)

Can also be proved using the function *vars*

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

lemma *VAR-neqv-PLUS*: *VAR m* ∼ *PLUS X Y* ⟹ *False*
by (*drule exprel-imp-eq-freediscrim*, *simp*)

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

theorem *Var-neq-FnCall* [*iff*]: *Var N* ≠ *FnCall F Xs*
apply (*cases Xs rule*: *eq-Abs-ExpList*)
apply (*auto simp add*: *FnCall Var-def*)
apply (*drule exprel-imp-eq-freediscrim*, *simp*)
done

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*
by (*simp add*: *congruent-def exprel-imp-eq-freefun*)

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

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*)
by (*erule listrel.induct*, *simp-all*)

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

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

lemma *FnCall-FnCall-eq* [*iff*]:
  (*FnCall F Xs* = *FnCall F′ Xs′*) = (*F*=*F′* & *Xs*=*Xs′*)
proof
  assume *FnCall F Xs* = *FnCall F′ Xs′*
  hence *fun* (*FnCall F Xs*) = *fun* (*FnCall F′ Xs′*)
    and *args* (*FnCall F Xs*) = *args* (*FnCall F′ Xs′*) **by** *auto*
  thus *F*=*F′* & *Xs*=*Xs′* **by** *simp*
next
  assume *F*=*F′* & *Xs*=*Xs′* **thus** *FnCall F Xs* = *FnCall F′ Xs′* **by** *simp*
qed

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* (⋃ *U* ∈ *Rep-Exp X*. {*freediscrim U*})

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

Now prove the four equations for *discrim*

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

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

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

The structural induction rule for the abstract type

theorem *exp-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 (cases exp, rule eq-Abs-ExpList [of list], clarify)
  fix U Us
  show P1 (Abs-Exp (exprel `` {U})) ∧
       P2 (Abs-ExpList Us)
  proof (induct U and Us)
    case (VAR nat)
    with V show ?case by (simp add: Var-def)
  next
    case (PLUS X Y)
    with P [of Abs-Exp (exprel `` {X}) Abs-Exp (exprel `` {Y})]
    show ?case by (simp add: Plus)
  next
    case (FNCALL nat list)
    with F [of Abs-ExpList list]
    show ?case by (simp add: FnCall)
  next
    case Nil-freeExp
    with Nil show ?case by simp
  next
    case Cons-freeExp
     with Cons
    show ?case by simp
  qed
qed

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 ∘ 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)
  by (induct t and ts) simp-all
```

Alternative induction rule

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

end
```

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

lemma *subst-all-aexp*:
  $evala\ env\ (substa\ s\ a) = evala\ (\lambda x.\ evala\ env\ (s\ x))\ a$
and *subst-all-bexp*:
  $evalb\ env\ (substb\ s\ b) = evalb\ (\lambda x.\ evala\ env\ (s\ x))\ b$
  by (*induct a* **and** *b*) *auto*

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 (λx. map-tree f (ts x))

lemma tree-map-compose: map-tree g (map-tree f t) = map-tree (g ∘ f) t
  by (induct t) simp-all

consts
  exists-tree :: ('a => bool) => 'a tree => bool
primrec
  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)
  by (induct ts) auto
```

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)
by (induct k, auto)
```

