Programowanie z typami zależnymi i dowodzenie twierdzeń

Równość

14.04.2015

Typowanie jako relacja indukcyjna

```
Inductive exp : Set :=
| Nat : nat → exp
| Plus : exp → exp → exp
| Bool : bool → exp
| And : exp → exp → exp.
```

Typowanie jako relacja indukcyjna

```
Inductive type : Set := TNat | TBool.

Inductive hasType : exp → type → Prop :=
| HtNat : ∀ n,
  hasType (Nat n) TNat
| HtPlus : ∀ e1 e2,
  hasType e1 TNat
  → hasType e2 TNat
  → hasType (Plus e1 e2) TNat
| HtBool : ∀ b,
  hasType (Bool b) TBool
| HtAnd : ∀ e1 e2,
  hasType e1 TBool
  → hasType e2 TBool
  → hasType (And e1 e2) TBool.
```

Rozstrzygalność porównywania typów

```
Definition eq_type_dec : ∀ t1 t2 : type, {t1 = t2} + {t1 ≠ t2}.
  decide equality.
Defined.
```

Dowód przez indukcję po dowodzie

```
Lemma hasType_det : ∀ e t1,
  hasType e t1
  → ∀ t2, hasType e t2
    → t1 = t2.
  induction 1; inversion 1; crush.
Qed.
```

Patrz plik: hastype.v

Konwersja — równość definicyjna, obliczeniowa

ang. *d*efinitional equality

reguła konwersji

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} B \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$$

$$\Gamma \vdash A =_{\beta\eta\delta\zeta\iota} B$$

jeśli

- $\Gamma \vdash A \rhd^* A'$
- $\Gamma \vdash B \rhd^* B'$
- $A' = B'$ lub
  $(A' = \lambda x : T.A''$ i $\Gamma, x : T \vdash B' x =_{\beta\eta\delta\zeta\iota} A'')$ lub
  $(B' = \lambda x : T.B''$ i $\Gamma, x : T \vdash A' x =_{\beta\eta\delta\zeta\iota} B'')$

$$\Gamma \vdash A \rhd B$$

domknięcie przechodnie redukcji beta, iota, delta i zeta.

Konwersja — równość definicyjna, obliczeniowa

ang. *d*efinitional equality

reguła konwersji

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} B \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$$

$$\Gamma \vdash A =_{\beta\eta\delta\zeta\iota} B$$

jeśli

- $\Gamma \vdash A \triangleright^* A'$
- $\Gamma \vdash B \triangleright^* B'$
- $A' = B'$ lub
  $(A' = \lambda x : T.A''$ i $\Gamma, x : T \vdash B'x =_{\beta\eta\delta\zeta\iota} A'')$ lub
  $(B' = \lambda x : T.B''$ i $\Gamma, x : T \vdash A'x =_{\beta\eta\delta\zeta\iota} B'')$

$$\Gamma \vdash A \triangleright B$$

domknięcie przechodnie redukcji beta, iota, delta i zeta.

Reguły redukcji na przykładzie

```
Definition pred' (x : nat) :=
  match x with
    | O ⇒ O
    | S n' ⇒ let y := n' in y
  end.

Theorem reduce_me : pred' 1 = 0.


  cbv delta.
```

Reguły redukcji w taktyce cbv

```
============================
 (fun x : nat ⇒ match x with
                | 0 ⇒ 0
                | S n' ⇒ let y := n' in y
                end) 1 = 0
```

```
cbv beta.
```

Reguły redukcji w taktyce cbv

```
============================
 match 1 with
 | 0 ⇒ 0
 | S n' ⇒ let y := n' in y
 end = 0
```

```
cbv iota.
```

Reguły redukcji w taktyce cbv

```
============================
 (fun n' : nat ⇒ let y := n' in y) 0 = 0

cbv beta.

============================
 (let y := 0 in y) = 0

cbv zeta.

============================
 0 = 0
```

eq — równość definiowalna

ang. *p*ropositional equality
definiowana jako relacja indukcyjna

$$\texttt{Inductive}\ \mathbf{eq}\ (A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \texttt{eq\_refl} : x = x$$

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                        P x -> forall y : A, x = y -> P y
```

eq — równość definiowalna

ang. *p*ropositional equality
definiowana jako relacja indukcyjna

$$\texttt{Inductive } \mathbf{eq}\ (A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \texttt{eq\_refl} : x = x$$

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                        P x -> forall y : A, x = y -> P y
```

Problemy z równością

lemmaUIP jest niedowodliwy:

```
Lemma lemmaUIP : ∀ (x : A) (pf : x = x), pf = eq_refl x.
```

dowodliwy jest lemma2:

```
Lemma lemma2 : ∀ (x : A) (pf : x = x), O = match pf with eq_refl ⇒
O end.
```

