inductive types

logical verification

week 4

2004 09 29

subject

type theory $=$ typed λ-calculus $+$ inductive types

Curry-Howard-de Bruijn

logic	$\sim$	**type theory**
formula	$\sim$	type
proof	$\sim$	term
detour elimination	$\sim$	β-reduction
minimal logic	$\sim$	simply typed λ-calculus
intuitionistic logic	$\sim$	simply typed λ-calculus + inductive types
classical logic	$\sim$	$\ldots$ + exceptions

examples of inductive types

- booleans
- natural numbers
- integers
- pairs
- linear lists
- binary trees
- logical operations

inductive types

- types
- **recursive functions**
  - definition using pattern matching
  - ι-reduction = evaluation of recursive functions
- proof by cases
  proof by **induction**
  - induction principle

booleans: type

Coq definition:

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

booleans: recursive function

definition of negation:

```
Definition neg (b : bool) : bool :=
  match b with
    true => false
  | false => true
  end.
```

ι-reduction

$$\texttt{neg true} \twoheadrightarrow \texttt{false}$$

booleans: proof by cases

```
forall b : bool, neg (neg b) = b
```

tactics

- `elim b.`
- `simpl.`
- `reflexivity.`

booleans: induction principle

```
bool_ind :
  forall P : bool -> Prop,
    P true -> P false -> forall b : bool, P b
```

$$\forall P \in (\textsf{bool}\rightarrow\textsf{Prop}).\ \ P(\textsf{true}) \Rightarrow P(\textsf{false}) \Rightarrow\ \forall b \in \textsf{bool}.\, P(b)$$

natural numbers: type

Coq definition:

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat.
```

natural numbers: recursive function

definition of plus:

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

ι-reduction

plus (S O) (S O) $\twoheadrightarrow$ S (plus O (S O)) $\twoheadrightarrow$ S (S O)

natural numbers: proof by induction

```
forall n : nat, plus n O = n
```

tactics

- elim n.
- induction n.
- rewrite IHn.

natural numbers: induction principle

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

$$\forall P.\ \ P(0) \Rightarrow\ (\forall n \in \mathbb{N}.\ P(n) \Rightarrow P(n+1)) \Rightarrow\ \forall n \in \mathbb{N}. P(n)$$

lists: type

Coq definition:

```
Inductive natlist : Set :=
  nil : natlist
| cons : nat -> natlist -> natlist
```

the list $1, 2, 3, 4$ is encoded by

`cons 1 (cons 2 (cons 3 (cons 4 nil)))`

lists: recursive function

definition of append:

```
Fixpoint append (l k : natlist) {struct l} : natlist :=
  match l with
    nil => k
  | cons n l' => cons n (append l' k)
  end.
```

ι-reduction

```
append (cons 0 nil) nil   ->>   cons 0 (append nil nil)
                          ->>   cons 0 nil
```

lists: induction principle

```
natlist_ind :
  forall P : natlist -> Prop, P nil ->
    (forall (n : nat) (l : natlist), P l -> P (cons n l)) ->
      forall l : natlist, P l
```

truth: type

Coq definition:

```
Inductive True : Prop :=
  I : True.
```

falsity: type

Coq definition:

```
Inductive False : Prop :=
  .
```

falsity: induction principle

```
False_ind :
  forall P : Prop, False -> P
```

$$\dfrac{\begin{array}{c}\vdots \\ \bot\end{array}}{P}\ E\bot$$

Prop versus Set

```
S O          :      nat     :     Set
                                       :
                                          Type_0   :   Type_1   :   ...
                                       :
fun x:A => x   :   A -> A   :   Prop
```

types

```
   fun x : A => ...
forall x : A , ...

             A : Prop
             A : Set
             A : Type
```

Prop versus bool

I : True : Prop

true : bool : Set

inductive types: True and bool

true is not a type at all:

Curry-Howard-de Bruijn only for True and Prop

church natural numbers

encoding of natural numbers in untyped λ-calculus

$$
\begin{aligned}
0 &= \lambda f.\, \lambda x.\, x \\
1 &= \lambda f.\, \lambda x.\, f\, x \\
2 &= \lambda f.\, \lambda x.\, f\, (f\, x) \\
&\vdots \\
S &= \lambda n.\, \lambda f.\, \lambda x.\, f\, (n\, f\, x)
\end{aligned}
$$

church natural numbers can be typed

type A

$$
\begin{array}{lcl}
0 & = & \lambda f : A \to A.\ \lambda x : A.\ x \quad \mathbf{:} \quad (A \to A) \to (A \to A) \\
S & = & \lambda n : (A \to A) \to (A \to A).\ \lambda f : A \to A.\ \lambda x : A.\ f\,(n\,f\,x)
\end{array}
$$

type of church natural numbers

$$(A \to A) \to (A \to A)$$

inductive natural numbers

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat.
```

why inductive types built-in if we can define them?

- more efficient
- different reduction behavior

another function on lists: reverse

```
Fixpoint reverse (l : natlist) : natlist :=
  match l with
    nil => nil
  | cons n l' => append (reverse l') (cons n nil)
  end.
```

reverse is an involution

```
forall l : natlist, reverse (reverse l) = l
```