Multiplication of ordinals

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

lemma add-mult-distrib: mult i (add j k) = add (mult i j) (mult i k)
apply (induct k)
```

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

lemma *mult-assoc*: *mult* (*mult i j*) *k* = *mult i* (*mult j k*)
apply (*induct k*)
apply (*auto simp add*: *add-mult-distrib*)
done

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*ˆ+

lemma *wf-brouwer-pred*: *wf brouwer-pred*
  by(*unfold wf-def brouwer-pred-def*, *clarify*, *induct-tac x*, *blast*+)

lemma *wf-brouwer-order*: *wf brouwer-order*
  by(*unfold brouwer-order-def*, *rule wf-trancl*[*OF wf-brouwer-pred*])

lemma [*simp*]: (*j*, *Succ j*) : *brouwer-order*
  by(*auto simp add*: *brouwer-order-def brouwer-pred-def*)

lemma [*simp*]: (*f n*, *Lim f*) : *brouwer-order*
  by(*auto simp add*: *brouwer-order-def brouwer-pred-def*)

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*))
by (*induct k*, *auto*)

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
  $iter\ f\ 0 = id$
  $iter\ f\ (Suc\ n) = f \circ (iter\ f\ n)$

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

consts
  $cantor :: ordinal => ordinal => ordinal$
primrec
  $cantor\ a\ Zero = Succ\ a$
  $cantor\ a\ (Succ\ b) = \bigsqcup(\lambda x.\ cantor\ x\ b)\ a$
  $cantor\ a\ (Limit\ f) = Limit\ (\lambda n.\ cantor\ a\ (f\ n))$

consts
  $Nabla :: (ordinal => ordinal) => (ordinal => ordinal) \quad (\nabla)$
primrec
  $\nabla f\ Zero = f\ Zero$
  $\nabla f\ (Succ\ a) = f\ (Succ\ (\nabla f\ a))$
  $\nabla f\ (Limit\ h) = Limit\ (\lambda n.\ \nabla f\ (h\ n))$

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: $a \in A$ ==> $a \in$ *σ-algebra A*
    UNIV: *UNIV* $\in$ *σ-algebra A*
    complement: $a \in$ *σ-algebra A* ==> $-a \in$ *σ-algebra A*
    Union: (!!*i*::*nat*. *a i* $\in$ *σ-algebra A*) ==> ($\bigcup i$. *a i*) $\in$ *σ-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*: {} $\in$ *σ-algebra A*
proof −
  have *UNIV* $\in$ *σ-algebra A* **by** (*rule σ-algebra.UNIV*)
  hence −*UNIV* $\in$ *σ-algebra A* **by** (*rule σ-algebra.complement*)
  also have −*UNIV* = {} **by** *simp*
  finally show *?thesis* **.**
qed

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

assume $!!i::nat.\ a\ i \in \sigma\text{-}algebra\ A$
hence $!!i::nat.\ -(a\ i) \in \sigma\text{-}algebra\ A$ **by** $(rule\ \sigma\text{-}algebra.complement)$
hence $(\bigcup i.\ -(a\ i)) \in \sigma\text{-}algebra\ A$ **by** $(rule\ \sigma\text{-}algebra.Union)$
hence $-(\bigcup i.\ -(a\ i)) \in \sigma\text{-}algebra\ A$ **by** $(rule\ \sigma\text{-}algebra.complement)$
also have $-(\bigcup i.\ -(a\ i)) = (\bigcap i.\ a\ i)$ **by** *simp*
finally show *?thesis* **.**
qed

end

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*ˆ∗

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*ˆ∗ |] ==>
∀ *y*'. (*x*,*y*') ∈ *r* −−> (∃ *z*. (*y*',*z*) ∈ *r*ˆ∗ & (*y*,*z*) ∈ *r*)
apply (*unfold diamond-def*)
apply (*erule rtrancl-induct*)
apply (*meson rtrancl-refl*)
apply (*meson rtrancl-trans r-into-rtrancl*)
done

lemma *diamond-rtrancl*: *diamond*(*r*) ==> *diamond*(*r*ˆ∗)
apply (*simp* (*no-asm-simp*) *add*: *diamond-def*)
apply (*rule impI* [*THEN allI*, *THEN allI*])
apply (*erule rtrancl-induct*, *blast*)
apply (*meson rtrancl-trans r-into-rtrancl diamond-strip-lemmaE*)
done

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*
by (*unfold I-def*, *blast*)

lemma *K1-contractD* [*elim*!]: *K*##*x* −*1*−> *z* ==> (∃ *x*′. *z* = *K*##*x*′ & *x* −*1*−> *x*′)
by *blast*

lemma *Ap-reduce1* [*intro*]: *x* −−−> *y* ==> *x*##*z* −−−> *y*##*z*
apply (*erule rtrancl-induct*)
apply (*blast intro*: *rtrancl-trans*)+
done

lemma *Ap-reduce2* [*intro*]: *x* −−−> *y* ==> *z*##*x* −−−> *z*##*y*
apply (*erule rtrancl-induct*)
apply (*blast intro*: *rtrancl-trans*)+
done

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

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

lemma *KIII-contract3*: *K*##*I*##((*K*##*I*)##(*K*##*I*)) −*1*−> *I*
by *blast*

lemma *not-diamond-contract*: ~ *diamond*(*contract*)
apply (*unfold diamond-def*)
apply (*best intro*: *KIII-contract1 KIII-contract2 KIII-contract3*)

done

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')
by *blast*

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

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

The rules above are not essential but make proofs much faster

Church-Rosser property for parallel contraction

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

Equivalence of *p* −−−> *q* and *p* ===> *q*.

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

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

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

lemma *reduce-I*: $I\#\#x$ $--->$ x
by (*unfold I-def*, *blast*)

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

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

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

end

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 $H \mid= p$ then $G \mid= p$ where $G \in Fin(H)$

10.1 The datatype of propositions

datatype
'*a pl* = *false* | *var* '*a* (#- [*1000*]) | −> '*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
$H \mid- p == p \in thms(H)$

inductive *thms*(*H*)
intros
H [*intro*]: $p{\in}H$ ==> $H \mid- p$
K: $H \mid- p{-}{>}q{-}{>}p$
S: $H \mid- (p{-}{>}q{-}{>}r) -{>} (p{-}{>}q) -{>} p{-}{>}r$

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* ∈ *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*

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 |= *p* == (∀ *tt*. (∀ *q*∈*H*. *tt*[[*q*]]) −−> *tt*[[*p*]])

10.4 Proof theory of propositional logic

lemma *thms-mono*: *G*<=*H* ==> *thms*(*G*) <= *thms*(*H*)
apply (*unfold thms.defs*)
apply (*rule lfp-mono*)
apply (*assumption* | *rule basic-monos*)+
done

lemma *thms-I*: H |− *p*−>*p*
— Called I for Identity Combinator, not for Introduction.
by (*best intro*: *thms.K thms.S thms.MP*)

10.4.1 Weakening, left and right

lemma *weaken-left*: [| *G* ⊆ *H*; *G*|−*p* |] ==> *H*|−*p*
— Order of premises is convenient with *THEN*
by (*erule thms-mono* [*THEN subsetD*])

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*
by (*fast intro*: *thms.K thms.MP*)

10.4.2 The deduction theorem

theorem *deduction*: *insert p H* |− *q* ==> *H* |− *p*−>*q*
apply (*erule thms.induct*)
apply (*fast intro*: *thms-I thms.H thms.K thms.S thms.DN*
thms.S [*THEN thms.MP*, *THEN thms.MP*] *weaken-right*)+
done

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* |− *p* ==> *H* |= *p*
apply (*unfold sat-def*)
apply (*erule thms.induct*, *auto*)
done

10.5 Completeness

10.5.1 Towards the completeness proof

lemma *false-imp*: *H* |− *p*−>*false* ==> *H* |− *p*−>*q*
apply (*rule deduction*)
apply (*erule weaken-left-insert* [*THEN thms-notE*])
apply *blast*
done

lemma *imp-false*:
[| *H* |− *p*; *H* |− *q*−>*false* |] ==> *H* |− (*p*−>*q*)−>*false*
apply (*rule deduction*)
apply (*blast intro*: *weaken-left-insert thms.MP thms.H*)
done

lemma *hyps-thms-if*: *hyps p tt* |− (*if tt*[[*p*]] *then p else p*−>*false*)
— Typical example of strengthening the induction statement.
apply *simp*
apply (*induct-tac p*)

apply (*simp-all add*: *thms-I thms.H*)
apply (*blast intro*: *weaken-left-Un1 weaken-left-Un2 weaken-right*
imp-false false-imp)
done

lemma *sat-thms-p*: {} |= *p* ==> *hyps p tt* |− *p*
— Key lemma for completeness; yields a set of assumptions satisfying *p*
apply (*unfold sat-def*)
apply (*drule spec*, *erule mp* [*THEN if-P*, *THEN subst*],
rule-tac [*2*] *hyps-thms-if*, *simp*)
done

For proving certain theorems in our new propositional logic.

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

The excluded middle in the form of an elimination rule.

lemma *thms-excluded-middle*: *H* |− (*p*−>*q*) −> ((*p*−>*false*)−>*q*) −> *q*
apply (*rule deduction* [*THEN deduction*])
apply (*rule thms.DN* [*THEN thms.MP*], *best*)
done

lemma *thms-excluded-middle-rule*:
[| *insert p H* |− *q*; *insert* (*p*−>*false*) *H* |− *q* |] ==> *H* |− *q*
— Hard to prove directly because it requires cuts
by (*rule thms-excluded-middle* [*THEN thms.MP*, *THEN thms.MP*], *auto*)

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*} ⊆ *hyps p* (*t* − {*v*}).

lemma *hyps-Diff*: *hyps p* (*t*−{*v*}) <= *insert* (#*v*−>*false*) ((*hyps p t*)−{#*v*})
by (*induct-tac p*, *auto*)

For the case *hyps p t* − *insert* (#*v* −> *Fls*) *Y* |− *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*})
by (*induct-tac p*, *auto*)

Two lemmas for use with *weaken-left*

lemma *insert-Diff-same*: *B*−*C* <= *insert a* (*B*−*insert a C*)
by *fast*

lemma *insert-Diff-subset2*: *insert a* (*B*−{*c*}) − *D* <= *insert a* (*B*−*insert c D*)
by *fast*

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

lemma *hyps-finite*: *finite*(*hyps p t*)
by (*induct-tac p*, *auto*)

lemma *hyps-subset*: *hyps p t* <= (*UN v*. {#*v*, #*v*−>*false*})
by (*induct-tac p*, *auto*)

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*
apply (*rule hyps-subset* [*THEN hyps-finite* [*THEN finite-subset-induct*]])
 apply (*simp add*: *sat-thms-p*, *safe*)

Case *hyps p t*−*insert*(#*v*,*Y*) |− *p*

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

Case *hyps p t*−*insert*(#*v* −> *false*,*Y*) |− *p*

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

The base case for completeness

lemma *completeness-0*: {} |= *p* ==> {} |− *p*
apply (*rule Diff-cancel* [*THEN subst*])
apply (*erule completeness-0-lemma* [*THEN spec*])
done

A semantic analogue of the Deduction Theorem

lemma *sat-imp*: *insert p H* |= *q* ==> *H* |= *p*−>*q*
by (*unfold sat-def*, *auto*)

theorem *completeness* [*rule-format*]: *finite H* ==> ∀ *p*. *H* |= *p* −−> *H* |− *p*
apply (*erule finite-induct*)
 apply (*blast intro*: *completeness-0*)
apply (*iprover intro*: *sat-imp thms.H insertI1 weaken-left-insert* [*THEN thms.MP*])
done

theorem *syntax-iff-semantics*: *finite H* ==> (*H* |− *p*) = (*H* |= *p*)
by (*blast intro*: *soundness completeness*)

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.   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 == ⋃ M ∈ sexp. ⋃ N ∈ 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*)
by (*simp add*: *sexp-case-def*, *blast*)

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

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

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

apply (simp add: In0-def)
apply (erule sexp.NumbI [THEN sexp.SconsI])
done

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

declare sexp.intros [intro,simp]

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

lemma Scons-D: Scons M N ∈ sexp ==> M ∈ sexp & N ∈ sexp
apply (erule setup-induction)
apply (erule sexp.induct, blast+)
done

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

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
by (simp add: pred-sexp-def, blast)

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

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

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

declare cut-apply [simp]

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

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

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

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

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

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

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

end
```

theory *SList* **imports** *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
  by (blast intro: list.NIL-I)

