Typing recursors & Induction principles for predicates

Rob Tiemens

r.tiemens@student.ru.nl

course: Type theory

30 November 2012

Topics

Typing recursors
- Non-dependant recursion
- Dependant pattern matching
- Dependently typed recursors

Induction principles for predicates

Typing recursors

- Inductive type of sort `Set`
- Coq generates a recursive function with suffix `rec` and `ind`

Typing recursors

Natural numbers: recursor associated with an inductive type

For instance natural numbers

```
nat_rec
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Typing recursors

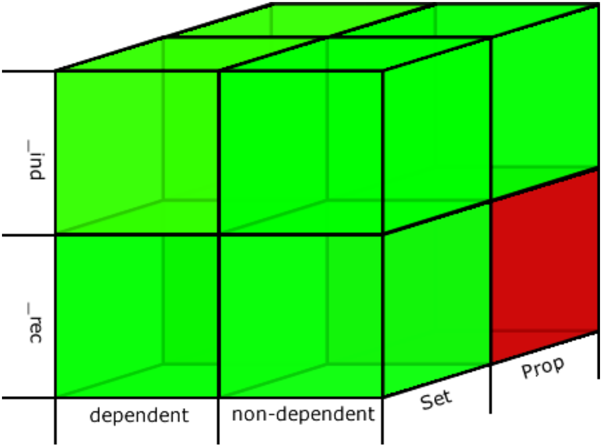
Natural numbers: recursor associated with an inductive type

For instance natural numbers

```
nat_rec
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n

nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Typing recursors



Typing recursors

Non-dependant recursion

But what about about `Fixpoint`?

Typing recursors

Non-dependant recursion

```
Fixpoint f (x:nat) : A :=
match x with
    0 -> exp
    S p -> exp2
end.
```

Typing recursors

Non-dependant recursion

```
Fixpoint f (x:nat) : A :=
match x with
    O -> exp
    S p -> exp2
end.
```

```
Fixpoint f (x:nat) : A :=
match x with
    O -> exp
    S p -> exp2; p (f p)
end.
```

a: set   exp: A   exp: nat -> A -> A

Typing recursors

Non-dependant recursion: example

```
Fixpoint plus (n p:nat) {struct n} : nat :=
  match n with
          | O => p
          | S m => S (plus m p)
 end.
```

```
Fixpoint plus' (n p:nat) {struct n} : nat :=
  match n with
          | O => p
          | S m => plus' m (S p)
 end.
```

Typing recursors

Dependant pattern matching

Typing recursors

Dependant pattern matching

```
Inductive listn : nat -> Set :=
 | niln : listn 0
 | consn : forall n:nat, nat -> listn n -> listn (S n).
```

Typing recursors

Dependant pattern matching

```
Inductive listn : nat -> Set :=
 | niln : listn 0
 | consn : forall n:nat, nat -> listn n -> listn (S n).

listn_rec
     : forall P : forall n : nat, listn n -> Set,
       P 0 niln ->
       (forall (n n0 : nat) (l : listn n), P n l ->
                    P (S n) (consn n n0 l)) ->
       forall (n : nat) (l : listn n), P n l

nat_rec
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Typing recursors

Dependant pattern matching

Function with dependent types.

```
Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l}
: listn (n + m) :=
 match l in listn n return listn (n + m) with
 | niln => l'
 | consn n' a y => consn (n' + m) a (concat n' y m l')
end.
```

Typing recursors

Dependant pattern matching

Function with dependent types.

