

Dependent types and equality

Daria Walukiewicz-Chrząszcz

4 May 2021

Dependent type of n -tuples

Section tuple.

Variable T : Type.

```
Fixpoint tuple (n : nat) : Type :=  
  match n with  
  | 0 => unit  
  | S n => T * tuple n  
  end.
```

```
Definition tuple_hd a : tuple (S a) -> T := @fst _.
```

How to define the last element of a nonempty tuple ?

Dependent type of n -tuples

Section tuple.

Variable T : Type.

```
Fixpoint tuple (n : nat) : Type :=  
  match n with  
  | 0 => unit  
  | S n => T * tuple n  
  end.
```

Definition tuple_hd a : tuple (S a) -> T := @fst _.

How to define the last element of a nonempty tuple ?

Dependent type of n -tuples

Section tuple.

Variable T : Type.

```
Fixpoint tuple (n : nat) : Type :=  
  match n with  
  | 0 => unit  
  | S n => T * tuple n  
  end.
```

Definition tuple_hd a : tuple (S a) -> T := @fst _.

How to define the last element of a nonempty tuple ?

Last element by proof

```

Definition grabtype n: Type :=
  match n with 0 => unit | S n => T end.

```

Lemma lastL: forall (n: nat), tuple n -> grabtype n.

Proof.

```

induction n.

```

```

- simpl; trivial.

```

```

- simpl.

```

```

  destruct n.

```

```

  + intro H; destruct H; assumption.

```

```

  + simpl in IHn.

```

```

    intro.

```

```

    apply IHn.

```

```

    destruct X.

```

```

    simpl in t0.

```

```

    exact t0.

```

```

Defined.

```

Last element by proof

```

Definition grabtype n: Type :=
  match n with 0 => unit | S n => T end.

```

Lemma lastL: forall (n: nat), tuple n -> grabtype n.

Proof.

```

induction n.
- simpl; trivial.
- simpl.
  destruct n.
  + intro H; destruct H; assumption.
  + simpl in IHn.
    intro.
    apply IHn.
    destruct X.
    simpl in t0.
    exact t0.

```

Defined.

Definition of lastOfNonempty

```
Definition lastOfNonempty (n:nat)(t:tuple (S n)): T :=  
  lastL (S n) t.
```

```
Variable a b c: T.
```

```
Definition f: tuple 1 := (a, tt).
```

```
Definition g: tuple 2 := (b, f).
```

```
Definition h: tuple 3 := (c, g).
```

```
Eval compute in (lastOfNonempty h).
```

Definition of lastOfNonempty

```
Definition lastOfNonempty (n:nat)(t:tuple (S n)): T :=  
  lastL (S n) t.
```

Variable a b c: T.

```
Definition f: tuple 1 := (a,tt).
```

```
Definition g: tuple 2 := (b, f).
```

```
Definition h: tuple 3 := (c, g).
```

Eval compute in (lastOfNonempty h).

Last element by Fixpoint

```

Fixpoint lastF (n: nat): tuple n -> grabtype n:=
match n as x return (tuple x -> grabtype x) with
| 0 => fun t => t
| S m => fun t (* : tuple S m *) =>
  (match m as n1
   return ((tuple n1 -> grabtype n1) -> T * tuple n1 -> T)
  with
  | 0 => fun _ H => let (e, _) := H in e
  | S n1 => fun IHn0 X =>
    IHn0 (let (_,t0) := X in t0 )
  end) (lastF m) t
end.

```

Equivalence of two definitions of last

```
Lemma last_eq: forall n (t:tuple n), lastL n t = lastF n t.  
Proof.  
  intros.  
  reflexivity.  
Qed.
```

Typing as an inductive predicate (1)

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

Typing as an inductive predicate (2)

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.

Decidability of equality

Definition eq_type_dec : $\forall t1 t2 : \mathbf{type}, \{t1 = t2\} + \{t1 \neq t2\}$.
decide equality.

Defined.

Unicity of typing — by induction on proof

```
Lemma hasType_det :  $\forall e t1,$   
  hasType e t1  
   $\rightarrow \forall t2, \mathbf{hasType} e t2$   
     $\rightarrow t1 = t2.$   
  induction 1; inversion 1; auto.  
Qed.
```

See: `hastype.v`

Conversion — definitional equality

conversion rule

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

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

if

- $\Gamma \vdash A \triangleright^* A'$
- $\Gamma \vdash B \triangleright^* B'$
- A' and B' identical up to irrelevant subterms (of type $T : \text{SProp}$),
or
 ($A' = \lambda x : T. A''$ and $\Gamma, x : T \vdash B' x =_{\beta\eta\delta\zeta\iota} A''$) or
 ($B' = \lambda x : T. B''$ and $\Gamma, x : T \vdash A' x =_{\beta\eta\delta\zeta\iota} B''$)

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

transitive closure of beta, iota, delta and zeta reductions

Conversion — definitional equality

conversion rule

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

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

if

- $\Gamma \vdash A \triangleright^* A'$
- $\Gamma \vdash B \triangleright^* B'$
- A' and B' identical up to irrelevant subterms (of type $T : \text{SProp}$),
or
 $(A' = \lambda x : T. A'' \text{ and } \Gamma, x : T \vdash B'x =_{\beta\eta\delta\zeta\iota} A'')$ or
 $(B' = \lambda x : T. B'' \text{ and } \Gamma, x : T \vdash A'x =_{\beta\eta\delta\zeta\iota} B'')$

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

transitive closure of beta, iota, delta and zeta reductions

Reduction rules — examples

```
Definition pred' (x : nat) :=  
  match x with  
  | 0 => 0  
  | S n' => let y := n' in y  
end.
```

Theorem reduce_me : pred' 1 = 0.