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*
by (*simp add*: *List-def*)

lemma *ListD*: *x* : *List* ==> *x* : *list* (*range Leaf*)
by (*simp add*: *List-def*)

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

lemma *list-mono*: *A*<=*B* ==> *list*(*A*) <= *list*(*B*)
apply (*unfold list.defs*)
apply (*rule lfp-mono*)
apply (*assumption* | *rule basic-monos*)+
done

lemma *list-sexp*: *list*(*sexp*) <= *sexp*
apply (*unfold NIL-def CONS-def list.defs*)
apply (*rule lfp-lowerbound*)
apply (*fast intro*: *sexp.intros sexp-In0I sexp-In1I*)
done

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

lemma *list-induct*:
  [| *P*(*Nil*);
     !!*x xs*. *P*(*xs*) ==> *P*(*x* # *xs*) |] ==> *P*(*l*)
apply (*unfold Nil-def Cons-def*)
apply (*rule Rep-List-inverse* [*THEN subst*])

apply (*rule Rep-List* [*unfolded List-def*, *THEN list.induct*], *simp*)
apply (*erule Abs-List-inverse* [*unfolded List-def*, *THEN subst*], *blast*)
done

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

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

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

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

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

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

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

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

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

lemma *CONS-D*: *CONS M N*: *list*(*A*) ==> *M*: *A* & *N*: *list*(*A*)
apply (*erule setup-induction*)
apply (*erule list.induct*, *blast*+)
done

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

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

lemma *not-Cons-self2*: ∀ *x*. *l* ~= *x*#*l*
by (*induct-tac l rule*: *list-induct*, *simp-all*)

lemma *neq-Nil-conv2*: (*xs* ~= []) = (∃ *y ys*. *xs* = *y*#*ys*)
by (*induct-tac xs rule*: *list-induct*, *auto*)

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

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

lemma *List-rec-unfold-lemma*:
(%*M*. *List-rec M c d*) ==
wfrec (*trancl pred-sexp*) (%*g*. *List-case c* (%*x y*. *d x y* (*g y*)))
by (*simp add*: *List-rec-def*)

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*^+
by (*simp add*: *CONS-def In1-def*)

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

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

done

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

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

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

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

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

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

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

lemma *Rep-map-Cons* [*simp*]:
Rep-map f(*x*#*xs*) = *CONS*(*f x*)(*Rep-map f xs*)
by (*simp add*: *Rep-map-def*)

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

lemma *Abs-map-NIL* [*simp*]: *Abs-map g NIL* = *Nil*
by (*simp add*: *Abs-map-def*)

lemma *Abs-map-CONS* [*simp*]:
[| *M*: *sexp*; *N*: *sexp* |] ==> *Abs-map g* (*CONS M N*) = *g*(*M*) # *Abs-map g N*
by (*simp add*: *Abs-map-def*)

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

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

Better to have a single theorem with a conjunctive conclusion.

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

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

declare *def-list-rec-NilCons* [*OF null-def*, *simp*]
declare *def-list-rec-NilCons* [*OF hd-def*, *simp*]
declare *def-list-rec-NilCons* [*OF tl-def*, *simp*]
declare *def-list-rec-NilCons* [*OF ttl-def*, *simp*]
declare *def-list-rec-NilCons* [*OF append-def*, *simp*]
declare *def-list-rec-NilCons* [*OF member-def*, *simp*]
declare *def-list-rec-NilCons* [*OF map-def*, *simp*]
declare *def-list-rec-NilCons* [*OF filter-def*, *simp*]
declare *def-list-rec-NilCons* [*OF list-all-def*, *simp*]

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

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

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

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

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

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

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

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

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

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

lemma *list-all-True* [*simp*]: (*Alls x*:*xs*. *True*) = *True*
by (*induct-tac xs rule*: *list-induct*, *simp-all*)

lemma *list-all-conj* [*simp*]:
  list-all p (*xs*@*ys*) = ((*list-all p xs*) & (*list-all p ys*))
by (*induct-tac xs rule*: *list-induct*, *simp-all*)

lemma *list-all-mem-conv*: (*Alls x*:*xs*. *P*(*x*)) = (!*x*. *x mem xs* --> *P*(*x*))
apply (*induct-tac xs rule*: *list-induct*, *simp-all*)
apply *blast*
done

lemma *nat-case-dist* : (! *n*. *P n*) = (*P 0* & (! *n*. *P* (*Suc n*)))
apply *auto*
apply (*induct-tac n*, *auto*)
done

lemma *alls-P-eq-P-nth*: (*Alls u*:*A*. *P u*) = (!*n*. *n* < *length A* --> *P*(*nth n A*))
apply (*induct-tac A rule*: *list-induct*, *simp-all*)
apply (*rule trans*)
apply (*rule-tac* [*2*] *nat-case-dist* [*symmetric*], *simp-all*)
done

lemma *list-all-imp*:
  [| !*x*. *P x* --> *Q x*; (*Alls x*:*xs*. *P*(*x*)) |] ==> (*Alls x*:*xs*. *Q*(*x*))
by (*simp add*: *list-all-mem-conv*)

lemma *Abs-Rep-map*:
  (!!*x*. *f*(*x*): *sexp*) ==>
    Abs-map g (*Rep-map f xs*) = *map* (%*t*. *g*(*f*(*t*))) *xs*
apply (*induct-tac xs rule*: *list-induct*)
apply (*simp-all add*: *Rep-map-type list-sexp* [*THEN subsetD*])
done

lemma *map-ident* [*simp*]: *map*(%*x*. *x*)(*xs*) = *xs*
by (*induct-tac xs rule*: *list-induct*, *simp-all*)

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

lemma *map-compose*: *map*(*f o g*)(*xs*) = *map f* (*map g xs*)
apply (*simp add*: *o-def*)
apply (*induct-tac xs rule*: *list-induct*, *simp-all*)
done

lemma *mem-map-aux1* [*rule-format*]:
  x mem (*map f q*) −−> (∃ *y*. *y mem q* & *x* = *f y*)
by (*induct-tac q rule*: *list-induct*, *simp-all*, *blast*)

lemma *mem-map-aux2* [*rule-format*]:
  (∃ *y*. *y mem q* & *x* = *f y*) −−> *x mem* (*map f q*)
by (*induct-tac q rule*: *list-induct*, *auto*)

lemma *mem-map*: *x mem* (*map f q*) = (∃ *y*. *y mem q* & *x* = *f y*)
apply (*rule iffI*)
apply (*erule mem-map-aux1*)
apply (*erule mem-map-aux2*)
done

lemma *hd-append* [*rule-format*]: *A* ~= [] −−> *hd*(*A* @ *B*) = *hd*(*A*)
by (*induct-tac A rule*: *list-induct*, *auto*)

lemma *tl-append* [*rule-format*]: *A* ~= [] −−> *tl*(*A* @ *B*) = *tl*(*A*) @ *B*
by (*induct-tac A rule*: *list-induct*, *auto*)

lemma *take-Suc1* [*simp*]: *take* [] (*Suc x*) = []
by *simp*

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

lemma *drop-0* [*simp*]: *drop xs 0* = *xs*
by (*simp add*: *drop-def*)

