Dependent structures

Daria Walukiewicz-Chrząszcz

9 april 2019

Destruction - match

$$\text{match } m \text{ as } x \text{ in } I \_ \ \vec{y} \text{ return } (P\ \vec{y}\ x) \text{ with}$$

$$(c_1\ x_{11}\ ...\ x_{1k_1}) \Rightarrow f_1 \mid \ldots \mid (c_n\ x_{n1} ... x_{nk_n}) \Rightarrow f_n \text{ end}$$

for $m : I\vec{a}\vec{b}$ the expression above has type $P\vec{b}m$, where

$$I : \forall (p_1 : A_1) \ldots (p_p : A_p)(z_1 : Z_1) \ldots (z_m : Z_m).s$$

$$P : \forall (z_1 : Z_1) \ldots (z_m : Z_m)(c : I\vec{a}\vec{z}).Type$$

If

$$c_i : \forall (p_1 : A_1) \ldots (p_p : A_p)(v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).I\vec{p}\vec{w}$$

then

$$f_i : \forall (v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).P\vec{w}(c_i\vec{a}\vec{v})$$

Destruction - match

match m as x in I _ $\vec{y}$ return $(P\ \vec{y}\ x)$ with

$(c_1\ x_{11}\ ...\ x_{1k_1}) \Rightarrow f_1 \mid\ \ldots\ \mid (c_n\ x_{n1}...x_{nk_n}) \Rightarrow f_n$ end

for $m : I\vec{a}\vec{b}$ the expression above has type $P\vec{b}m$, where

$$I : \forall (p_1 : A_1) \ldots (p_p : A_p)(z_1 : Z_1) \ldots (z_m : Z_m).s$$

$$P : \forall (z_1 : Z_1) \ldots (z_m : Z_m)(c : I\vec{a}\vec{z}).Type$$

If

$$c_i : \forall (p_1 : A_1) \ldots (p_p : A_p)(v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).I\vec{p}\vec{w}$$

then

$$f_i : \forall (v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).P\vec{w}(c_i\vec{a}\vec{v})$$

Destruction - match

match m as x in I _ $\vec{y}$ return $(P\ \vec{y}\ x)$ with

$(c_1\ x_{11}\ \dots\ x_{1k_1}) \Rightarrow f_1 \mid \dots \mid (c_n\ x_{n1} \dots x_{nk_n}) \Rightarrow f_n$ end

for $m : I\vec{a}\vec{b}$ the expression above has type $P\vec{b}m$, where

$$I : \forall (p_1 : A_1) \dots (p_p : A_p)(z_1 : Z_1) \dots (z_m : Z_m).s$$

$$P : \forall (z_1 : Z_1) \dots (z_m : Z_m)(c : I\vec{a}\vec{z}).Type$$

If

$$c_i : \forall (p_1 : A_1) \dots (p_p : A_p)(v_1 : V_1) \dots (v_{k_i} : V_{k_i}).I\vec{p}\vec{w}$$

then

$$f_i : \forall (v_1 : V_1) \dots (v_{k_i} : V_{k_i}).P\vec{w}(c_i\vec{a}\vec{v})$$

Destruction - match

$$\text{match } m \text{ as } x \text{ in } I \ \_ \ \vec{y} \text{ return } (P\ \vec{y}\ x) \text{ with}$$

$$(c_1\ x_{11}\ ...\ x_{1k_1}) \Rightarrow f_1 \mid \ldots \mid (c_n\ x_{n1}...x_{nk_n}) \Rightarrow f_n \text{ end}$$

for $m : I\vec{a}\vec{b}$ the expression above has type $P\vec{b}m$, where

$$I : \forall (p_1 : A_1) \ldots (p_p : A_p)(z_1 : Z_1) \ldots (z_m : Z_m).s$$

$$P : \forall (z_1 : Z_1) \ldots (z_m : Z_m)(c : I\vec{a}\vec{z}).Type$$

If

$$c_i : \forall (p_1 : A_1) \ldots (p_p : A_p)(v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).I\vec{p}\vec{w}$$

then

$$f_i : \forall (v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).P\vec{w}(c_i\vec{a}\vec{v})$$

