Programowanie z typami zależnymi i dowodzenie twierdzeń

Zależne struktury danych

31.03.2015

Listy z długością

```
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).
```

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

Listy z długością

```
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).
```

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

Listy z długością

```
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).
```

Wartości typu **fin** 3 to: First 2, `Next` (First 1), i `Next` (`Next` (First 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 powiązac 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 powiązac 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
```

Przykład dowodu z get

```
Section ilist_map.
  Variables A B : Set.
  Variable f : A → B.

  Fixpoint imap n (ls : ilist A n) : ilist B n :=
    match ls with
      | Nil ⇒ Nil
      | Cons _ x ls' ⇒ Cons (f x) (imap ls')
    end.

  Theorem get_imap : ∀ n (idx : fin n) (ls : ilist A n),
    get (imap ls) idx = f (get ls idx).
    induction ls; dep_destruct idx; crush.
  Qed.
End ilist_map.
```

Uwaga na dep_destruct

Przykład dowodu z get

```
Section ilist_map.
  Variables A B : Set.
  Variable f : A → B.

  Fixpoint imap n (ls : ilist A n) : ilist B n :=
    match ls with
      | Nil ⇒ Nil
      | Cons _ x ls' ⇒ Cons (f x) (imap ls')
    end.


  Theorem get_imap : ∀ n (idx : fin n) (ls : ilist A n),
    get (imap ls) idx = f (get ls idx).
    induction ls; dep_destruct idx; crush.
  Qed.
End ilist_map.
```

Uwaga na dep_destruct

Przykład dowodu z get

```
Section ilist_map.
  Variables A B : Set.
  Variable f : A → B.

  Fixpoint imap n (ls : ilist A n) : ilist B n :=
    match ls with
      | Nil ⇒ Nil
      | Cons _ x ls' ⇒ Cons (f x) (imap ls')
    end.


  Theorem get_imap : ∀ n (idx : fin n) (ls : ilist A n),
    get (imap ls) idx = f (get ls idx).
    induction ls; dep_destruct idx; crush.
  Qed.
End ilist_map.
```

Uwaga na `dep_destruct`

Listy heterogeniczne

```
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).
```

Listy heterogeniczne

```
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).
```

Listy heterogeniczne

```
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).
```

Funkcja hget

```
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
```

Funkcja hget cd.

```
    | HCons _ _ x 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 x _ ⇒ x
        | HNext _ _ mem' ⇒ fun _ get_mls' ⇒ get_mls' mem'
      end x (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 _ _ x 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 x _ ⇒ x
        | HNext _ _ mem' ⇒ fun _ get_mls' ⇒ get_mls' mem'
      end x (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 ⇒ 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** ⇒ 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 — 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)

Rekurencyjna wersja `ilist` i `fin`

```
Section filist.
  Variable A : Set.

  Fixpoint filist (n : nat) : Set :=
    match n with
      | O ⇒ unit
      | S n' ⇒ A × filist n'
    end%type.

  Fixpoint ffin (n : nat) : Set :=
    match n with
      | O ⇒ Empty_set
      | S n' ⇒ option (ffin n')
    end.
```

Wartościami ffin 3 są: None, Some None, i Some (Some None)

Rekurencyjna wersja `ilist` i `fin`

```
Section filist.
  Variable A : Set.

  Fixpoint filist (n : nat) : Set :=
    match n with
      | O ⇒ unit
      | S n' ⇒ A × filist n'
    end%type.

  Fixpoint ffin (n : nat) : Set :=
    match n with
      | O ⇒ Empty_set
      | S n' ⇒ option (ffin n')
    end.
```

Wartościami ffin 3 są: None, Some None, i Some (Some None)

Rekurencyjna wersja ilist i fin

```
Section filist.
  Variable A : Set.

  Fixpoint filist (n : nat) : Set :=
    match n with
      | O ⇒ unit
      | S n' ⇒ A × filist n'
    end%type.

  Fixpoint ffin (n : nat) : Set :=
    match n with
      | O ⇒ Empty_set
      | S n' ⇒ option (ffin n')
    end.
```