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

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

lemma *copy-0* [*simp*]: *copy x 0* = []
by (*simp add*: *copy-def*)

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

lemma *foldl-Nil* [*simp*]: *foldl f a* [] = *a*
by (*simp add*: *foldl-def*)

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

lemma *foldr-Nil* [*simp*]: *foldr f a* [] = *a*
by (*simp add*: *foldr-def*)

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

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

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

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

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

lemma *zipWith-Cons-Cons* [*simp*]:
zipWith f (*a*#*as*,*b*#*bs*) = *f*(*a*,*b*) # *zipWith f* (*as*,*bs*)

by (*simp add*: *zipWith-def*)

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

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

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

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

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

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

lemma *list-all-map-eq*: (*Alls u*:*xs*. *f*(*u*) = *g*(*u*)) −−> *map f xs* = *map g xs*
by (*induct-tac xs rule*: *list-induct*, *simp-all*)

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

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

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

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

lemma *length-map*: *length*(*map f xs*) = *length*(*xs*)
by (*induct-tac xs rule*: *list-induct*, *simp-all*)

lemma *take-Nil* [*simp*]: *take* [] *n* = []
by (*induct-tac n*, *simp-all*)

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

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

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

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

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

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

lemma *drop-Nil* [*simp*]: *drop* [] *n* = []
by (*induct-tac n*, *auto*)

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

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

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

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

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

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

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

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

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

done

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *foldr-append2*:

[| !a. f e a = a; !a b c. f a (f b c) = f(f a b) c |]
==> *foldr* f e (A @ B) = f (*foldr* f e A) (*foldr* f e B)
apply *auto*
apply (*rule foldr-neutr-distr*, *auto*)
done

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

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

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

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

lemma *nth-app-cancel-right* [*rule-format*]:
$\forall i$. i < *length*(A) --> *nth* i(A@B) = *nth* i A
apply (*induct-tac* A *rule*: *list-induct*, *simp-all*)
apply (*rule allI*)
apply (*induct-tac* i, *simp-all*)
done

lemma *nth-app-cancel-left* [*rule-format*]:
$\forall n$. n = *length*(A) --> *nth*(n+i)(A@B) = *nth* i B
by (*induct-tac* A *rule*: *list-induct*, *simp-all*)

lemma *flat-append* [*simp*]: *flat*(*xs*@*ys*) = *flat*(*xs*) @ *flat*(*ys*)
by (*induct-tac xs rule*: *list-induct*, *auto*)

lemma *filter-flat*: *filter* p (*flat* S) = *flat*(*map* (*filter* p) S)
by (*induct-tac* S *rule*: *list-induct*, *auto*)

```
lemma rev-append [simp]: rev(xs@ys) = rev(ys) @ rev(xs)
by (induct-tac xs rule: list-induct, auto)

lemma rev-rev-ident [simp]: rev(rev l) = l
by (induct-tac l rule: list-induct, auto)

lemma rev-flat: rev(flat ls) = flat (map rev (rev ls))
by (induct-tac ls rule: list-induct, auto)

lemma rev-map-distrib: rev(map f l) = map f (rev l)
by (induct-tac l rule: list-induct, auto)

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

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

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 ∈ llist(A)
    CONS-I:         [| a ∈ A;  M ∈ llist(A) |] ==> CONS a M ∈ llist(A)

coinductive LListD(r)
  intros
    NIL-I:  (NIL, NIL) ∈ LListD(r)
    CONS-I:         [| (a,b) ∈ r;  (M,N) ∈ LListD(r) |]
                 ==> (CONS a M, CONS b N) ∈ LListD(r)
```

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

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* | (∃ *M a*. *z* = *CONS a M* & *a* ∈ *A* & *M* ∈ *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*) |
(∃ *M N a b*. *z* = (*CONS a M*, *CONS b N*) & (*a*, *b*) ∈ *r* & (*M*, *N*) ∈ *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)*
apply (*unfold llist.defs*)
apply (*rule gfp-mono*)
apply (*assumption | rule basic-monos*)+
done

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

11.1 Type checking by coinduction

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

lemma *llist-coinduct*:
  [| $M \in X$; X <= *list-Fun A* (X *Un llist*(A)) |] ==> $M \in$ *llist*(A)
apply (*unfold list-Fun-def*)
apply (*erule llist*.*coinduct*)
apply (*erule subsetD* [*THEN CollectD*], *assumption*)
done

lemma *list-Fun-NIL-I* [*iff*]: *NIL* $\in$ *list-Fun A X*
by (*unfold list-Fun-def NIL-def*, *fast*)

lemma *list-Fun-CONS-I* [*intro*!,*simp*]:
  [| $M \in A$; $N \in X$ |] ==> *CONS M N* $\in$ *list-Fun A X*
apply (*unfold list-Fun-def CONS-def*, *fast*)
done

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

lemma *list-Fun-llist-I*: $M \in$ *llist*(A) ==> $M \in$ *list-Fun A* (X *Un llist*(A))
apply (*unfold llist*.*defs list-Fun-def*)
apply (*rule gfp-fun-UnI2*)
apply (*rule monoI*, *blast*)
apply *assumption*
done

11.2 *LList-corec* satisfies the desired recurion equation

A continuity result?

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

lemma *CONS-mono*: [| M<=M'; N<=N' |] ==> *CONS M N* <= *CONS* M' N'
apply (*unfold CONS-def*)
apply (*assumption* | *rule In1-mono Scons-mono*)+
done

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

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))
apply *(unfold LList-corec-def)*
apply *(rule UN-least)*
apply *(case-tac k)*
apply *(simp-all (no-asm-simp))*
apply *(rule allI impI subset-refl [THEN CONS-mono] UNIV-I [THEN UN-upper])+*
done

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

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

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

definitional version of same

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

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

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

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))*
 by *(fast intro!: LListD.intros [unfolded NIL-def CONS-def]*
   elim: LListD.cases [unfolded NIL-def CONS-def])

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

The domain of the *LListD* relation

lemma *Domain-LListD*:
  Domain (*LListD*(*diag A*)) <= *llist*(*A*)
apply (*unfold llist.defs NIL-def CONS-def*)
apply (*rule gfp-upperbound*)

avoids unfolding *LListD* on the rhs

apply (*rule-tac P* = %*x*. *Domain x* <= *?B* **in** *LListD-unfold* [*THEN ssubst*], *simp*)
apply *blast*
done

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

lemma *LListD-subset-diag*: *LListD*(*diag A*) <= *diag*(*llist*(*A*))
apply (*rule subsetI*)
apply (*rule-tac p* = *x* **in** *PairE*, *safe*)
apply (*rule diag-eqI*)
apply (*rule LListD-implies-ntrunc-equality* [*THEN ntrunc-equality*], *assumption*)
apply (*erule DomainI* [*THEN Domain-LListD* [*THEN subsetD*]])
done