Destruction - match

$$\text{match } m \text{ as } x \text{ in } I \ \_ \ \vec{y} \text{ return } (P \ \vec{y} \ x) \text{ with}$$

$$(c_1 \ x_{11} \ ... \ x_{1k_1}) \Rightarrow f_1 \mid \ \ldots \ \mid (c_n \ x_{n1}...x_{nk_n}) \Rightarrow f_n \text{ end}$$

for $m : I\vec{a}\vec{b}$ the expression above has type $P\vec{b}m$, where

$$I : \forall (p_1 : A_1) \ldots (p_p : A_p)(z_1 : Z_1) \ldots (z_m : Z_m).s$$

$$P : \forall (z_1 : Z_1) \ldots (z_m : Z_m)(c : I\vec{a}\vec{z}).Type$$

If

$$c_i : \forall (p_1 : A_1) \ldots (p_p : A_p)(v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).I\vec{p}\vec{w}$$

then

$$f_i : \forall (v_1 : V_1) \ldots (v_{k_i} : V_{k_i}).P\vec{w}(c_i\vec{a}\vec{v})$$

Lists with length: ilist

Section ilist.

  Variable A : Set.

  Inductive **ilist** : **nat** → Set :=
  | Nil : **ilist** O
  | Cons : ∀ n, A → **ilist** n → **ilist** (S n).

Function app on ilist

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2 ≡ f1
      | Cons x ls1' ⇒ Cons x (app' ls1' ls2) ≡ f2 _ x ls'
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Since $Nil\ :\ ilist\ 0$ one has

$$f1\ :\ P\ 0\ Nil$$

Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ ilist(0 + n2)$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(S\ n' + n2)$$

Function app on ilist

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2 ≡ f1
      | Cons x ls1' ⇒ Cons x (app' ls1' ls2) ≡ f2 _ x ls'
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Since $Nil\ :\ ilist\ 0$ one has

$$f1\ :\ P\ 0\ Nil$$

Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ ilist(0 + n2)$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(S\ n' + n2)$$

Function app on ilist

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2 ≡ f1
      | Cons x ls1' ⇒ Cons x (app' ls1' ls2) ≡ f2 _ x ls'
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Since $Nil\ :\ ilist\ 0$ one has

$$f1\ :\ P\ 0\ Nil$$

Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ ilist(0 + n2)$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(S\ n' + n2)$$

Function app on ilist

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2 ≡ f1
      | Cons x ls1' ⇒ Cons x (app' ls1' ls2) ≡ f2 _ x ls'
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Since $Nil\ :\ ilist\ 0$ one has

$$f1\ :\ P\ 0\ Nil$$

Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ ilist(0 + n2)$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(S\ n' + n2)$$

Function app on ilist

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2 ≡ f1
      | Cons x ls1' ⇒ Cons x (app' ls1' ls2) ≡ f2 _ x ls'
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Since $Nil\ :\ ilist\ 0$ one has

$$f1\ :\ P\ 0\ Nil$$

Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ ilist(0 + n2)$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(S\ n' + n2)$$

Elimination of equality

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l ≡ f1
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist\ m$$

Since `eq_refl` $: \forall\, (A : Set)(a : A).$eq $A\ a\ a$ one has

$$f1\ :\ P\ n\ (\texttt{eq\_refl}\ nat\ n)$$

therefore

$$f1\ :\ ilist\ n$$

- That is an elimination from `Prop` to `Set` for a singleton type
- That is how tactic `rewrite` works

Elimination of equality

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l ≡ f1
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist\ m$$

Since `eq_refl` : $\forall\ (A : Set)(a : A)$.eq $A\ a\ a$ one has

$$f1\ :\ P\ n\ (\texttt{eq\_refl}\ nat\ n)$$

therefore

$$f1\ :\ ilist\ n$$

- That is an elimination from Prop to Set for a singleton type
- That is how tactic `rewrite` works

Elimination of equality

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l ≡ f1
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist\ m$$

Since eq_refl : $\forall\ (A : Set)(a : A)$.eq $A\ a\ a$ one has

$$f1\ :\ P\ n\ (\texttt{eq\_refl}\ nat\ n)$$

therefore

$$f1\ :\ ilist\ n$$

- That is an elimination from Prop to Set for a singleton type
- That is how tactic rewrite works

Elimination of equality

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist  n): ilist  m :=
match h in _=m with return (ilist  m)
eq_refl => l ≡ f1
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist\ m$$