```
Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l}
: listn (n + m) :=
 match l in listn n return listn (n + m) with
 | niln => l'
 | consn n' a y => consn (n' + m) a (concat n' y m l')
end.
```

match `l` **in** `listn n` **return** `listn (n + m)` **with**

`fun (n:nat) (l:listn n) => listn (n+m)`

Typing recursors

Dependently typed recursors

constructor $c\ a_1 : t_1 \cdots a_n : t_n$
then expression has to be type

$$\forall (b_1 : t'_1) \cdots (b_l : t'_l), c\ \ b_{i\,l} \cdots b_{i\,k}$$

Typing recursors

Dependently typed recursors

constructor $c\ a_1 : t_1 \cdots a_n : t_n$
then expression has to be type

$$\forall (b_1 : t'_1) \cdots (b_l : t'_l), c\ \ b_{i\,l} \cdots b_{i\,k}$$

- $j_1 = 1$ and $t'_1 = t_1$
- $j_{i+1} = j_i + 1$ and $t'_{j\,i} = t_1$ if not an instance

Typing recursors

Dependently typed recursors

constructor $c\ a_1 : t_1 \cdots a_n : t_n$
then expression has to be type

$$\forall (b_1 : t'_1) \cdots (b_l : t'_l), c\ b_{i\,l} \cdots b_{i\,k}$$

- $j_1 = 1$ and $t'_1 = t_1$
- $j_{i+1} = j_i + 1$ and $t'_{j\,i} = t_1$ if not an instance
- $t'_{j\,i} = t_i$ and $t'_{j\,i} = (fb_j)$

Typing recursors

Dependently typed recursors

constructor $c\ a_1 : t_1 \cdots a_n : t_n$
then expression has to be type

$$\forall (b_1 : t'_1) \cdots (b_l : t'_l), c\ \ b_{i\,l} \cdots b_{i\,k}$$

- $j_1 = 1$ and $t'_1 = t_1$
- $j_{i+1} = j_i + 1$ and $t'_{j\,i} = t_1$ if not an instance
- $t'_{j\,i} = t_i$ and $t'_{j\,i} = (fb_j)$
- if t_i is a function type $\forall (c_1 : \tau_1) \cdots (c_m : \tau_m), \tau$ where τ is an instance of the inductive type then

$$t_{j_i+1} = \forall (c_1 : \tau_1) \cdots (c_m : \tau_m), f(b_j, c_1, \cdots c_m)$$

Typing recursors

Dependently typed recursors: Recursor for the natural numbers

$$f : nat \to set$$

Typing recursors

Dependently typed recursors: Recursor for the natural numbers

$f : nat \rightarrow set$

First recursor is $f\ 0$

Typing recursors

Dependently typed recursors: Recursor for the natural numbers

$f : nat \to set$

First recursor is $f\ 0$

Second constructor has one argument of type `nat`

b_1 of type `nat`

b_2 of type $f\ b_1$

$\forall (b_1 : nat)(b_2 : (f\ b_1)), f(S\ b_1)$

Typing recursors

Dependently typed recursors: Recursor for the natural numbers

$f : nat \rightarrow set$

First recursor is $f\ 0$

Second constructor has one argument of type `nat`

b_1 of type `nat`

b_2 of type $f\ b_1$

$\forall (b_1 : nat)(b_2 : (f\ b_1)), f(S\ b_1)$

$$nat\_rec : \forall f : nat \rightarrow Set, f\ 0 \rightarrow (\forall n : nat, f\ n \rightarrow f\ (S\ n)) \rightarrow \forall n : nat, f\ n$$

same as slide 4

Typing recursors

Non-dependant recursion: example

```
Definition plus2 :=
  nat_rec
    (fun n:nat => nat -> nat)
    (fun p:nat => p)
    (fun (n':nat) (plus_n':nat -> nat) (p:nat) => S (plus_n' p)).

Definition mult2 :=
  nat_rec (fun n:nat => nat) 0 (fun p v:nat => S(S v)).


Definition even :=
nat_rec
  (fun n:nat => bool)
  true
  (fun (n':nat) (even': (fun _ : nat => bool) n')  => even' ).
```

Typing recursors

Non-dependant recursion: Binary trees

```
Inductive N_btree : Set :=
  N_leaf : N_btree |
  N_bnode : nat -> N_btree -> N_btree -> N_btree.
```

- first argument is `(f:N_btree -> Set)`

Typing recursors

Non-dependant recursion: Binary trees

```
Inductive N_btree : Set :=
  N_leaf : N_btree |
  N_bnode : nat -> N_btree -> N_btree -> N_btree.
```

- first argument is `(f:N_btree -> Set)`
- second is `(f N_leaf)`

Typing recursors

Non-dependant recursion: Binary trees

```
Inductive N_btree : Set :=
  N_leaf : N_btree |
  N_bnode : nat -> N_btree -> N_btree -> N_btree.
```

- first argument is `(f:N_btree -> Set)`
- second is `(f N_leaf)`
- 3rd argument

$$a_1 : \mathrm{nat}$$
$$a_2 : \mathrm{N\_btree}$$
$$a_3 : \mathrm{N\_btree}$$

Typing recursors

Non-dependant recursion: Binary trees