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*
apply (*unfold LListD-Fun-def*)
apply (*assumption* | *rule basic-monos*)+
done

lemma *LListD-coinduct*:
  [| $M \in X$; *X* <= *LListD-Fun r* (*X Un LListD*(*r*)) |] ==> $M \in LListD(r)$
apply (*unfold LListD-Fun-def*)
apply (*erule LListD.coinduct*)
apply (*erule subsetD* [*THEN CollectD*], *assumption*)
done

lemma *LListD-Fun-NIL-I*: (*NIL*,*NIL*) ∈ *LListD-Fun r s*
by (*unfold LListD-Fun-def NIL-def*, *fast*)

lemma *LListD-Fun-CONS-I*:
  [| *x*∈*A*; (*M*,*N*):*s* |] ==> (*CONS x M*, *CONS x N*) ∈ *LListD-Fun* (*diag A*) *s*
apply (*unfold LListD-Fun-def CONS-def*, *blast*)
done

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

lemma *LListD-Fun-LListD-I*:
  M ∈ *LListD*(*r*) ==> *M* ∈ *LListD-Fun r* (*X Un LListD*(*r*))
apply (*unfold LListD.defs LListD-Fun-def*)
apply (*rule gfp-fun-UnI2*)
apply (*rule monoI*, *blast*)
apply *assumption*
done

This converse inclusion helps to strengthen *LList-equalityI*

lemma *diag-subset-LListD*: *diag*(*llist*(*A*)) <= *LListD*(*diag A*)
apply (*rule subsetI*)
apply (*erule LListD-coinduct*)
apply (*rule subsetI*)
apply (*erule diagE*)
apply (*erule ssubst*)
apply (*erule llist.cases*)
apply (*simp-all add*: *diagI LListD-Fun-NIL-I LListD-Fun-CONS-I*)
done

lemma *LListD-eq-diag*: *LListD*(*diag A*) = *diag*(*llist*(*A*))
apply (*rule equalityI LListD-subset-diag diag-subset-LListD*)+
done

lemma *LListD-Fun-diag-I*: *M* ∈ *llist*(*A*) ==> (*M*,*M*) ∈ *LListD-Fun* (*diag A*) (*X Un diag*(*llist*(*A*)))
apply (*rule LListD-eq-diag* [*THEN subst*])
apply (*rule LListD-Fun-LListD-I*)
apply (*simp add*: *LListD-eq-diag diagI*)
done

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

done

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*
apply (*rule ext*)

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

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

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

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

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

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

lemma
 assumes *prem1*:
  !!*x*. *h1 x* = (*case f x of None* => *NIL* | *Some*(*z*,*w*) => *CONS z* (*h1 w*))
 and *prem2*:
  !!*x*. *h2 x* = (*case f x of None* => *NIL* | *Some*(*z*,*w*) => *CONS z* (*h2 w*))
 shows *h1*=*h2*
apply (*rule ntrunc-equality* [*THEN ext*])
apply (*rule-tac x* = *x* **in** *spec*)
apply (*induct-tac k rule*: *nat-less-induct*)

apply (*rename-tac n*)
apply (*rule allI*)
apply (*subst prem1*)
apply (*subst prem2*, *simp*)
apply (*intro strip*)
apply (*case-tac n*)
apply (*rename-tac* [*2*] *m*)
apply (*case-tac* [*2*] *m*)
apply *simp-all*
done

11.5 Lconst: defined directly by *lfp*

But it could be defined by corecursion.

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

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

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

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

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

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

Thus we could have used gfp in the definition of Lconst

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

11.6 Isomorphisms

lemma *LListI*: *x* ∈ *llist* (*range Leaf*) ==> *x* ∈ *LList*
by (*unfold LList-def*, *simp*)

lemma *LListD*: *x* ∈ *LList* ==> *x* ∈ *llist* (*range Leaf*)
by (*unfold LList-def*, *simp*)

11.6.1 Distinctness of constructors

lemma *LCons-not-LNil* [*iff*]: ~ *LCons x xs* = *LNil*
apply (*unfold LNil-def LCons-def*)
apply (*subst Abs-LList-inject*)
apply (*rule llist.intros CONS-not-NIL rangeI LListI Rep-LList* [*THEN LListD*])+
done

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

11.6.2 llist constructors

lemma *Rep-LList-LNil*: *Rep-LList LNil* = *NIL*
apply (*unfold LNil-def*)
apply (*rule llist.NIL-I* [*THEN LListI*, *THEN Abs-LList-inverse*])
done

lemma *Rep-LList-LCons*: *Rep-LList*(*LCons x l*) = *CONS* (*Leaf x*) (*Rep-LList l*)
apply (*unfold LCons-def*)
apply (*rule llist.CONS-I* [*THEN LListI*, *THEN Abs-LList-inverse*]
    rangeI Rep-LList [*THEN LListD*])+
done

11.6.3 Injectiveness of *CONS* and *LCons*

lemma *CONS-CONS-eq2*: (*CONS M N*=*CONS M′ N′*) = (*M*=*M′* & *N*=*N′*)
apply (*unfold CONS-def*)
apply (*fast elim*!: *Scons-inject*)
done

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

For reasoning about abstract llist constructors

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

lemma *LCons-LCons-eq* [*iff*]: (*LCons x xs*=*LCons y ys*) = (*x*=*y* & *xs*=*ys*)
apply (*unfold LCons-def*)
apply (*subst Abs-LList-inject*)
apply (*auto simp add*: *Rep-LList-inject*)
done

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

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-Fun (*diag* A) ((%u.($f(u)$,$g(u)$))`*llist*(A) *Un*
diag(*llist*(A)))
|] ==> $f(M) = g(M)$
apply (*rule LList-equalityI*)
apply (*erule imageI*)
apply (*rule image-subsetI*)
apply (*erule-tac aa=x* **in** *llist.cases*)
apply (*erule ssubst*, *erule ssubst*, *erule LListD-Fun-diag-I*, *blast*)
done

11.8 The functional *Lmap*

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

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

Another type-checking proof by coinduction

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

This type checking rule synthesises a sufficiently large set for f

lemma *Lmap-type2*: $M \in llist(A)$ ==> *Lmap f M* $\in$ *llist*(f`A)
apply (*erule Lmap-type*)
apply (*erule imageI*)
done

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)
apply (*unfold o-def*)
apply (*drule imageI*)
apply (*erule LList-equalityI*, *safe*)
apply (*erule llist.cases*, *simp-all*)
apply (*rule LListD-Fun-NIL-I imageI UnI1 rangeI* [*THEN LListD-Fun-CONS-I*])+
apply *assumption*
done

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

apply (*drule imageI*)
apply (*erule LList-equalityI*, *safe*)
apply (*erule llist.cases*, *simp-all*)
apply (*rule LListD-Fun-NIL-I imageI UnI1 rangeI* [*THEN LListD-Fun-CONS-I*])+
apply *assumption*
done

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

lemma *Lappend-NIL-NIL* [*simp*]: *Lappend NIL NIL* = *NIL*
apply (*unfold Lappend-def*)
apply (*rule LList-corec* [*THEN trans*], *simp*)
done

lemma *Lappend-NIL-CONS* [*simp*]:
  Lappend NIL (*CONS N N'*) = *CONS N* (*Lappend NIL N'*)
apply (*unfold Lappend-def*)
apply (*rule LList-corec* [*THEN trans*], *simp*)
done

lemma *Lappend-CONS* [*simp*]:
  Lappend (*CONS M M'*) *N* = *CONS M* (*Lappend M' N*)
apply (*unfold Lappend-def*)
apply (*rule LList-corec* [*THEN trans*], *simp*)
done