Since `eq_refl` : $\forall\ (A : Set)(a : A).\texttt{eq}\ A\ a\ a$ one has

$$f1\ :\ P\ n\ (\texttt{eq\_refl}\ nat\ n)$$

therefore

$$f1\ :\ ilist\ n$$

- That is an elimination from Prop to Set for a singleton type
- That is how tactic rewrite works

Elimination of equality

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist  n): ilist  m :=
match h in _=m with return (ilist  m)
eq_refl => l ≡ f1
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist\ m$$

Since `eq_refl` : $\forall\ (A : Set)(a : A)$.eq $A\ a\ a$ one has

$$f1\ :\ P\ n\ (\texttt{eq\_refl}\ nat\ n)$$

therefore

$$f1\ :\ ilist\ n$$

- That is an elimination from `Prop` to `Set` for a singleton type
- That is how tactic `rewrite` works

Elimination of equality

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l ≡ f1
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist\ m$$

Since `eq_refl` : $\forall\ (A : Set)(a : A)$.eq $A\ a\ a$ one has

$$f1\ :\ P\ n\ (\texttt{eq\_refl}\ nat\ n)$$

therefore

$$f1\ :\ ilist\ n$$

- That is an elimination from `Prop` to `Set` for a singleton type
- That is how tactic `rewrite` works

Function hd1 on ilist

```
Definition hd1 n (ls : ilist (S n)) : A :=
match
  ls as ls0 in (ilist n0)
  return
      (match n0 with
      | 0 => unit
      | S n1 => A
      end)
with

| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end

P= fun  (n0:nat)(ls0:ilist  n0) =>
                  (match n0 with
                        | 0 => unit
                        | S n1 => A
                  end)
```

Function hd1 on ilist

```
Definition hd1 n (ls : ilist (S n)) : A :=
match
  ls as ls0 in (ilist n0)
  return
      (match n0 with
      | 0 => unit
      | S n1 => A
      end)
with

| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end

P= fun  (n0:nat)(ls0:ilist  n0) =>
                  (match n0 with
                        | 0 => unit
                        | S n1 => A
                  end)
```

Function hd1 on ilist cont.

```
P= fun  (n0:nat)(ls0:ilist  n0) =>
                   (match n0 with
                         | 0 => unit
                         | S n1 => A
                   end)
```

Since $Nil : ilist\ 0$ one has

$$f1 : P\ 0\ Nil$$

Since $Cons : \forall (n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2 : \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (S\ n')\ (Cons\ n'\ a\ l)$$

therefore

$$f1 : unit$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

Function hd1 on ilist cont.

```
P= fun  (n0:nat)(ls0:ilist  n0) =>
                  (match n0 with
                        | 0 => unit
                        | S n1 => A
                  end)
```

Since $Nil : ilist\ 0$ one has

$$f1 : P\ 0\ Nil$$

Since $Cons : \forall (n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2 : \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (S\ n')\ (Cons\ n'\ a\ l)$$

therefore

$$f1 : unit$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

Function hd1 on ilist cont.

```
P= fun  (n0:nat)(ls0:ilist  n0) =>
                   (match n0 with
                         | 0 => unit
                         | S n1 => A
                   end)
```

Since $Nil : ilist\ 0$ one has

$$f1 : P\ 0\ Nil$$

Since $Cons : \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2 : \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (S\ n')\ (Cons\ n'\ a\ l)$$

therefore

$$f1 : unit$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

Function hd1 on ilist cont.

```
P= fun  (n0:nat)(ls0:ilist  n0) =>
                (match n0 with
                     | 0 => unit
                     | S n1 => A
                end)
```

Since $Nil : ilist\, 0$ one has

$$f1 : P\; 0\; Nil$$

Since $Cons : \forall (n' : nat)(a : A)(l : ilist\; n'), ilist\; (S\; n')$ one has

$$f2 : \forall (n' : nat)(a : A)(l : ilist\; n'), P\; (S\; n')\; (Cons\; n'\; a\; l)$$

therefore

$$f1 : unit$$

$$f2 : \forall\; (n' : nat)(a : A)(l : ilist\; n'), A$$

Function hd1 on ilist cont.

```
P= fun  (n0:nat)(ls0:ilist  n0) =>
                 (match n0 with
                       | 0 => unit
                       | S n1 => A
                 end)
```

Since $Nil : ilist\ 0$ one has

$$f1 : P\ 0\ Nil$$

Since $Cons : \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2 : \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (S\ n')\ (Cons\ n'\ a\ l)$$

therefore

$$f1 : unit$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

Function hd2 on ilist

```
Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
       return (match n with O => unit | S  _=> A end) with
| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end.
```