Proof.

```
  cbv delta.
```

Reduction rules —tactic cbv

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

cbv beta.

Reduction rules —tactic cbv

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

cbv iota.

Reduction rules —tactic cbv

=====

 $(\text{fun } n' : \text{nat} \Rightarrow \text{let } y := n' \text{ in } y) 0 = 0$

cbv beta.

=====

 $(\text{let } y := 0 \text{ in } y) = 0$

cbv zeta.

=====

 $0 = 0$

eq — propositional equality

defined as inductive relation

```
Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : x = x
```

```
@eq_refl: forall (A : Type) (x : A), eq A x x
```

eq is Leibnitz equality:

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

eq — propositional equality

defined as inductive relation

```
Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : x = x
```

```
@eq_refl: forall (A : Type) (x : A), eq A x x
```

eq is Leibnitz equality:

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

eq — propositional equality

defined as inductive relation

```
Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : x = x
```

```
@eq_refl: forall (A : Type) (x : A), eq A x x
```

eq is Leibnitz equality:

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

Problems with equality

lemmaUIP is not provable:

Lemma lemmaUIP : $\forall (x : A) (pf : x = x), pf = \text{eq_refl } x$.

but lemma2 is provable:

Lemma lemma2 : $\forall (x : A) (pf : x = x),$
 $\text{O} = \text{match } pf \text{ with eq_refl} \Rightarrow \text{O end}.$

The proof of 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.
```

UIP_refl axiom

Check UIP_refl.

UIP_refl

$$: \forall (U : \text{Type}) (x : U) (p : x = x), p = \text{eq_refl } x$$

UIP_refl is equivalent with Streicher_K axiom

Check Streicher_K.

Streicher_K

$$: \forall (U : \text{Type}) (x : U) (P : x = x \rightarrow \text{Prop}), \\ P \text{ eq_refl} \rightarrow \forall p : x = x, P p$$

Streicher's axiom K is consistent with CIC and not provable in CIC

For decidable types...

i.e. for types satisfying:

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

`UIP_dec`, `UIP_refl` and `K_dec`

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

forall (x y:A) (p:x = x), p = eq_refl x

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

hold without additional axioms (see module `Eqdep_dec` from the standard library)

Lemma `UIP_refl_nat` is provable in Coq

```
UIP_refl_nat
  :  $\forall (x : nat) (p : x = x), p = eq\_refl\ x$ 
```

(see file `UIP_refl_nat.v`)

Problems with equality cont.

The following theorem cannot “be stated”

Theorem `vappend_assoc` : $\forall a b c$
 $(va : \text{vector } a) (vb : \text{vector } b) (vc : \text{vector } c),$
`vappend (vappend va vb) vc = vappend va (vappend vb vc).`

Error:

The term `"vappend va (vappend vb vc)"`
 has type `"vector (a + (b + c))"`
 while it is expected to have type `"vector (a + b + c)"`.

The need of the “type-cast”

```

Theorem vappend_assoc : ∀ a b c
  (va : vector a) (vb : vector b) (vc : vector c),
  vappend (vappend va vb) vc =
    match Plus.plus_assoc a b cin (_ = X)
    return vector X with
    | eq_refl ⇒ vappend va (vappend vb vc)
  end.

```

Heterogenic equality

```
Inductive JMeq (A : Type) (x : A) :  $\forall B : \text{Type}, B \rightarrow \text{Prop} :=$   
  JMeq_refl : JMeq x x
```

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


Relationship between eq and 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
```

Relationship between eq and JMeq

Lemma eq_JMeq : $\forall (A : \text{Type}) (x y : A), x = y \rightarrow x == y$.

intros; rewrite H; reflexivity.

Qed.

But the reverse implication is not provable (it is an axiom):

Check JMeq_eq.

JMeq_eq

: $\forall (A : \text{Type}) (x y : A), x == y \rightarrow x = y$

Relationship between eq and JMeq

Lemma eq_JMeq : $\forall (A : \text{Type}) (x y : A), x = y \rightarrow x == y$.

intros; rewrite H; reflexivity.

Qed.

But the reverse implication is not provable (it is an axiom):

Check *JMeq_eq*.

JMeq_eq

: $\forall (A : \text{Type}) (x y : A), x == y \rightarrow x = y$

JMeq_eq axiom

- can be safely added to CIC
- can be used by rewrite tactic according to:

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

The proof of `pairC'`

Two ways:

- using `JMeq_eq` axiom
- using standard induction rule for `JMeq`

Axioms in use can be listed:

```
Print Assumptions pairC'.
```

See file `JMeqRew.v`

The proof of `pairC'`

Two ways:

- using `JMeq_eq` axiom
- using standard induction rule for `JMeq`

Axioms in use can be listed:

```
Print Assumptions pairC'.
```

See file `JMeqRew.v`

Proofs using `UIP_refl_nat` and transparent type-cast

Two proofs of Lemma `vappend_assoc`

- 1 using `UIP_refl_nat`
- 2 using transparent definition of `plus_assoc`.

See file `vappend_assoc.v`