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

lemma *Lappend-NIL* [*simp*]: $M \in llist(A)$ ==> *Lappend NIL M* = *M*
by (*erule LList-fun-equalityI*, *simp-all*)

lemma *Lappend-NIL2*: $M \in llist(A)$ ==> *Lappend M NIL* = *M*
by (*erule LList-fun-equalityI*, *simp-all*)

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* $\in llist(A)$
apply (*rule-tac X* = $\bigcup u \in llist\ (A)\ .\ \bigcup v \in llist\ (A)\ .\ \{Lappend\ u\ v\}$ **in** *llist-coinduct*)
apply *fast*
apply *safe*
apply (*erule-tac aa* = *u* **in** *llist.cases*)
apply (*erule-tac aa* = *v* **in** *llist.cases*, *simp-all*)
apply *blast*
done

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)
apply (*rule-tac* X = (%*u*. *Lappend* *u* N) '*llist* (A) **in** *llist-coinduct*)
apply (*erule imageI*)
apply (*rule image-subsetI*)
apply (*erule-tac aa* = *x* **in** *llist.cases*)
apply (*simp add*: *list-Fun-llist-I*, *simp*)
done

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*
by (*unfold llist-case-def LNil-def*, *simp*)

lemma *llist-case-LCons* [*simp*]:
llist-case c d (*LCons M N*) = *d M N*
apply (*unfold llist-case-def LCons-def*, *simp*)
done

Elimination is case analysis, not induction.

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

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*)
apply (*rule-tac X* = *range* (%*x*. *LList-corec x ?g*) **in** *llist-coinduct*)
apply (*rule rangeI*, *safe*)
apply (*subst LList-corec*, *force*)
done

lemma *llist-corec*:

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

definitional version of same

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

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*)
by (*auto simp add*: *LListD-Fun-def*)

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

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

lemma *prod-fun-range-eq-diag*:
  prod-fun Rep-LList Rep-LList ‘ *range*(%*x*. (*x*, *x*)) =
  diag(*llist*(*range Leaf*))
apply (*rule equalityI*, *blast*)
apply (*fast elim*: *LListI* [*THEN Abs-LList-inverse*, *THEN subst*])
done

Used with *lfilter*

lemma *llistD-Fun-mono*:
  A<=*B* ==> *llistD-Fun A* <= *llistD-Fun B*
apply (*unfold llistD-Fun-def prod-fun-def*, *auto*)
apply (*rule image-eqI*)

prefer *2* **apply** (*blast intro*: *rev-subsetD* [*OF - LListD-Fun-mono*], *force*)
done

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*
apply (*unfold llistD-Fun-def*)
apply (*rule Rep-LList-inject* [*THEN iffD1*])
apply (*rule-tac r* = *prod-fun Rep-LList Rep-LList* '*r* **and** *A* = *range* (*Leaf*) **in** *LList-equalityI*)
apply (*erule prod-fun-imageI*)
apply (*erule image-mono* [*THEN subset-trans*])
apply (*rule image-compose* [*THEN subst*])
apply (*rule prod-fun-compose* [*THEN subst*])
apply (*subst image-Un*)
apply (*subst prod-fun-range-eq-diag*)
apply (*rule LListD-Fun-subset-Times-llist* [*THEN prod-fun-lemma*])
apply (*rule subset-Times-llist* [*THEN Un-least*])
apply (*rule diag-subset-Times*)
done

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

lemma *llistD-Fun-LNil-I* [*simp*]: (*LNil*,*LNil*) ∈ *llistD-Fun*(*r*)
apply (*unfold llistD-Fun-def LNil-def*)
apply (*rule LListD-Fun-NIL-I* [*THEN prod-fun-imageI*])
done

lemma *llistD-Fun-LCons-I* [*simp*]:
  (*l1*,*l2*):*r* ==> (*LCons x l1*, *LCons x l2*) ∈ *llistD-Fun*(*r*)
apply (*unfold llistD-Fun-def LCons-def*)
apply (*rule rangeI* [*THEN LListD-Fun-CONS-I*, *THEN prod-fun-imageI*])
apply (*erule prod-fun-imageI*)
done

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

lemma *llistD-Fun-range-I*: (*l*,*l*) ∈ *llistD-Fun*(*r Un range*(%*x*.(*x*,*x*)))
apply (*unfold llistD-Fun-def*)
apply (*rule Rep-LList-inverse* [*THEN subst*])
apply (*rule prod-fun-imageI*)
apply (*subst image-Un*)
apply (*subst prod-fun-range-eq-diag*)
apply (*rule Rep-LList* [*THEN LListD*, *THEN LListD-Fun-diag-I*])
done

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

lemma *llist-fun-equalityI*:

[| $f(LNil)=g(LNil)$;
!!x l. $(f(LCons\ x\ l),g(LCons\ x\ l))$
$\in$ *llistD-Fun*(*range*(%u. $(f(u),g(u))$) *Un range*(%v. (v,v)))
|] ==> $f(l) = (g(l :: 'a\ llist) :: 'b\ llist)$
apply (*rule-tac* r = *range* (%u. $(f\ (u),g\ (u))$) **in** *llist-equalityI*)
apply (*rule rangeI*, *clarify*)
apply (*rule-tac* $l = u$ **in** *llistE*)
apply (*simp-all add*: *llistD-Fun-range-I*)
done

11.13 The functional *lmap*

lemma *lmap-LNil* [*simp*]: *lmap f LNil = LNil*
by (*rule lmap-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)

lemma *lmap-LCons* [*simp*]: *lmap f* (*LCons M N*) = *LCons* (*f M*) (*lmap f N*)
by (*rule lmap-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)

11.13.1 Two easy results about *lmap*

lemma *lmap-compose* [*simp*]: *lmap* (*f o g*) *l* = *lmap f* (*lmap g l*)
by (*rule-tac* $l = l$ **in** *llist-fun-equalityI*, *simp-all*)

lemma *lmap-ident* [*simp*]: *lmap* (%x. x) $l = l$
by (*rule-tac* $l = l$ **in** *llist-fun-equalityI*, *simp-all*)

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

lemma *iterates*: *iterates f x* = *LCons x* (*iterates f* (*f x*))
by (*rule iterates-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)

lemma *lmap-iterates* [*simp*]: *lmap f* (*iterates f x*) = *iterates f* (*f x*)
apply (*rule-tac* r = *range* (%u. (*lmap f* (*iterates f u*),*iterates f* (*f u*))) **in** *llist-equalityI*)
apply (*rule rangeI*, *safe*)
apply (*rule-tac* $x1 = f\ (u)$ **in** *iterates* [*THEN ssubst*])
apply (*rule-tac* $x1 = u$ **in** *iterates* [*THEN ssubst*], *simp*)
done

lemma *iterates-lmap*: *iterates f x* = *LCons x* (*lmap f* (*iterates f x*))
apply (*subst lmap-iterates*)
apply (*rule iterates*)
done

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 $(g\hat{}n)(x)$

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)

apply (*induct-tac n*, *simp-all*)
done

lemma *fun-power-Suc*: *nat-rec* (*g x*) (%*m*. *g*) *n* = *nat-rec x* (%*m*. *g*) (*Suc n*)
by (*induct-tac n*, *simp-all*)