$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow (match\ n\ with\ 0\ \Rightarrow\ unit \mid S\ \_\ \Rightarrow\ A\ end)$

Since $Nil\ :\ ilist\ 0$ one has $f1\ :\ P\ 0\ Nil$

Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ unit$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

```
Definition hd2 n (ls : ilist (S n)) := hd_pom (S n) ls.
```

Function hd2 on ilist

```
Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
       return (match n with O => unit | S  _=> A end) with
| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end.
```

$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow (match\ n\ with\ 0\ \Rightarrow\ unit \mid S\ \_\ \Rightarrow\ A\ end)$

Since $Nil\ :\ ilist\ 0$ one has $f1\ :\ P\ 0\ Nil$
Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ unit$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

```
Definition hd2 n (ls : ilist (S n)) := hd_pom (S n) ls.
```

Function hd2 on ilist

```
Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
       return (match n with O => unit | S  _=> A end) with
| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end.
```

$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow (match\ n\ with\ 0\ \Rightarrow\ unit \mid S\ \_\ \Rightarrow\ A\ end)$

Since $Nil\ :\ ilist\ 0$ one has $f1\ :\ P\ 0\ Nil$

Since $Cons\ :\ \forall (n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ unit$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

```
Definition hd2 n (ls : ilist (S n)) := hd_pom (S n) ls.
```

Function hd2 on ilist

```
Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
       return (match n with O => unit | S  _=> A end) with
| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end.
```

$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow (match\ n\ with\ 0\ \Rightarrow\ unit \mid S\ \_\ \Rightarrow\ A\ end)$

Since $Nil\ :\ ilist\ 0$ one has $f1\ :\ P\ 0\ Nil$
Since $Cons\ :\ \forall (n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ unit$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

```
Definition hd2 n (ls : ilist (S n)) := hd_pom (S n) ls.
```

Function hd2 on ilist

```
Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
       return (match n with O => unit | S _=> A end) with
| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow (match\ n\ with\ 0\ \Rightarrow\ unit \mid S\ \_\ \Rightarrow\ A\ end)$$

Since $Nil\ :\ ilist\ 0$ one has $f1\ :\ P\ 0\ Nil$
Since $Cons\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall(n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ unit$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

```
Definition hd2 n (ls : ilist (S n)) := hd_pom (S n) ls.
```

Function hd2 on ilist

```
Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
       return (match n with 0 => unit | S _=> A end) with
| Nil => tt ≡ f1
| Cons h _=> h ≡ f2 _ h _
end.
```

$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow (match\ n\ with\ 0\ \Rightarrow\ unit \mid S\ \_\ \Rightarrow\ A\ end)$

Since $Nil\ :\ ilist\ 0$ one has $f1\ :\ P\ 0\ Nil$
Since $Cons\ :\ \forall (n' : nat)(a : A)(l : ilist\ n'), ilist\ (S\ n')$ one has

$$f2\ :\ \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$$

therefore

$$f1\ :\ unit$$

$$f2\ :\ \forall\ (n' : nat)(a : A)(l : ilist\ n'), A$$

```
Definition hd2 n (ls : ilist (S n)) := hd_pom (S n) ls.
```

Initial subsets of natural numbers

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).

  Inductive fin : nat → Set :=
  | First : ∀ n, fin (S n)
  | Next : ∀ n, fin n → fin (S n).
```

Values of type **fin** 3 are: First 2, Next (First 1), i Next (Next (First 0)).

Note: there are no terms of type **fin** 0 !

Initial subsets of natural numbers

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).

  Inductive fin : nat → Set :=
  | First : ∀ n, fin (S n)
  | Next : ∀ n, fin n → fin (S n).
```

Values of type **fin** 3 are: First 2, `Next` (First 1), i `Next` (`Next` (First 0)).

Note: there are no terms of type **fin** 0 !

Initial subsets of natural numbers

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).

  Inductive fin : nat → Set :=
  | First : ∀ n, fin (S n)
  | Next : ∀ n, fin n → fin (S n).
```

Values of type **fin** 3 are: First 2, Next (First 1), i Next (Next (First 0)).

Note: there are no terms of type **fin** 0 !

Initial subsets of natural numbers

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).

