

Zależne struktury danych

Daria Walukiewicz-Chrząszcz

28 marca 2017

Destrukcja - 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

dla $m : I \vec{a} \vec{b}$ powyższe wyrażenie ma typ $P \vec{b} m$

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

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

I jeśli

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

to

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

Destrukcja - 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

dla $m : I \vec{a} \vec{b}$ powyższe wyrażenie ma typ $P \vec{b} m$

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

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

I jeśli

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

to

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

Destrukcja - 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

dla $m : I \vec{a} \vec{b}$ powyższe wyrażenie ma typ $P \vec{b} m$

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

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

I jeśli

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

to

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

Destrukcja - 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

dla $m : I \vec{a} \vec{b}$ powyższe wyrażenie ma typ $P \vec{b} m$

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

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

I jeśli

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

to

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

Destrukcja - 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

dla $m : I \vec{a} \vec{b}$ powyższe wyrażenie ma typ $P \vec{b} m$

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

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

I jeśli

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

to

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

Funkcja app na listach z długością

```

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
  | Cons _ x ls1' => Cons x (app' ls1' ls2)
  end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (Sn')$ to powinno być

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

czyli

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

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

Funkcja app na listach z długością

```

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
  | Cons _ x ls1' => Cons x (app' ls1' ls2)
  end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (Sn')$ to powinno być

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

czyli

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

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

Funkcja app na listach z długością

```

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
  | Cons _ x ls1' => Cons x (app' ls1' ls2)
  end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (Sn')$ to powinno być

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

czyli

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

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

Funkcja app na listach z długością

```

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
  | Cons _ x ls1' => Cons x (app' ls1' ls2)
  end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall(n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (Sn')$ to powinno być

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

czyli

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

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

Funkcja app na listach z długością

```

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
  | Cons _ x ls1' => Cons x (app' ls1' ls2)
  end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall(n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (Sn')$ to powinno być

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

czyli

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

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

Eliminacja równości

```

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

```

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

Ponieważ $\text{eq_refl} : \forall (A : \text{Set})(a : A).\text{eq } A \ a \ a$ oraz $h : \text{eq } \text{nat } \ n \ m$ to powinno być

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

czyli

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

- To była eliminacja z Prop w Set dla typu singletonowego
- Tak działa taktyka `rewrite`

Eliminacja równości

```

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

```

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

Ponieważ $\text{eq_refl} : \forall (A : \text{Set})(a : A).\text{eq } A \ a \ a$ oraz $h : \text{eq } \text{nat } \ n \ m$ to powinno być

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

czyli

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

- To była eliminacja z Prop w Set dla typu singletonowego
- Tak działa taktyka `rewrite`

Eliminacja równości

```

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

```

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

Ponieważ $\text{eq_refl} : \forall (A : \text{Set})(a : A).\text{eq } A a a$ oraz $h : \text{eq } \text{nat } n m$ to powinno być

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

czyli

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

- To była eliminacja z Prop w Set dla typu singletonowego
- Tak działa taktyka `rewrite`

Eliminacja równości

```

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

```

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

Ponieważ $\text{eq_refl} : \forall (A : \text{Set})(a : A).\text{eq } A a a$ oraz $h : \text{eq } \text{nat } n m$ to powinno być

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

czyli

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

- To była eliminacja z Prop w Set dla typu singletonowego
- Tak działa taktyka `rewrite`

Eliminacja równości

```

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

```

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

Ponieważ $\text{eq_refl} : \forall (A : \text{Set})(a : A).\text{eq } A a a$ oraz $h : \text{eq } \text{nat } n m$ to powinno być

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

czyli

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

- To była eliminacja z Prop w Set dla typu singletonowego
- Tak działa taktyka `rewrite`

Eliminacja równości

```

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

```

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

Ponieważ $\text{eq_refl} : \forall (A : \text{Set})(a : A).\text{eq } A a a$ oraz $h : \text{eq } \text{nat } n m$ to powinno być

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

czyli

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

- To była eliminacja z Prop w Set dla typu singletonowego
- Tak działa taktyka rewrite

Funkcja hd1 na listach z długością

```
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
| Cons _h _ => h
end
```

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

Funkcja hd1 na listach z długością

```
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
| Cons _h _ => h
end

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

Funkcja hd1 na listach z długością cd.

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

Ponieważ $Nil : ilist\ 0$ to powinno być

$$f1 : P\ 0\ Nil$$

Ponieważ $Cons : \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (Sn')$ to powinno być

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

czyli

$$f1 : unit$$

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

Funkcja hd1 na listach z długością cd.