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

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

lemma *iterates-equality*:
  (!!*x*. *h*(*x*) = *LCons x* (*lmap f* (*h x*))) ==> *h* = *iterates*(*f*)
apply (*rule ext*)
apply (*rule-tac*
    $r = \bigcup u.$ *range* (%*n*. (*nat-rec* (*h u*) (%*m y*. *lmap f y*) *n*,
        nat-rec (*iterates f u*) (%*m y*. *lmap f y*) *n*))
    in *llist-equalityI*)
apply (*rule UN1-I range-eqI Pair-cong nat-rec-0* [*symmetric*])+
apply *clarify*
apply (*subst iterates*, *atomize*)
apply (*drule-tac x*=*u* **in** *spec*)
apply (*erule ssubst*)
apply (*subst fun-power-lmap*)
apply (*subst fun-power-lmap*)
apply (*rule llistD-Fun-LCons-I*)
apply (*rule lmap-iterates* [*THEN subst*])
apply (*subst fun-power-Suc*)
apply (*subst fun-power-Suc*, *blast*)
done

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

lemma *lappend-LNil-LNil* [*simp*]: *lappend LNil LNil* = *LNil*
apply (*unfold lappend-def*)
apply (*rule llist-corec* [*THEN trans*], *simp*)
done

lemma *lappend-LNil-LCons* [*simp*]:
  lappend LNil (*LCons l l'*) = *LCons l* (*lappend LNil l'*)
apply (*unfold lappend-def*)
apply (*rule llist-corec* [*THEN trans*], *simp*)
done

lemma *lappend-LCons* [*simp*]:
  lappend (*LCons l l'*) *N* = *LCons l* (*lappend l' N*)
apply (*unfold lappend-def*)
apply (*rule llist-corec* [*THEN trans*], *simp*)
done

lemma *lappend-LNil* [*simp*]: *lappend LNil l* = *l*
by (*rule-tac l* = *l* **in** *llist-fun-equalityI*, *simp-all*)

lemma *lappend-LNil2* [*simp*]: *lappend l LNil* = *l*
by (*rule-tac l* = *l* **in** *llist-fun-equalityI*, *simp-all*)

The infinite first argument blocks the second

lemma *lappend-iterates* [*simp*]: *lappend* (*iterates f x*) *N* = *iterates f x*
apply (*rule-tac r* = *range* (%*u*. (*lappend* (*iterates f u*) *N*,*iterates f u*))
  in *llist-equalityI*)
apply (*rule rangeI*)
apply (*safe*)
apply (*subst* (*1 2*) *iterates*)
apply *simp*
done

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*)
apply (*rule-tac r* = $\bigcup$*n*. *range* (%*l*. (*lmap f* (*lappend l n*),
  lappend (*lmap f l*) (*lmap f n*)))
  in *llist-equalityI*)
apply (*rule UN1-I*)
apply (*rule rangeI*, *safe*)
apply (*rule-tac l* = *l* **in** *llistE*)
apply (*rule-tac l* = *n* **in** *llistE*, *simp-all*)
apply (*blast intro*: *llistD-Fun-LCons-I*)
done

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

lemma *lmap-lappend-distrib′*:
 lmap f (*lappend l n*) = *lappend* (*lmap f l*) (*lmap f n*)
apply (*rule-tac l* = *l* **in** *llist-fun-equalityI*, *simp*)
apply *simp*
done

Without strong coinduction, three case analyses might be needed

lemma *lappend-assoc′*: *lappend* (*lappend l1 l2*) *l3* = *lappend l1* (*lappend l2 l3*)
apply (*rule-tac l* = *l1* **in** *llist-fun-equalityI*, *simp*)
apply *simp*
done

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: $p\ x ==> (LCons\ x\ l,\ LCons\ x\ l) \in findRel\ p$
    seek: $[|\ {}^{\sim}p\ x;\ (l,l') \in findRel\ p\ |] ==> (LCons\ x\ l,\ l') \in findRel\ p$

declare *findRel.intros* [*intro*]

constdefs
  find :: $['a => bool, 'a\ llist] => 'a\ llist$
    $find\ p\ l == @l'.\ (l,l'): findRel\ p\ |\ (l' = LNil\ \&\ l\ {}^{\sim}: Domain(findRel\ p))$

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

12.1 *findRel*: basic laws

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

lemma *findRel-functional* [*rule-format*]:
    $(l,l'): findRel\ p ==> (l,l''): findRel\ p\ --> l'' = l'$
by (*erule findRel.induct*, *auto*)

lemma *findRel-imp-LCons* [*rule-format*]:
    $(l,l'): findRel\ p ==> \exists\, x\ l''.\ l' = LCons\ x\ l''\ \&\ p\ x$
by (*erule findRel.induct*, *auto*)

lemma *findRel-LNil* [*elim!*]: $(LNil,l): findRel\ p ==> R$
by (*blast elim*: *findRel.cases*)

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

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

lemma *Domain-findRel-iff*:

(*l* ∈ *Domain* (*findRel p*)) = (∃ *x l′*. (*l*, *LCons x l′*) ∈ *findRel p* & *p x*)
by (*blast dest*: *findRel-imp-LCons*)

lemma *Domain-findRel-mono*:
[| !!*x*. *p x* ==> *q x* |] ==> *Domain* (*findRel p*) <= *Domain* (*findRel q*)
apply *clarify*
apply (*erule findRel.induct*, *blast*+)
done

12.3 *find*: basic equations

lemma *find-LNil* [*simp*]: *find p LNil* = *LNil*
by (*unfold find-def*, *blast*)

lemma *findRel-imp-find* [*simp*]: (*l*,*l′*) ∈ *findRel p* ==> *find p l* = *l′*
apply (*unfold find-def*)
apply (*blast dest*: *findRel-functional*)
done

lemma *find-LCons-found*: *p x* ==> *find p* (*LCons x l*) = *LCons x l*
by (*blast intro*: *findRel-imp-find*)

lemma *diverge-find-LNil* [*simp*]: *l* ~: *Domain*(*findRel p*) ==> *find p l* = *LNil*
by (*unfold find-def*, *blast*)

lemma *find-LCons-seek*: ~ (*p x*) ==> *find p* (*LCons x l*) = *find p l*
apply (*case-tac LCons x l* ∈ *Domain* (*findRel p*))
apply *auto*
apply (*blast intro*: *findRel-imp-find*)
done

lemma *find-LCons* [*simp*]:
find p (*LCons x l*) = (*if p x then LCons x l else find p l*)
by (*simp add*: *find-LCons-seek find-LCons-found*)

12.4 *lfilter*: basic equations

lemma *lfilter-LNil* [*simp*]: *lfilter p LNil* = *LNil*
by (*rule lfilter-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)

lemma *diverge-lfilter-LNil* [*simp*]:
l ~: *Domain*(*findRel p*) ==> *lfilter p l* = *LNil*
by (*rule lfilter-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)

lemma *lfilter-LCons-found*:
p x ==> *lfilter p* (*LCons x l*) = *LCons x* (*lfilter p l*)
by (*rule lfilter-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)

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

by (*rule lfilter-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)

lemma *lfilter-LCons-seek*: ~ (*p x*) ==> *lfilter p* (*LCons x l*) = *lfilter p l*
apply (*rule lfilter-def* [*THEN def-llist-corec*, *THEN trans*], *simp*)
apply (*case-tac LCons x l* ∈ *Domain* (*findRel p*))
 apply (*simp add*: *Domain-findRel-iff*, *auto*)
done

lemma *lfilter-LCons* [*simp*]:
  lfilter p (*LCons x l*) =
    (*if p x then LCons x* (*lfilter p l*) *else lfilter p l*)
by (*simp add*: *lfilter-LCons-found lfilter-LCons-seek*)

