

Dependent structures

Daria Walukiewicz-Chrząszcz

18 April 2023

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

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

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

If

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

then

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

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

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

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

if

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

then

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

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

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

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

if

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

then

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

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

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

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

If

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

then

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

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

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

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

If

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

then

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

Lists with length: `ilist`

Section `ilist`.

Variable $A : \text{Set}$.

Inductive **`ilist`** : $\text{nat} \rightarrow \text{Set} :=$

| `Nil` : **`ilist`** `O`

| `Cons` : $\forall n, A \rightarrow \text{ilist } n \rightarrow \text{ilist } (S \ n)$.

Function app on ilist

Fixpoint app' $n1$ ($ls1 : \mathbf{ilist}\ n1$) $n2$ ($ls2 : \mathbf{ilist}\ n2$) : $\mathbf{ilist}\ (n1 + n2)$
 :=
 match $ls1$ in ($\mathbf{ilist}\ n1$) return ($\mathbf{ilist}\ (n1 + n2)$) with
 | Nil $\Rightarrow ls2 \equiv f1$
 | Cons $x\ ls1' \Rightarrow$ Cons x (app' $ls1'\ ls2$) $\equiv f2\ _ x\ ls'$
 end.

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

Since $Nil : \text{ilist } 0$, one has

$$f1 : P\ 0\ Nil$$

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

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

therefore

$$f1 : \text{ilist}(0 + n2)$$

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

Function app on ilist

Fixpoint app' $n1$ ($ls1 : \mathbf{ilist}\ n1$) $n2$ ($ls2 : \mathbf{ilist}\ n2$) : $\mathbf{ilist}\ (n1 + n2)$
 $:=$
 match $ls1$ in ($\mathbf{ilist}\ n1$) return ($\mathbf{ilist}\ (n1 + n2)$) with
 | Nil $\Rightarrow ls2 \equiv f1$
 | Cons $x\ ls1' \Rightarrow$ Cons x (app' $ls1'\ ls2$) $\equiv f2_x\ ls'$
 end.

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

Since $Nil : \text{ilist } 0$, one has

$$f1 : P\ 0\ Nil$$

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

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

therefore

$$f1 : \text{ilist}(0 + n2)$$

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

Function app on ilist

Fixpoint app' $n1$ ($ls1 : \mathbf{ilist}\ n1$) $n2$ ($ls2 : \mathbf{ilist}\ n2$) : $\mathbf{ilist}\ (n1 + n2)$
 $:=$
 match $ls1$ in ($\mathbf{ilist}\ n1$) return ($\mathbf{ilist}\ (n1 + n2)$) with
 | $\text{Nil} \Rightarrow ls2 \equiv f1$
 | $\text{Cons } x\ ls1' \Rightarrow \text{Cons } x\ (\text{app}'\ ls1'\ ls2) \equiv f2 _ x\ ls'$
 end.

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

Since $\text{Nil} : \text{ilist } 0$, one has

$$f1 : P\ 0\ \text{Nil}$$

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

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

therefore

$$f1 : \text{ilist}(0 + n2)$$

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

Function app on ilist

Fixpoint app' $n1$ ($ls1 : \mathbf{ilist}\ n1$) $n2$ ($ls2 : \mathbf{ilist}\ n2$) : $\mathbf{ilist}\ (n1 + n2)$
 $:=$
 match $ls1$ in ($\mathbf{ilist}\ n1$) return ($\mathbf{ilist}\ (n1 + n2)$) with
 | Nil $\Rightarrow ls2 \equiv f1$
 | Cons $x\ ls1' \Rightarrow$ Cons x (app' $ls1'\ ls2$) $\equiv f2_x\ ls'$
 end.

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

Since $Nil : \text{ilist } 0$, one has

$$f1 : P\ 0\ Nil$$

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

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

therefore

$$f1 : \text{ilist}(0 + n2)$$

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

Function app on ilist

Fixpoint $\text{app}' \ n1 \ (ls1 : \mathbf{ilist} \ n1) \ n2 \ (ls2 : \mathbf{ilist} \ n2) : \mathbf{ilist} \ (n1 + n2)$
 $:=$
 match $ls1$ in $(\mathbf{ilist} \ n1)$ return $(\mathbf{ilist} \ (n1 + n2))$ with
 | $\text{Nil} \Rightarrow ls2 \equiv f1$
 | $\text{Cons } x \ ls1' \Rightarrow \text{Cons } x \ (\text{app}' \ ls1' \ ls2) \equiv f2 _ x \ ls'$
 end.