```
N_bnode : nat -> N_btree -> N_btree -> N_btree.
```

a_1 : nat a_2 : N_btree a_3 : N_btree

1 $j_1 = 1$ and b_1 must be type nat

Typing recursors

Non-dependant recursion: Binary trees

```
N_bnode : nat -> N_btree -> N_btree -> N_btree.
```

a_1 : nat a_2 : N_btree a_3 : N_btree

1. $j_1 = 1$ and b_1 must be type nat
2. $j_2 = 2$ must b_2 be type N_btree

   N_btree is the inductive type studied. So $j_3 = 4$ and b_3 must have type $(f\ \ b_2)$

Typing recursors

Non-dependant recursion: Binary trees

```
N_bnode : nat -> N_btree -> N_btree -> N_btree.
```

a_1 : nat a_2 : N_btree a_3 : N_btree

1. $j_1 = 1$ and b_1 must be type nat
2. $j_2 = 2$ must b_2 be type N_btree

   N_btree is the inductive type studied. So $j_3 = 4$ and b_3 must have type $(f\ \ b_2)$
3. b_4 must be N_btree

   So $b_5 = (f\ \ b_4)$

Typing recursors

Non-dependant recursion: Binary trees

The whole type is for the second constructor is:

$$\forall (b_1 : nat)(b_2) : N\_tree)(b_3 : f\ \ b_2)(b_4 : N\_tree)(b_5 : f\ \ b_4), f(N\_bnode\ b_1\ \ b_2\ \ b_3)$$

Typing recursors

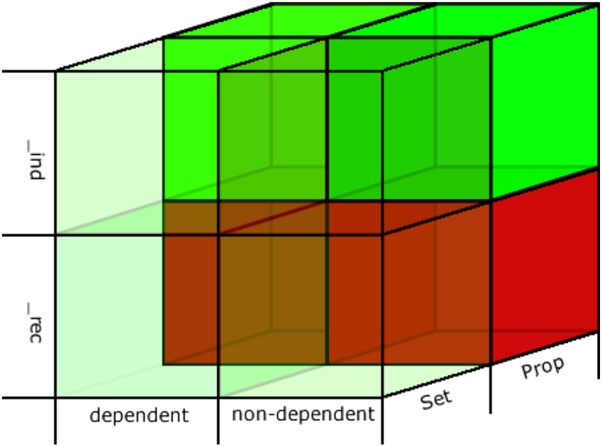
Non-dependant recursion: Binary trees

The whole type is for the second constructor is:

$$\forall (b_1 : nat)(b_2) : N\_tree)(b_3 : f\ \ b_2)(b_4 : N\_tree)(b_5 : f\ \ b_4), f(N\_bnode\ b_1\ \ b_2\ \ b_3)$$

```
N_btree_rec
     : forall f : N_btree -> Set,
       f N_leaf ->
       (forall (n : nat) (n0 : N_btree),
        f n0 -> forall n1 : N_btree, f n1 -> f (N_bnode n n0 n1)) ->
       forall n : N_btree, f n
```

Induction principles for predicates



Induction principles for predicates

- *Maximal induction*
- *Simplified induction principle*

Induction principles for predicates

- *Maximal induction*
  - Done by Wessel last week
- *Simplified induction principle*
  - Today

Induction principles for predicates

Definition

```
Inductive even : nat -> Prop :=
  | even0 : even 0
  | evenS : forall n, even n => even (S (S n)).


Scheme even_ind_max := Induction for even Sort Prop.

Check even_ind_max.
Check even_ind.
```

Induction principles for predicates

Proof irrelevance

```
even_ind_max
     : forall P : forall n : nat, even n -> Prop,
       P 0 even0 ->
       (forall (n : nat) (e : even n), P n e ->
           P (S (S n)) (evenS n e)) ->
       forall (n : nat) (e : even n), P n e
```

Induction principles for predicates

Proof irrelevance

```
even_ind_max
     : forall P : forall n : nat, even n -> Prop,
       P 0 even0 ->
       (forall (n : nat) (e : even n), P n e ->
           P (S (S n)) (evenS n e)) ->
       forall (n : nat) (e : even n), P n e
```

provability of a proposition, **not** the *proofs* of it,

```
even_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n
```

Induction principles for predicates

```
Definition even_plus: forall n, even n -> forall m, even m
                         -> even (n + m) :=
  fun n even_n m even_m =>
    even_ind (fun n' => even (n' + m))
      even_m
      (fun n' even_n' IH => evenS (n' + m) IH)
      n
      even_n.
```

The end