Wartościami ffin 3 są: None, Some None, i Some (Some None)

Funkcja fget

```
Fixpoint fget (n : nat) : filist n → ffin n → A :=
  match n with
    | O ⇒ fun _ idx ⇒ match idx with end
    | S n' ⇒ fun ls idx ⇒
      match idx with
        | None ⇒ fst ls
        | Some idx' ⇒ fget n' (snd ls) idx'
      end
  end.

End filist.
```

Listy heterogeniczne rekurencyjne

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

  Fixpoint fhlist (ls : list A) : Type :=
    match ls with
      | nil ⇒ unit
      | x :: ls' ⇒ B x × fhlist ls'
    end%type.

  Variable elm : A.

  Fixpoint fmember (ls : list A) : Type :=
    match ls with
      | nil ⇒ Empty_set
      | x :: ls' ⇒ (x = elm) + fmember ls'
    end%type.
```

Listy heterogeniczne rekurencyjne

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

  Fixpoint fhlist (ls : list A) : Type :=
    match ls with
      | nil ⇒ unit
      | x :: ls' ⇒ B x × fhlist ls'
    end%type.

  Variable elm : A.

  Fixpoint fmember (ls : list A) : Type :=
    match ls with
      | nil ⇒ Empty_set
      | x :: ls' ⇒ (x = elm) + fmember ls'
    end%type.
```

Funkcja fhget

```
Fixpoint fhget (ls : list A) : fhlist ls → fmember ls → B elm :=
  match ls with
    | nil ⇒ fun _ idx ⇒ match idx with end
    | _ :: ls' ⇒ fun mls idx ⇒
      match idx with
        | inl pf ⇒ match pf with
                     | eq_refl ⇒ fst mls
                   end
        | inr idx' ⇒ fhget ls' (snd mls) idx'
      end
  end.
```

Potrzebny match po pf

Funkcja fhget

```
Fixpoint fhget (ls : list A) : fhlist ls → fmember ls → B elm :=
  match ls with
    | nil ⇒ fun _ idx ⇒ match idx with end
    | _ :: ls' ⇒ fun mls idx ⇒
      match idx with
        | inl pf ⇒ match pf with
                     | eq_refl ⇒ fst mls
                   end
        | inr idx' ⇒ fhget ls' (snd mls) idx'
      end
  end.
```

Potrzebny match po pf

Typ drzew o arności zadanej listą

```
Section tree.
  Variable A : Set.

  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, ilist tree n → tree.
End tree.
```

Funkcja ifolfdr

```
Section ifoldr.
  Variables A B : Set.
  Variable f : A → B → B.
  Variable i : B.

  Fixpoint ifoldr n (ls : ilist A n) : B :=
    match ls with
      | Nil ⇒ i
      | Cons _ x ls' ⇒ f x (ifoldr ls')
    end.
End ifoldr.
```

Przykłady użycia `ifolfdr`

```
Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ ls ⇒ ifoldr (fun t' n ⇒ sum t' + n) O ls
  end.
```

```
Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ ls ⇒ Node (imap inc ls)
  end.
```

```
Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  induction t; crush.
```

Zasada indukcji generowana automatycznie jest za słaba. Można ją wzmocnić lub próbować zmienić typ `tree`

Przykłady użycia `ifolfdr`

```
Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ ls ⇒ ifoldr (fun t' n ⇒ sum t' + n) O ls
  end.

Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ ls ⇒ Node (imap inc ls)
  end.
```

```
Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  induction t; crush.
```

Zasada indukcji generowana automatycznie jest za słaba. Można ją wzmocnić lub próbować zmienić typ `tree`

Przykłady użycia `ifolfdr`

```
Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ ls ⇒ ifoldr (fun t' n ⇒ sum t' + n) O ls
  end.

Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ ls ⇒ Node (imap inc ls)
  end.

Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  induction t; crush.
```

Zasada indukcji generowana automatycznie jest za słaba. Można ją wzmocnić lub próbować zmienić typ `tree`