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

Since $\text{Nil} : \text{ilist } 0$, one has

$$f1 : P \ 0 \ \text{Nil}$$

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

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

therefore

$$f1 : \text{ilist}(0 + n2)$$

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

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
```

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

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

$$f1 : P n (\text{eq_refl } \text{nat } n)$$

therefore

$$f1 : \text{ilist } n$$

- 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
```

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

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

$$f1 : P n (\text{eq_refl } nat n)$$

therefore

$$f1 : \text{ilist } n$$

- 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
  
```

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

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

$$f1 : P n (\text{eq_refl } \text{nat } n)$$

therefore

$$f1 : \text{ilist } n$$

- 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

```

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

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

$$f1 : P n (\text{eq_refl } \text{nat } n)$$

therefore

$$f1 : \text{ilist } n$$

- 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

```

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

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

$$f1 : P n (\text{eq_refl } \text{nat } n)$$

therefore

$$f1 : \text{ilist } n$$

- 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

```

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

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

$$f1 : P n (\text{eq_refl } \text{nat } n)$$

therefore

$$f1 : \text{ilist } n$$

- 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 :unit
```

```
| Cons h _ => h ≡ f2 _ h _ :A
```

```
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 :unit
```

```
| Cons h _ => h ≡ f2 _ h _ :A
```

```
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 0 => unit | S _=> A end) with
  | Nil => tt ≡ f1
  | Cons h _=> h ≡ f2 _ h _
end.

```

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

Since $\text{Nil} : \text{ilist } 0$, one has $f1 : P \ 0 \ \text{Nil}$

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

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

therefore

$$f1 : \text{unit}$$

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

```

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.

```

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

Since $\text{Nil} : \text{ilist } 0$, one has $f1 : P \ 0 \ \text{Nil}$

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

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

therefore

$$f1 : \text{unit}$$

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

```

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.

```

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

Since $\text{Nil} : \text{ilist } 0$, one has $f1 : P \ 0 \ \text{Nil}$

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

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

therefore

$$f1 : \text{unit}$$

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

```

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.

```

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

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

$$f1 : \text{unit}$$

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

```

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.

```

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

Since $\text{Nil} : \text{ilist } 0$, one has $f1 : P \ 0 \ \text{Nil}$

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

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

therefore

$$f1 : \text{unit}$$

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

```

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.

```

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

Since $\text{Nil} : \text{ilist } 0$, one has $f1 : P \ 0 \ \text{Nil}$

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

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

therefore

$$f1 : \text{unit}$$

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

```

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` \rightarrow `Set` :=

| `Nil` : `ilist` `0`

| `Cons` : $\forall n, A \rightarrow \text{ilist } n \rightarrow \text{ilist } (S n)$.

Inductive `fin` : `nat` \rightarrow `Set` :=

| `First` : $\forall n, \text{fin } (S n)$

| `Next` : $\forall n, \text{fin } n \rightarrow \text{fin } (S n)$.

Values of type `fin 3` are: `First 2`, `Next (First 1)` and `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` \rightarrow `Set` :=

| `Nil` : `ilist` `O`

| `Cons` : $\forall n, A \rightarrow$ `ilist` `n` \rightarrow `ilist` (`S n`).

Inductive `fin` : `nat` \rightarrow `Set` :=

| `First` : $\forall n,$ `fin` (`S n`)

| `Next` : $\forall n,$ `fin` `n` \rightarrow `fin` (`S n`).

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

| `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)` and `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 0`

| `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)` and `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 _ ⇒ x
      | Next idx' ⇒ get ls' idx'
      end
  end.

```

There is a problem with type-checking the 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 _ ⇒ x
    | Next idx' ⇒ get ls' idx'
    end
  end.

```

There is a problem with type-checking the 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 $A : \text{Type}$.

Variable $B : A \rightarrow \text{Type}$.

Inductive `hlist` : `list A` \rightarrow `Type` :=

| `HNil` : `hlist nil`

| `HCons` : $\forall (x : A) (ls : \text{list } A), B\ x \rightarrow \text{hlist } ls \rightarrow \text{hlist } (x :: ls)$.

Variable $elm : A$.