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

Ponieważ $Nil : ilist\ 0$ to powinno być

$$f1 : P\ 0\ Nil$$

Ponieważ $Cons : \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (Sn')$ to powinno być

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

czyli

$$f1 : unit$$

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

Funkcja hd1 na listach z długością cd.

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

Ponieważ $Nil : ilist\ 0$ to powinno być

$$f1 : P\ 0\ Nil$$

Ponieważ $Cons : \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (Sn')$ to powinno być

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

czyli

$$f1 : unit$$

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

Funkcja hd1 na listach z długością cd.

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

Ponieważ $Nil : ilist\ 0$ to powinno być

$$f1 : P\ 0\ Nil$$

Ponieważ $Cons : \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (Sn')$ to powinno być

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

czyli

$$f1 : unit$$

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

Funkcja hd1 na listach z długością cd.

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

Ponieważ $Nil : ilist\ 0$ to powinno być

$$f1 : P\ 0\ Nil$$

Ponieważ $Cons : \forall(n' : nat)(a : A)(l : ilist\ n'), ilist\ (Sn')$ to powinno być

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

czyli

$$f1 : unit$$

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

Funkcja hd2 na listach z długością

```

Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
  return (match n with 0 => unit | S_=> A end) with
| Nil => tt
| Cons _h _=> h
end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (S n')$ to powinno być

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

czyli

$$f1 : \text{unit}$$

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

Funkcja hd2 na listach z długością

```

Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
  return (match n with 0 => unit | S_=> A end) with
| Nil => tt
| Cons _h _=> h
end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (S n')$ to powinno być

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

czyli

$$f1 : \text{unit}$$

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

Funkcja hd2 na listach z długością

```

Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
  return (match n with 0 => unit | S_=> A end) with
| Nil => tt
| Cons _h _=> h
end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (S n')$ to powinno być

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

czyli

$$f1 : \text{unit}$$

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

Funkcja hd2 na listach z długością

```

Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
  return (match n with 0 => unit | S_=> A end) with
| Nil => tt
| Cons _h _=> h
end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (S n')$ to powinno być

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

czyli

$$f1 : \text{unit}$$

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

Funkcja hd2 na listach z długością

```

Definition hd_pom n (ls : ilist n) :=
match ls in (ilist n)
  return (match n with 0 => unit | S_=> A end) with
| Nil => tt
| Cons _h _=> h
end.

```

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

Ponieważ $\text{Nil} : \text{ilist } 0$ to powinno być

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

Ponieważ $\text{Cons} : \forall (n' : \text{nat})(a : A)(l : \text{ilist } n'), \text{ilist } (S n')$ to powinno być

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

czyli

$$f1 : \text{unit}$$

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

Listy z długością

Section `ilist`.

Variable `A : Set`.

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

| `Nil` : `ilist` `0`

| `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`).

Wartości typu `fin 3` to: `First 2`, `Next (First 1)`, i `Next (Next (First 0))`.

Uwaga: nie ma termów typu `fin 0` !

Listy z długością

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)`.

Wartości typu `fin 3` to: `First 2`, `Next (First 1)`, i `Next (Next (First 0))`.

Uwaga: nie ma termów typu `fin 0` !

Listy z długością

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)`.

Wartości typu `fin 3` to: `First 2`, `Next (First 1)`, i `Next (Next (First 0))`.

Uwaga: nie ma termów typu `fin 0` !

Listy z długością

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)`.

Wartości typu `fin 3` to: `First 2`, `Next (First 1)`, i `Next (Next (First 0))`.

Uwaga: nie ma termów typu `fin 0` !

Funkcja get — szkielet

```

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.

```

Funkcja get — cd.

```

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 ilist (pred n') → A with
    | First _ ⇒ fun _ ⇒ x
    | Next _ idx' ⇒ fun ls' ⇒ get ls' idx'
    end ls'
  end.

```

Trzeci return konieczny, żeby powiązać typ idx' i typ idx . Pozostaje problem z wywołaniem rekurencyjnym `get`

Funkcja get — cd.

```

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 ilist (pred n') → A with
    | First _ ⇒ fun _ ⇒ x
    | Next _ idx' ⇒ fun ls' ⇒ get ls' idx'
    end ls'
  end.

```

Trzeci return konieczny, żeby powiązać typ idx' i typ idx . Pozostaje problem z wywołaniem rekurencyjnym `get`

Funkcja get — cd.

```

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.