  Inductive fin : nat → Set :=
  | First : ∀ n, fin (S n)
  | Next : ∀ n, fin n → fin (S n).
```

Values of type **fin** 3 are: First 2, `Next` (First 1), i `Next` (`Next` (First 0)).

Note: there are no terms of type **fin** 0 !

Function get (1)

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒ ?
    | Cons x ls' ⇒ fun idx ⇒
      match idx with
        | First _ ⇒ x
        | Next idx' ⇒ get ls' idx'
      end
  end.
```

Function get (2)

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls in ilist k return fin k → A with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ ⇒ tt
      end
    | Cons x ls' ⇒ fun idx ⇒
      match idx in fin n' return A with
        | First _ ⇒ fun _ ⇒ x
        | Next idx' ⇒ fun ls' ⇒ get ls' idx'
      end ls'
  end.
```

The third return needed to connect the type of idx' and the type of idx. There is a problem with recursive call to get

Function get (2)

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls in ilist k return fin k → A with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ ⇒ tt
      end
    | Cons x ls' ⇒ fun idx ⇒
      match idx in fin n' return A with
        | First _ ⇒ fun _ ⇒ x
        | Next idx' ⇒ fun ls' ⇒ get ls' idx'
      end ls'
  end.
```

The third `return` needed to connect the type of `idx'` and the type of `idx`. There is a problem with recursive call to get

Function get (3)

```
Fixpoint get n (ls : ilist n) : fin n → A :=
  match ls with
    | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
                                    | O ⇒ A
                                    | S _ ⇒ unit
                                  end) with
        | First _ ⇒ tt
        | Next _ ⇒ tt
      end
    | Cons x ls' ⇒ fun idx ⇒
      match idx in fin n' return (fin (pred n') → A) → A with
        | First _ ⇒ fun _ ⇒ x
        | Next idx' ⇒ fun get_ls' ⇒ get_ls' idx'
      end (get ls')
  end.
End ilist.
```

Examples

```
Arguments Nil [A]. Arguments Cons [A n].
Arguments First [n]. Arguments Next [n].

Check Cons 0 (Cons 1 (Cons 2 Nil)).

  Cons 0 (Cons 1 (Cons 2 Nil))
     : ilist nat 3

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.

     = 0
     : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).

     = 1
     : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).

     = 2
     : nat
```

Examples

```
Arguments Nil [A]. Arguments Cons [A n].
Arguments First [n]. Arguments Next [n].
```

```
Check Cons 0 (Cons 1 (Cons 2 Nil)).
```

Cons 0 (Cons 1 (Cons 2 Nil))
: **ilist nat** 3

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.
```

= 0
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).
```

= 1
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).
```

= 2
: **nat**

Examples

```
Arguments Nil [A]. Arguments Cons [A n].
Arguments First [n]. Arguments Next [n].
Check Cons 0 (Cons 1 (Cons 2 Nil)).
```

Cons 0 (Cons 1 (Cons 2 Nil))
: **ilist nat** 3

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.
```

= 0
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).
```

= 1
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).
```

= 2
: **nat**

Examples

```
Arguments Nil [A]. Arguments Cons [A n].
Arguments First [n]. Arguments Next [n].
Check Cons 0 (Cons 1 (Cons 2 Nil)).

  Cons 0 (Cons 1 (Cons 2 Nil))
      : ilist nat 3

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.

      = 0
      : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).

      = 1
      : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).

      = 2
      : nat
```

Examples

```
Arguments Nil [A]. Arguments Cons [A n].
Arguments First [n]. Arguments Next [n].
Check Cons 0 (Cons 1 (Cons 2 Nil)).
```

Cons 0 (Cons 1 (Cons 2 Nil))
: **ilist nat** 3

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.
```

= 0
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).
```

= 1
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).
```

= 2
: **nat**

Examples

```
Arguments Nil [A]. Arguments Cons [A n].
Arguments First [n]. Arguments Next [n].
Check Cons 0 (Cons 1 (Cons 2 Nil)).
```

Cons 0 (Cons 1 (Cons 2 Nil))
: **ilist nat** 3

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.
```

= 0
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).
```

= 1
: **nat**

```
Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).
```

= 2
: **nat**

Examples

```
Arguments Nil [A]. Arguments Cons [A n].
Arguments First [n]. Arguments Next [n].
Check Cons 0 (Cons 1 (Cons 2 Nil)).

  Cons 0 (Cons 1 (Cons 2 Nil))
      : ilist nat 3

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) First.