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

lemma *lfilter-eq-LNil*: *lfilter p l* = *LNil* ==> *l* ~: *Domain*(*findRel p*)
apply (*auto iff*: *Domain-findRel-iff*)
done

lemma *lfilter-eq-LCons* [*rule-format*]:
  lfilter p l = *LCons x l′* −−>
      (∃ *l″*. *l′* = *lfilter p l″* & (*l*, *LCons x l″*) ∈ *findRel p*)
apply (*subst lfilter-def* [*THEN def-llist-corec*])
apply (*case-tac l* ∈ *Domain* (*findRel p*))
 apply (*auto iff*: *Domain-findRel-iff*)
done

lemma *lfilter-cases*: *lfilter p l* = *LNil* |
    (∃ *y l′*. *lfilter p l* = *LCons y* (*lfilter p l′*) & *p y*)
apply (*case-tac l* ∈ *Domain* (*findRel p*))
apply (*auto iff*: *Domain-findRel-iff*)
done

12.5 *lfilter*: simple facts by coinduction

lemma *lfilter-K-True*: *lfilter* (%*x*. *True*) *l* = *l*
by (*rule-tac l* = *l* **in** *llist-fun-equalityI*, *simp-all*)

lemma *lfilter-idem*: *lfilter p* (*lfilter p l*) = *lfilter p l*
apply (*rule-tac l* = *l* **in** *llist-fun-equalityI*, *simp-all*)
apply *safe*

Cases: *p x* is true or false

apply (*rule lfilter-cases* [*THEN disjE*])
 apply (*erule ssubst*, *auto*)
done

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*)
by (*erule findRel.induct*, *auto*)

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

lemma *findRel-not-conj-Domain* [*rule-format*]:
(*l*,*l''*) ∈ *findRel* (%*x*. *p x* & *q x*)
==> (*l*, *LCons x l'*) ∈ *findRel q* --> ~ *p x* -->
l' ∈ *Domain* (*findRel* (%*x*. *p x* & *q x*))
by (*erule findRel.induct*, *auto*)

lemma *findRel-conj2* [*rule-format*]:
(*l*,*lxx*) ∈ *findRel q*
==> *lxx* = *LCons x lx* --> (*lx*,*lz*) ∈ *findRel*(%*x*. *p x* & *q x*) --> ~ *p x*
--> (*l*,*lz*) ∈ *findRel* (%*x*. *p x* & *q x*)
by (*erule findRel.induct*, *auto*)

lemma *findRel-lfilter-Domain-conj* [*rule-format*]:
(*lx*,*ly*) ∈ *findRel p*
==> ∀ *l*. *lx* = *lfilter q l* --> *l* ∈ *Domain* (*findRel*(%*x*. *p x* & *q x*))
apply (*erule findRel.induct*)
apply (*blast dest*!: *sym* [*THEN lfilter-eq-LCons*] *intro*: *findRel-conj*, *auto*)
apply (*drule sym* [*THEN lfilter-eq-LCons*], *auto*)
apply (*drule spec*)
apply (*drule refl* [*THEN rev-mp*])
apply (*blast intro*: *findRel-conj2*)
done

lemma *findRel-conj-lfilter* [*rule-format*]:
(*l*,*l''*) ∈ *findRel*(%*x*. *p x* & *q x*)
==> *l''* = *LCons y l'* -->
(*lfilter q l*, *LCons y* (*lfilter q l'*)) ∈ *findRel p*
by (*erule findRel.induct*, *auto*)

lemma *lfilter-conj-lemma*:
(*lfilter p* (*lfilter q l*), *lfilter* (%*x*. *p x* & *q x*) *l*)
∈ *llistD-Fun* (*range* (%*u*. (*lfilter p* (*lfilter q u*),
lfilter (%*x*. *p x* & *q x*) *u*)))
apply (*case-tac l* ∈ *Domain* (*findRel q*))
apply (*subgoal-tac* [*2*] *l* ~: *Domain* (*findRel* (%*x*. *p x* & *q x*)))
prefer *3* **apply** (*blast intro*: *rev-subsetD* [*OF* - *Domain-findRel-mono*])

There are no *qs* in *l*: both lists are *LNil*

apply (*simp-all add*: *Domain-findRel-iff*, *clarify*) case $q\ x$

apply (*case-tac p x*)
 apply (*simp-all add*: *findRel-conj* [*THEN findRel-imp-lfilter*])

case $q\ x$ and ~($p\ x$)

apply (*case-tac* $l' \in$ *Domain* (*findRel* (%x. $p\ x$ & $q\ x$)))

subcase: there is no p & q in l' and therefore none in l

 apply (*subgoal-tac* [*2*] l ~: *Domain* (*findRel* (%x. $p\ x$ & $q\ x$)))
  prefer *3* **apply** (*blast intro*: *findRel-not-conj-Domain*)
 apply (*subgoal-tac* [*2*] *lfilter* $q\ l'$ ~: *Domain* (*findRel* p))
  prefer *3* **apply** (*blast intro*: *findRel-lfilter-Domain-conj*)

... and therefore too, no p in *lfilter* $q\ l'$. Both results are *LNil*

 apply (*simp-all add*: *Domain-findRel-iff*, *clarify*)

subcase: there is a p & q in l' and therefore also one in l

apply (*subgoal-tac* (l, *LCons xa* $l'a$) $\in$ *findRel* (%x. $p\ x$ & $q\ x$))
 prefer *2* **apply** (*blast intro*: *findRel-conj2*)
apply (*subgoal-tac* (*lfilter* $q\ l'$, *LCons xa* (*lfilter* $q\ l'a$)) $\in$ *findRel* p)
 apply *simp*
apply (*blast intro*: *findRel-conj-lfilter*)
done

lemma *lfilter-conj*: *lfilter* p (*lfilter* $q\ l$) = *lfilter* (%x. $p\ x$ & $q\ x$) l
apply (*rule-tac* $l = l$ **in** *llist-fun-equalityI*, *simp-all*)
apply (*blast intro*: *lfilter-conj-lemma rev-subsetD* [*OF - llistD-Fun-mono*])
done

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

lemma *findRel-lmap-Domain*:
  (l,l') $\in$ *findRel*(%x. p ($f\ x$)) ==> *lmap* $f\ l \in$ *Domain*(*findRel* p)
by (*erule findRel.induct*, *auto*)

lemma *lmap-eq-LCons* [*rule-format*]: *lmap* $f\ l$ = *LCons* $x\ l'$ -->
      ($\exists\, y\ l''$. $x = f\ y$ & l' = *lmap* $f\ l''$ & l = *LCons* $y\ l''$)
apply (*subst lmap-def* [*THEN def-llist-corec*])
apply (*rule-tac* $l = l$ **in** *llistE*, *auto*)
done

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

$(l,\ LCons\ y\ l'') \in findRel(\%x.\ p(f\ x)))$
apply (*erule findRel.induct*, *simp-all*)
apply (*blast dest*!: *lmap-eq-LCons*)+
done

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

lemma *lfilter-lmap*: *lfilter p* (*lmap f l*) = *lmap f* (*lfilter* (*p o f*) *l*)
apply (*rule-tac l* = *l* **in** *llist-fun-equalityI*, *simp-all*)
apply *safe*
apply (*case-tac lmap f l* ∈ *Domain* (*findRel p*))
apply (*simp add*: *Domain-findRel-iff*, *clarify*)
apply (*frule lmap-LCons-findRel*, *force*)
apply (*subgoal-tac l* ~: *Domain* (*findRel* (%*x*. *p* (*f x*))), *simp*)
apply (*blast intro*: *findRel-lmap-Domain*)
done

end