```

Implicit Arguments Nil [A]. Implicit Arguments First [n].

Przykłady

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
```

Przykłady

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**

Przykłady

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**

Przykłady

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**

Przykłady

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
```

Przykłady

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**

Przykłady

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
```

Przykłady

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
```

Listy heterogeniczne

Section `hlist`.

Variable A : Type.

Variable B : $A \rightarrow$ 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` : $\forall x\ ls, \text{member } ls \rightarrow \text{member } (x :: ls)$.

Listy heterogeniczne

Section `hlist`.

Variable A : Type.

Variable B : $A \rightarrow$ Type.

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

| HNil : **hlist** nil

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

Variable elm : A .

Inductive **member** : **list** $A \rightarrow$ Type :=

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

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

Listy heterogeniczne

Section `hlist`.

Variable A : Type.

Variable B : $A \rightarrow$ Type.

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

| HNil : **hlist** nil

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

Variable elm : A .

Inductive **member** : **list** $A \rightarrow$ Type :=

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

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

Funkcja hget

```

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

```

Funkcja hget cd.

```

| 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.

```

```

Implicit Arguments HNil [A B].
Implicit Arguments HCons [A B x ls].
Implicit Arguments HFirst [A elm ls].
Implicit Arguments HNext [A elm x ls].

```

Funkcja hget cd.

```

| 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.

```

```

Implicit Arguments HNil [A B].
Implicit Arguments HCons [A B x ls].
Implicit Arguments HFirst [A elm ls].
Implicit Arguments HNext [A elm x ls].

```

Przykłady list heterogenicznych

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**

Przykłady list heterogenicznych

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**

Przykłady list heterogenicznych

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**

Przykłady list heterogenicznych

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**

Przykłady list heterogenicznych

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**

Przykłady list heterogenicznych

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 lambda rachunku z typami prostymi

```

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).

```

```

Implicit Arguments Const [ts].

```

Interpreter lambda rachunku z typami prostymi

```

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).

```

```

Implicit Arguments Const [ts].

```

Interpreter lambda rachunku z typami prostymi

```

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).

```

```

Implicit Arguments Const [ts].

```

Interpreter lambda rachunku z typami prostymi cd.

```

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 lambda rachunku z typami prostymi cd.

```

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 lambda rachunku z typami prostymi — przykłady

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 lambda rachunku z typami prostymi — przykłady

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 lambda rachunku z typami prostymi — przykłady

Eval simpl in expDenote Const HNil.

= tt : typeDenote Unit

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

= fun x : unit \Rightarrow x

: typeDenote (Arrow Unit Unit)

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

= fun x _ : unit \Rightarrow x

: typeDenote (Arrow Unit (Arrow Unit Unit))

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

= fun _ x0 : unit \Rightarrow x0

: typeDenote (Arrow Unit (Arrow Unit Unit))

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

= tt : typeDenote Unit

Interpreter lambda rachunku z typami prostymi — przykłady

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 lambda rachunku z typami prostymi — przykłady

Eval simpl in expDenote Const HNil.

= tt : typeDenote Unit

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

= fun x : unit \Rightarrow x

: typeDenote (Arrow Unit Unit)

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

= fun x _ : unit \Rightarrow x

: typeDenote (Arrow Unit (Arrow Unit Unit))

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

= fun _ x0 : unit \Rightarrow x0

: typeDenote (Arrow Unit (Arrow Unit Unit))

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

= tt : typeDenote Unit

Interpreter — podsumowanie

- składnia, reguły typowania i semantyka obliczania dla lambda rachunku z typami prostymi
- interpreter = implementacja w Coqu semantyki denotacyjnej
- własności modelowanego języka wynikają z własności CIC (zachowywanie typu, poprawność typowa, silna normalizacja)

Interpreter — podsumowanie

- składnia, reguły typowania i semantyka obliczania dla lambda rachunku z typami prostymi
- interpreter = implementacja w Coqu semantyki denotacyjnej
- własności modelowanego języka wynikają z własności CIC (zachowywanie typu, poprawność typowa, silna normalizacja)

Interpreter — podsumowanie

- składnia, reguły typowania i semantyka obliczania dla lambda rachunku z typami prostymi
- interpreter = implementacja w Coqu semantyki denotacyjnej
- własności modelowanego języka wynikają z własności CIC (zachowywanie typu, poprawność typowa, silna normalizacja)