Inductive `member` : `list A` \rightarrow `Type` :=

| `HFirst` : $\forall ls, \text{member } (elm :: ls)$

| `HNext` : $\forall x\ ls, \text{member } ls \rightarrow \text{member } (x :: ls)$.

Heterogenic lists

Section `hlist`.

Variable $A : \text{Type}$.

Variable $B : A \rightarrow \text{Type}$.

Inductive **`hlist`** : **`list A`** \rightarrow `Type` :=

| `HNil` : **`hlist`** `nil`

| `HCons` : $\forall (x : A) (ls : \text{list } A), B\ x \rightarrow \text{hlist } ls \rightarrow \text{hlist } (x :: ls)$.

Variable $elm : A$.

Inductive `member` : `list A` \rightarrow `Type` :=

| `HFirst` : $\forall ls, \text{member } (elm :: ls)$

| `HNext` : $\forall x\ ls, \text{member } ls \rightarrow \text{member } (x :: ls)$.

Heterogenic lists

Section `hlist`.

Variable $A : \text{Type}$.

Variable $B : A \rightarrow \text{Type}$.

Inductive **`hlist`** : **`list`** $A \rightarrow \text{Type} :=$

| **`HNil`** : **`hlist`** `nil`

| **`HCons`** : $\forall (x : A) (ls : \text{list } A), B\ x \rightarrow \text{hlist } ls \rightarrow \text{hlist } (x :: ls)$.

Variable $elm : A$.

Inductive **`member`** : **`list`** $A \rightarrow \text{Type} :=$

| **`HFirst`** : $\forall ls, \text{member } (elm :: ls)$

| **`HNext`** : $\forall x\ ls, \text{member } ls \rightarrow \text{member } (x :: ls)$.

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 $[A B \times Is]$.

Arguments HNil $[A B]$.

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

Example someValues : **hlist** (fun T : Set $\Rightarrow T$) someTypes :=
HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

= 5
: (fun T : Set $\Rightarrow T$) **nat**

Eval simpl in hget someValues (HNext HFirst).

= true
: (fun T : Set $\Rightarrow T$) **bool**

Examples of heterogenic lists

Arguments HCons $[A B \times Is]$.

Arguments HNil $[A B]$.

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

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

Eval simpl in hget someValues HFirst.

```
= 5
: (fun T : Set  $\Rightarrow$  T) nat
```

Eval simpl in hget someValues (HNext HFirst).

```
= true
: (fun T : Set  $\Rightarrow$  T) bool
```

Examples of heterogenic lists

Arguments HCons $[A B \times ls]$.

Arguments HNil $[A B]$.

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

Example someValues : **hlist** (fun T : Set $\Rightarrow T$) someTypes :=
HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

```
= 5
: (fun T : Set  $\Rightarrow T$ ) nat
```

Eval simpl in hget someValues (HNext HFirst).

```
= true
: (fun T : Set  $\Rightarrow T$ ) bool
```

Examples of heterogenic lists

Arguments HCons $[A B \times Is]$.

Arguments HNil $[A B]$.

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

Example someValues : **hlist** (fun T : Set $\Rightarrow T$) someTypes :=
HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

= 5
: (fun T : Set $\Rightarrow T$) **nat**

Eval simpl in hget someValues (HNext HFirst).

= true
: (fun T : Set $\Rightarrow T$) **bool**

Examples of heterogenic lists

Arguments HCons $[A B \times Is]$.

Arguments HNil $[A B]$.

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

Example someValues : **hlist** (fun T : Set $\Rightarrow T$) someTypes :=
HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

= 5
: (fun T : Set $\Rightarrow T$) **nat**

Eval simpl in hget someValues (HNext HFirst).

= true
: (fun T : Set $\Rightarrow T$) **bool**

Examples of heterogenic lists

Arguments HCons $[A B \times Is]$.

Arguments HNil $[A B]$.

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

Example someValues : **hlist** (fun T : Set $\Rightarrow T$) someTypes :=
HCons 5 (HCons true HNil).

Eval simpl in hget someValues HFirst.

= 5
: (fun T : Set $\Rightarrow T$) **nat**

Eval simpl in hget someValues (HNext HFirst).

= true
: (fun T : Set $\Rightarrow T$) **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 λ^{\rightarrow} — summary

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

Interpreter of λ^{\rightarrow} — summary

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

Interpreter of λ^{\rightarrow} — summary

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