Lambda-Calculus and Type Theory

ISR 2024
Obergurgl, Austria

Herman Geuvers & Niels van der Weide

Radboud University Nijmegen, The Netherlands

Lecture 5

Inductive Types

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.
```

Coq produces:

```
bool_rec.
bool_rect.
bool_ind.
```

Booleans: elimination

```
Check bool_rec.

bool_rec
     : forall P : bool -> Set, P true -> P false
          -> forall b : bool, P b


Check bool_ind.

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

Booleans: elimination

```
bool_rec
     : forall P : bool -> Set, P true -> P false
          -> forall b : bool, P b
```

Special case: instantiate with `P := fun b :bool => A` with `A : Set`

`bool_rec P : A -> A -> bool -> A`

If `a1 : A` and `a2 : A`, then

`bool_rec P a1 a2 : bool -> A`

ι-reduction

$$\texttt{bool\_rec P a1 a2 true} \twoheadrightarrow \texttt{a1}$$

$$\texttt{bool\_rec P a1 a2 false} \twoheadrightarrow \texttt{a2}$$

But usually we program functions using pattern matching

Booleans: recursive functions using pattern matching

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

- `destruct b.`
- `simpl.`
- `reflexivity.`

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) ↠ S (plus O (S O)) ↠ S (S O)
```

Natural numbers: proof by induction

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

tactics

- destruct 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  :  Type  :  ..
                              :
fun x:A => x  :  A -> A  :  Prop
```

Prop versus Set

```
S O        :    nat   :   Set
                                 :
                                    Type₀  :  Type₁  :
                                 :
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

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

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
- large eliminations

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
```

Coq only accepts well-defined inductive types

You cannot write

```
Inductive Lambda : Set :=
  var : nat -> Lambda
| app : Lambda -> Lambda -> Lambda
| abs : (Lambda -> Lambda) -> Lambda.
```

The type X you are defining should occur strictly positive in all σ_i, if

$$\text{constr} : \sigma_1 \to \ldots \to \sigma_n \to X$$

is a constructor declaration.

Coq only accepts well-defined recursive functions

Recursive call must be to a structurally smaller argument.

You cannot write

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

You cannot write

```
Fixpoint g (n : nat) {struct n} : nat :=
  match n with
    O => 1
  | S p => g (p mod 2)
  end.
```