      = 0
      : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).

      = 1
      : nat

Eval simpl in get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).

      = 2
      : nat
```

Examples

`Arguments` Nil [A]. `Arguments` Cons [A n].
`Arguments` First [n]. `Arguments` Next [n].

`Check` Cons 0 (Cons 1 (Cons 2 Nil)).

Cons 0 (Cons 1 (Cons 2 Nil))
: **ilist nat** 3

`Eval simpl in` get (Cons 0 (Cons 1 (Cons 2 Nil))) First.

= 0
: **nat**

`Eval simpl in` get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next First).

= 1
: **nat**

`Eval simpl in` get (Cons 0 (Cons 1 (Cons 2 Nil))) (Next (Next First)).

= 2
: **nat**

Heterogenic lists

```
Section hlist.
  Variable A : Type.
  Variable B : A → Type.

  Inductive hlist : list A → Type :=
  | HNil : hlist nil
  | HCons : ∀ (x : A) (ls : list A), B x → hlist ls → hlist (x :: ls).

  Variable elm : A.

  Inductive member : list A → Type :=
  | HFirst : ∀ ls, member (elm :: ls)
  | HNext : ∀ x ls, member ls → member (x :: ls).
```

Heterogenic lists

```
Section hlist.
  Variable A : Type.
  Variable B : A → Type.

  Inductive hlist : list A → Type :=
  | HNil : hlist nil
  | HCons : ∀ (x : A) (ls : list A), B x → hlist ls → hlist (x :: ls).

  Variable elm : A.

  Inductive member : list A → Type :=
  | HFirst : ∀ ls, member (elm :: ls)
  | HNext : ∀ x ls, member ls → member (x :: ls).
```

Heterogenic lists

```
Section hlist.
  Variable A : Type.
  Variable B : A → Type.

  Inductive hlist : list A → Type :=
  | HNil : hlist nil
  | HCons : ∀ (x : A) (ls : list A), B x → hlist ls → hlist (x :: ls).

  Variable elm : A.

  Inductive member : list A → Type :=
  | HFirst : ∀ ls, member (elm :: ls)
  | HNext : ∀ x ls, member ls → member (x :: ls).
```

Function hget(1)

```
Fixpoint hget ls (mls : hlist ls) : member ls → B elm :=
  match mls with
    | HNil ⇒ fun mem ⇒
      match mem in member ls' return (match ls' with
                                        | nil ⇒ B elm
                                        | _ :: _ ⇒ unit
                                      end) with
        | HFirst _ ⇒ tt
        | HNext _ _ ⇒ tt
      end
```

Function hget(2)

```
| HCons e mls' ⇒ fun mem ⇒
  match mem in member ls' return (match ls' with
                                   | nil ⇒ Empty_set
                                   | x' :: ls'' ⇒
                B x' → (member ls'' → B elm) → B elm
                                 end) with
     | HFirst _ ⇒ fun e' _ ⇒ e'
     | HNext _ mem' ⇒ fun _ get_mls' ⇒ get_mls' mem'
  end e (hget mls')
end.
End hlist.
```

Examples of heterogenic lists

```
Arguments HCons [A B x ls].
Arguments HNil [A B].

Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set => T) someTypes :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

    = 5
    : (fun T : Set => T) nat

Eval simpl in hget someValues (HNext HFirst).

    = true
    : (fun T : Set => T) bool
```

Examples of heterogenic lists

```
Arguments HCons [A B x ls].
Arguments HNil [A B].

Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set => T) someTypes :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

    = 5
    : (fun T : Set => T) nat

Eval simpl in hget someValues (HNext HFirst).

    = true
    : (fun T : Set => T) bool
```

Examples of heterogenic lists

```
Arguments HCons [A B x ls].
Arguments HNil [A B].

Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set => T) someTypes :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

    = 5
    : (fun T : Set => T) nat

Eval simpl in hget someValues (HNext HFirst).

    = true
    : (fun T : Set => T) bool
```

Examples of heterogenic lists

```
Arguments HCons [A B x ls].
Arguments HNil [A B].

Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set => T) someTypes :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

     = 5
     : (fun T : Set => T) nat

Eval simpl in hget someValues (HNext HFirst).

     = true
     : (fun T : Set => T) bool
```

Examples of heterogenic lists

```
Arguments HCons [A B x ls].
Arguments HNil [A B].

Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set => T) someTypes :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

     = 5
     : (fun T : Set => T) nat

Eval simpl in hget someValues (HNext HFirst).

     = true
     : (fun T : Set => T) bool
```

Examples of heterogenic lists

```
Arguments HCons [A B x ls].
Arguments HNil [A B].

Definition someTypes : list Set := nat :: bool :: nil.

Example someValues : hlist (fun T : Set => T) someTypes :=
  HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

    = 5
    : (fun T : Set => T) nat

Eval simpl in hget someValues (HNext HFirst).

    = true
    : (fun T : Set => T) bool
```

Interpreter of simply typed lambda calculus (1)

Inductive **type** : Set :=
| Unit : **type**
| Arrow : **type** → **type** → **type**.

Inductive **exp** : **list type** → **type** → Set :=
| Const : ∀ *ts*, **exp** *ts* Unit
| Var : ∀ *ts t*, **member** *t ts* → **exp** *ts t*
| App : ∀ *ts dom ran*, **exp** *ts* (Arrow *dom ran*) → **exp** *ts dom* → **exp** *ts ran*
| Abs : ∀ *ts dom ran*, **exp** (*dom* :: *ts*) *ran* → **exp** *ts* (Arrow *dom ran*).

Arguments Const [*ts*].

Interpreter of simply typed lambda calculus (1)

```
Inductive type : Set :=
| Unit : type
| Arrow : type → type → type.
```

```
Inductive exp : list type → type → Set :=
| Const : ∀ ts, exp ts Unit
| Var : ∀ ts t, member t ts → exp ts t
| App : ∀ ts dom ran, exp ts (Arrow dom ran) → exp ts dom → exp ts
ran
| Abs : ∀ ts dom ran, exp (dom :: ts) ran → exp ts (Arrow dom ran).
```

```
Arguments Const [ts].
```

Interpreter of simply typed lambda calculus (1)

Inductive **type** : Set :=
| Unit : **type**
| Arrow : **type** → **type** → **type**.

Inductive **exp** : **list type** → **type** → Set :=
| Const : ∀ *ts*, **exp** *ts* Unit
| Var : ∀ *ts t*, **member** *t ts* → **exp** *ts t*
| App : ∀ *ts dom ran*, **exp** *ts* (Arrow *dom ran*) → **exp** *ts dom* → **exp** *ts* *ran*
| Abs : ∀ *ts dom ran*, **exp** (*dom* :: *ts*) *ran* → **exp** *ts* (Arrow *dom ran*).

Arguments Const [*ts*].

Interpreter of simply typed lambda calculus (2)

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Unit ⇒ unit
    | Arrow t1 t2 ⇒ typeDenote t1 → typeDenote t2
  end.
```

```
Fixpoint expDenote ts t (e : exp ts t) : hlist typeDenote ts →
typeDenote t :=
  match e with
    | Const _ ⇒ fun _ ⇒ tt
    | Var mem ⇒ fun s ⇒ hget s mem
    | App e1 e2 ⇒ fun s ⇒ (expDenote e1 s) (expDenote e2 s)
    | Abs e' ⇒ fun s ⇒ fun x ⇒ expDenote e' (HCons x s)
  end.
```

Interpreter of simply typed lambda calculus (2)

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | Unit ⇒ unit
    | Arrow t1 t2 ⇒ typeDenote t1 → typeDenote t2
  end.
```

```
Fixpoint expDenote ts t (e : exp ts t) : hlist typeDenote ts →
typeDenote t :=
  match e with
    | Const _ ⇒ fun _ ⇒ tt
    | Var mem ⇒ fun s ⇒ hget s mem
    | App e1 e2 ⇒ fun s ⇒ (expDenote e1 s) (expDenote e2 s)
    | Abs e' ⇒ fun s ⇒ fun x ⇒ expDenote e' (HCons x s)
  end.
```

Interpreter of simply typed lambda calculus - examples

```
Eval simpl in expDenote Const HNil.
```

= tt : typeDenote Unit

```
Eval simpl in expDenote (Abs (dom := Unit) (Var HFirst)) HNil.
```

= fun *x* : **unit** ⇒ *x*
: typeDenote (Arrow Unit Unit)

```
Eval simpl in expDenote (Abs (dom := Unit)
  (Abs (dom := Unit) (Var (HNext HFirst)))) HNil.
```

= fun *x* _ : **unit** ⇒ *x*
: typeDenote (Arrow Unit (Arrow Unit Unit))