Dowód lemma2

```
Definition lemma2 :=
  fun (x : A) (pf : x = x) =>
    match pf return (0 = match pf with
                         | eq_refl => 0
                         end) with
      | eq_refl => eq_refl 0
    end.
```

Aksjomat UIP_refl

```
Check UIP_refl.
```

```
UIP_refl
     : ∀ (U : Type) (x : U) (p : x = x), p = eq_refl x
```

Aksjomat UIP_refl dowodliwy z aksjomatu Streicher_K

```
Check Streicher_K.
```

```
Streicher_K
     : ∀ (U : Type) (x : U) (P : x = x → Prop),
       P eq_refl → ∀ p : x = x, P p
```

Streicher's axiom K jest niesprzeczny z CIC i niedowodliwy w CIC

Dla typów rozstrzygalnych

czyli takich, że

```
Variable eq_dec : forall x y:A, {x = y} + {x <> y}.
```

zachodzą prawa `UIP_dec` i `K_dec`

```
forall (x y:A) (p1 p2:x = y), p1 = p2
```

```
forall (x:A) (P:x = x -> Prop), P (eq_refl x)
                                  -> forall p:x = x, P p
```

bez dodatkowych aksjomatów (patrz: moduł `Eqdep_dec` z biblioteki standardowej)

Problemy z równością cd.

Przypomnijmy:

```
Section fhapp.
  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.

End fhlists.
```

Problemy z równością cd.

Przypomnijmy:

```
Section fhapp.
  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.

End fhlists.
```

Funkcja fhapp

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

  Fixpoint fhapp (ls1 ls2 : list A)
    : fhlist B ls1 → fhlist B ls2 → fhlist B (ls1 ++ ls2) :=
    match ls1 with
      | nil ⇒ fun _ hls2 ⇒ hls2
      | _ :: _ ⇒ fun hls1 hls2 ⇒ (fst hls1 , fhapp _ _ (snd hls1) hls2)
    end.

  Implicit Arguments fhapp [ls1 ls2].
```

Poniższego twierdzenia "nie da się zapisać"

Theorem fhapp_assoc : $\forall$ *ls1 ls2 ls3*
(*hls1* : fhlist *B ls1*) (*hls2* : fhlist *B ls2*) (*hls3* : fhlist *B ls3*),
fhapp *hls1* (fhapp *hls2 hls3*) = fhapp (fhapp *hls1 hls2*) *hls3*.

```
The term "fhapp (ls1:=ls1 ++ ls2) (ls2:=ls3) (fhapp (ls1:=ls1)
(ls2:=ls2) hls1 hls2) hls3" has type "fhlist B ((ls1 ++ ls2)
++ ls3)" while it is expected to have type "fhlist B (ls1 ++
ls2 ++ ls3)"
```

Można wstawić "type-cast"

```
Theorem fhapp_assoc : ∀ ls1 ls2 ls3
  (pf : (ls1 ++ ls2) ++ ls3 = ls1 ++ (ls2 ++ ls3))
  (hls1 : fhlist B ls1) (hls2 : fhlist B ls2) (hls3 : fhlist B ls3),
  fhapp hls1 (fhapp hls2 hls3)
  = match pf in (_ = ls) return fhlist _ ls with
      | eq_refl ⇒ fhapp (fhapp hls1 hls2) hls3
    end.
```

Ale dowód jest wcale nie jest łatwy...

Równość heterogeniczna

```
Inductive JMeq (A : Type) (x : A) : ∀ B : Type, B → Prop :=
    JMeq_refl : JMeq x x
```

```
Infix "==" := JMeq (at level 70, no associativity).
```

Związki eq i JMeq

```
JMeq_rect
  : forall (A : Type) (x : A) (P : forall B : Type, B -> Type),
  P A x -> forall (B : Type) (b : B), x == b -> P B b

eq_rect
  : forall (A : Type) (x : A) (P : A -> Type),
  P x -> forall y : A, x = y -> P y
```

Związki eq i JMeq

```
Lemma eq_JMeq : ∀ (A : Type) (x y : A), x = y → x == y.
  intros; rewrite H; reflexivity.
Qed.
```

Ale odwrotna implikacja jest niedowodliwa:

```
Check JMeq_eq.
```

```
JMeq_eq
     : ∀ (A : Type) (x y : A), x == y → x = y
```

Związki eq i JMeq

```
Lemma eq_JMeq : ∀ (A : Type) (x y : A), x = y → x == y.
  intros; rewrite H; reflexivity.
Qed.
```

Ale odwrotna implikacja jest niedowodliwa:

```
Check JMeq_eq.
```

```
JMeq_eq
     : ∀ (A : Type) (x y : A), x == y → x = y
```

Aksjomat JMeq_eq

- może być bezpiecznie dodany do CIC
- może byc używany przez rewrite w postaci