Zmiana 1: użycie list rekurencyjnych

```
Section tree.
  Variable A : Set.

  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, filist tree n → tree.
```

```
Error: Non strictly positive occurrence of " tree" in " forall
n : nat, filist tree n -> tree"
```

Zmiana 1: użycie list rekurencyjnych

```
Section tree.
  Variable A : Set.

  Inductive tree : Set :=
  | Leaf : A → tree
  | Node : ∀ n, filist tree n → tree.
```

```
Error: Non strictly positive occurrence of " tree" in " forall
n : nat, filist tree n -> tree"
```

Zmiana 2: użycie typu indukcyjnego funkcyjnego

```
Inductive tree : Set :=
| Leaf : A → tree
| Node : ∀ n, (ffin n → tree) → tree.

End tree.

Implicit Arguments Node [A n].
```

Zmiana 2: użycie typu indukcyjnego funkcyjnego

```
Inductive tree : Set :=
| Leaf : A → tree
| Node : ∀ n, (ffin n → tree) → tree.

End tree.

Implicit Arguments Node [A n].
```

Funkcja rifoldr

```
Section rifoldr.
  Variables A B : Set.
  Variable f : A → B → B.
  Variable i : B.

  Fixpoint rifoldr (n : nat) : (ffin n → A) → B :=
    match n with
      | O ⇒ fun _ ⇒ i
      | S n' ⇒ fun get ⇒ f (get None) (rifoldr n' (fun idx ⇒ get
(Some idx)))
    end.
End rifoldr.

Implicit Arguments rifoldr [A B n].
```

Przykłady użycia `rifoldr`

```
Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ f ⇒ rifoldr plus O (fun idx ⇒ sum (f idx))
  end.
```

```
Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ f ⇒ Node (fun idx ⇒ inc (f idx))
  end.
```

Uwaga: implementacja `inc` bez użycia dodatkowych funkcji

Przykłady użycia `rifoldr`

```
Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ f ⇒ rifoldr plus O (fun idx ⇒ sum (f idx))
  end.

Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ f ⇒ Node (fun idx ⇒ inc (f idx))
  end.
```

Uwaga: implementacja `inc` bez użycia dodatkowych funkcji

Przykłady użycia `rifoldr`

```
Fixpoint sum (t : tree nat) : nat :=
  match t with
    | Leaf n ⇒ n
    | Node _ f ⇒ rifoldr plus O (fun idx ⇒ sum (f idx))
  end.

Fixpoint inc (t : tree nat) : tree nat :=
  match t with
    | Leaf n ⇒ Leaf (S n)
    | Node _ f ⇒ Node (fun idx ⇒ inc (f idx))
  end.
```

Uwaga: implementacja `inc` bez użycia dodatkowych funkcji

Dowód przez indukcję

```
Lemma plus_ge : ∀ x1 y1 x2 y2,
  x1 ≥ x2
  → y1 ≥ y2
  → x1 + y1 ≥ x2 + y2.
  crush.
Qed.

Lemma sum_inc' : ∀ n (f1 f2 : ffin n → nat),
  (∀ idx, f1 idx ≥ f2 idx)
  → rifoldr plus O f1 ≥ rifoldr plus O f2.
  Hint Resolve plus_ge.

  induction n; crush.
Qed.

Theorem sum_inc : ∀ t, sum (inc t) ≥ sum t.
  Hint Resolve sum_inc'.

  induction t; crush.
Qed.
```

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach

Podsumowanie

Typ indukcyjny:

- dużo anotacji przy `match`
- logika i taktyki Coqa dopracowane do typów indukcyjnych

Reprezentacja rekurencyjna:

- łatwiejsze początkowe definicje
- simpl może za dużo rozwinąć, np `filist A (S n)`
- mniej oczywiste dopasowywanie typów
- nadaje się do typów w których indeksy "oddają" kształt typu

Typ indukcyjny funkcyjny:

- trudnieszy do czytania kod (manipulacje indeksami)
- dobre przy małych formalizacjach