```
Eval simpl in expDenote (Abs (dom := Unit) (Abs (dom := Unit) (Var HFirst))) HNil.
```

= fun _ *x0* : **unit** ⇒ *x0*
: typeDenote (Arrow Unit (Arrow Unit Unit))

```
Eval simpl in expDenote (App (Abs (Var HFirst)) Const) HNil.
```

= tt : typeDenote Unit

Interpreter of simply typed lambda calculus - examples

`Eval simpl in` expDenote Const HNil.

= tt : typeDenote Unit

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Var HFirst)) HNil.

= `fun` *x* : **unit** ⇒ *x*
: typeDenote (Arrow Unit Unit)

`Eval simpl in` expDenote (Abs (*dom* := Unit)
(Abs (*dom* := Unit) (Var (HNext HFirst)))) HNil.

= `fun` *x* _ : **unit** ⇒ *x*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Abs (*dom* := Unit) (Var HFirst))) HNil.

= `fun` _ *x0* : **unit** ⇒ *x0*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (App (Abs (Var HFirst)) Const) HNil.

= tt : typeDenote Unit

Interpreter of simply typed lambda calculus - examples

`Eval simpl in` expDenote Const HNil.

= tt : typeDenote Unit

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Var HFirst)) HNil.

= `fun` *x* : **unit** ⇒ *x*
: typeDenote (Arrow Unit Unit)

`Eval simpl in` expDenote (Abs (*dom* := Unit)
(Abs (*dom* := Unit) (Var (HNext HFirst)))) HNil.

= `fun` *x* _ : **unit** ⇒ *x*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Abs (*dom* := Unit) (Var HFirst))) HNil.

= `fun` _ *x0* : **unit** ⇒ *x0*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (App (Abs (Var HFirst)) Const) HNil.

= tt : typeDenote Unit

Interpreter of simply typed lambda calculus - examples

`Eval simpl in` expDenote Const HNil.

= tt : typeDenote Unit

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Var HFirst)) HNil.

= `fun` *x* : **unit** ⇒ *x*
: typeDenote (Arrow Unit Unit)

`Eval simpl in` expDenote (Abs (*dom* := Unit)
(Abs (*dom* := Unit) (Var (HNext HFirst)))) HNil.

= `fun` *x* _ : **unit** ⇒ *x*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Abs (*dom* := Unit) (Var HFirst))) HNil.

= `fun` _ *x0* : **unit** ⇒ *x0*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (App (Abs (Var HFirst)) Const) HNil.

= tt : typeDenote Unit

Interpreter of simply typed lambda calculus - examples

`Eval simpl in` expDenote Const HNil.

= tt : typeDenote Unit

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Var HFirst)) HNil.

= `fun` *x* : **unit** ⇒ *x*
: typeDenote (Arrow Unit Unit)

`Eval simpl in` expDenote (Abs (*dom* := Unit)
(Abs (*dom* := Unit) (Var (HNext HFirst)))) HNil.

= `fun` *x* _ : **unit** ⇒ *x*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (Abs (*dom* := Unit) (Abs (*dom* := Unit) (Var HFirst))) HNil.

= `fun` _ *x0* : **unit** ⇒ *x0*
: typeDenote (Arrow Unit (Arrow Unit Unit))

`Eval simpl in` expDenote (App (Abs (Var HFirst)) Const) HNil.

= tt : typeDenote Unit

Interpreter of $\lambda^{\rightarrow}$— summary

- syntax, typing rules and semantics of evaluation for $\lambda^{\rightarrow}$
- interpreter = implementation of denotational semantics
- metatheorethical properties of $\lambda^{\rightarrow}$ follow from the properties of CIC (subject reduction, strong normalization)

Interpreter of $\lambda^{\rightarrow}$— summary

- syntax, typing rules and semantics of evaluation for $\lambda^{\rightarrow}$
- interpreter = implementation of denotational semantics
- metatheorethical properties of $\lambda^{\rightarrow}$ follow from the properties of CIC (subject reduction, strong normalization)

Interpreter of $\lambda^{\rightarrow}$— summary

- syntax, typing rules and semantics of evaluation for $\lambda^{\rightarrow}$
- interpreter = implementation of denotational semantics
- metatheorethical properties of $\lambda^{\rightarrow}$ follow from the properties of CIC (subject reduction, strong normalization)