```
JMeq_ind_r
  : forall (A : Type) (x : A) (P : A -> Type),
  P x -> forall y : A, x == y -> P y
```

Dowód `fhapp_assoc'`

Dwa sposoby:

- używający aksjomatu `JMeq_eq`
- używający naturalnej zasady indukcji dla JMeq

Użyte aksjomaty sprawdzamy za pomocą:

```
Print Assumptions fhapp_assoc'.
```

Patrz plik JMeqRew.v

Dowód `fhapp_assoc'`

Dwa sposoby:

- używający aksjomatu `JMeq_eq`
- używający naturalnej zasady indukcji dla JMeq

Użyte aksjomaty sprawdzamy za pomocą:

```
Print Assumptions fhapp_assoc'.
```

Patrz plik JMeqRew.v

Lemat fhapp_assoc'

```
Section fhapp'.
  Variable A : Type.
  Variable B : A → Type.

  Theorem fhapp_assoc' : ∀ ls1 ls2 ls3 (hls1 : fhlist B ls1) (hls2 : fhlist
B ls2)
    (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3) == fhapp (fhapp hls1 hls2) hls3.
    induction ls1; crush.
```

```
============================
 (a0, fhapp b (fhapp hls2 hls3)) == (a0, fhapp (fhapp b hls2) hls3)
```

```
  generalize (fhapp b (fhapp hls2 hls3))
    (fhapp (fhapp b hls2) hls3)
    (IHls1 _ _ b hls2 hls3).
```

```
  ============================
   ∀ (f : fhlist B (ls1 ++ ls2 ++ ls3))
     (f0 : fhlist B ((ls1 ++ ls2) ++ ls3)), f == f0 → (a0, f) == (a0,
f0)
```

```
============================
 (a0, fhapp b (fhapp hls2 hls3)) == (a0, fhapp (fhapp b hls2) hls3)


  generalize (fhapp b (fhapp hls2 hls3))
    (fhapp (fhapp b hls2) hls3)
    (IHls1 _ _ b hls2 hls3).


============================
 ∀ (f : fhlist B (ls1 ++ ls2 ++ ls3))
   (f0 : fhlist B ((ls1 ++ ls2) ++ ls3)), f == f0 → (a0, f) == (a0,
f0)
```

```
============================
 (a0, fhapp b (fhapp hls2 hls3)) == (a0, fhapp (fhapp b hls2) hls3)


  generalize (fhapp b (fhapp hls2 hls3))
    (fhapp (fhapp b hls2) hls3)
    (IHls1 _ _ b hls2 hls3).


============================
 ∀ (f : fhlist B (ls1 ++ ls2 ++ ls3))
   (f0 : fhlist B ((ls1 ++ ls2) ++ ls3)), f == f0 → (a0, f) == (a0,
f0)
```

```
  rewrite app_assoc.
```

```
============================
```

$\forall\ f\ f0 : \mathsf{fhlist}\ B\ (ls1 +\!\!+ ls2 +\!\!+ ls3),\ f == f0 \rightarrow (a0, f) == (a0, f0)$

```
  intros f f0 H; rewrite H; reflexivity.
Qed.

End fhapp'.
```

```
    rewrite app_assoc.
  ============================
   ∀ f f0 : fhlist B (ls1 ++ ls2 ++ ls3), f == f0 → (a0, f) == (a0,
f0)

    intros f f0 H; rewrite H; reflexivity.
  Qed.

End fhapp'.
```

Print Assumptions

```
Print Assumptions fhapp_assoc'.
```

```
Axioms:
```

JMeq_eq : $\forall\ (A : \texttt{Type})\ (x\ y : A),\ x == y \to x = y$

użyte w `rewrite H`

Dowód bez aksjomatów — def. pomocnicza

```
Lemma pair_cong : ∀ A1 A2 B1 B2 (x1 : A1) (x2 : A2) (y1 : B1) (y2 : B2),
  x1 == x2
  → y1 == y2
  → (x1, y1) == (x2, y2).
  intros until y2; intros Hx Hy; rewrite Hx; rewrite Hy; reflexivity.
Qed.

Hint Resolve pair_cong.
```

Dowód fhapp_assoc'' bez aksjomatów

```
Section fhapp''.
  Variable A : Type.
  Variable B : A → Type.

  Theorem fhapp_assoc'' : ∀ ls1 ls2 ls3 (hls1 : fhlist B ls1) (hls2 : fhlist
B ls2)
    (hls3 : fhlist B ls3),
    fhapp hls1 (fhapp hls2 hls3) == fhapp (fhapp hls1 hls2) hls3.
    induction ls1; crush.
  Qed.
End fhapp''.

Print Assumptions fhapp_assoc''.

Closed under the global context
```