

logical verification lecture 8
2011 04 29
dependent types

	logic		lambda		Coq
	prop1	~	$\lambda \rightarrow$		
	intuitionistic				ind pred
	classical				
	pred1	~	λP		ind data
	prop2	~	$\lambda 2$		prog ext

dependent types

- dependent types in Coq
- λP : lambda calculus with dependent types
- (a fragment of) λP corresponds to pred1
- λP as logical framework

dependent types in Coq: example

inductive definition of dependent natlists:

```
Inductive natlist_dep : nat -> Set :=  
  | nil_dep   : natlist_dep 0  
  | cons_dep  : forall n : nat,  
    nat -> natlist_dep n -> natlist_dep (S n).
```

the type of the type constructor:

```
natlist_dep : nat -> Set : Type
```

dependent types in Coq: example

inductive definition of dependent zeroes:

```

Fixpoint zeroes_dep (n:nat) : natlist_dep n :=
  match n
  return natlist_dep n with
  | 0 => nil_dep
  | S p => cons_dep p 0 (zeroes_dep p)
  end.

```

the type of the function:

```

zeroes_dep : forall n : nat, natlist_dep n

```

syntax of lambda P

sorts $*, \square$	lambda abstraction $\lambda x:A. N$
variables x, y, z, \dots	dependent product $\prod x:A. N$
	function application $F N$

$\prod x:A. B$ can be written as $A \rightarrow B$ if $x \notin B$

typing system

for every kind of term there is a rule:

- axiom rule (for $*$ and \square)
- variable rule
- product rule
- abstraction rule
- application rule

and two more rules:

- weakening rule
- conversion rule

typing system: application rule

new version—the product type may be dependent:

$$\frac{\Gamma \vdash F : \prod x:A. B \quad \Gamma \vdash N : A}{\Gamma \vdash F N : B[x := N]}$$

old version—the special case $\lambda \rightarrow$:

$$\frac{\Gamma \vdash F : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash F N : B}$$

typing system: abstraction rule

Curry-Howard-De Bruijn isomorphism

new version—we need to have that the product type is ok :

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x:A. B : * / \square}{\Gamma \vdash \lambda x:A. M : \Pi x:A. B}$$

(a fragment of) λP corresponds to minimal pred1

old version—the function type is itself not typed:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x:A. M : A \rightarrow B}$$

minimal pred1: only implication and for all

minimal pred1: introduction rules

formulas of minimal pred1 compared with prop1:

	prop1	pred1
a	+	+
$A \rightarrow B$	+	+
$\forall x. B$	-	+

$$\frac{B}{A \rightarrow B} I_{\rightarrow} \quad \frac{B}{\forall x. B} I_{\forall}$$

minimal pred1: elimination rules

$$\frac{A \rightarrow B \quad A}{B} E \rightarrow \quad \frac{\forall x. B}{B[x := M]} E\forall$$

minimal pred1 in λP : formulas

logic	λP	Coq
$P(M_1, \dots, M_n)$	$PM_1 \dots M_n : \star$	$P M_1 \dots M_n : \text{Prop}$
$A \rightarrow B$	$A \rightarrow B : \star$	$A \rightarrow B : \text{Prop}$
$\forall x. B$	$\Pi x : \text{Terms}. B : \star$	$\text{forall } x : \text{Terms}, B : \text{Prop}$

with $P : \Pi x_1 : \text{Terms}. \dots \Pi x_n : \text{Terms}. \star$
 or $P : \text{Terms} \rightarrow \dots \rightarrow \text{Terms} \rightarrow \star$

formulas use (algebraic) terms

minimal pred1 in λP : algebraic terms

logic	λP	Coq
	$\text{Terms} : \star$	$\text{Terms} : \text{Set}$
x	$x : \text{Terms}$	$x : \text{Terms}x$
$f(M_1, \dots, M_n)$	$f M_1 \dots M_n : \text{Terms}$	$f M_1 \dots M_n : \text{Terms}$

with $f : \Pi x_1 : \text{Terms}. \dots \Pi x_n : \text{Terms}. \text{Terms}$
 or $f : \text{Terms} \rightarrow \dots \rightarrow \text{Terms} \rightarrow \text{Terms}$

Curry-Howard-De Bruijn isomorphism

introduction \rightarrow		abstraction
introduction \forall		
elimination \rightarrow		application
elimination \forall		

introduction rules correspond to abstraction

$$\frac{B}{A \rightarrow B} I \rightarrow \quad \frac{B}{\forall x. B} I\forall$$

correspond to

$$\frac{M : B}{\lambda x : A. M : \Pi x : A. B}$$

in Coq: intro

elimination rules correspond to application

$$\frac{A \rightarrow B \quad A}{B} E \rightarrow \quad \frac{\forall x. B}{B[x := M]} E\forall$$

correspond to

$$\frac{F : \Pi x : A. B \quad M : A}{F M : B[x := M]}$$

in Coq: apply

example1: λP and minimal pred1

formula:

$$(\forall x. P(x)) \rightarrow P(M)$$

type:

$$(\Pi x : \text{Terms} . P x) \rightarrow (P M)$$

inhabitant:

$$\lambda u : (\Pi x : \text{Terms} . P x) . u M$$

example2: λP and minimal pred1

formula:

$$(\forall x. P(x) \rightarrow Q(x)) \rightarrow (\forall y. P(y)) \\ \rightarrow (\forall z. Q(z))$$

type:

$$(\Pi x : \text{Terms} . P x \rightarrow Q x) \rightarrow (\Pi y : \text{Terms} . P y) \\ \rightarrow (\Pi z : \text{Terms} . Q z)$$

inhabitant:

$$\lambda u : (\Pi x : \text{Terms} . P x \rightarrow Q x) . \lambda v : (\Pi y : \text{Terms} . P y) . \\ \lambda z : \text{Terms} . (u z) (v z)$$

example one: λP and minimal pred1

formula:

$(A \rightarrow \forall x. P(x)) \rightarrow \forall y. A \rightarrow P(y)$

type:

$(A \rightarrow \Pi x:\text{Terms} . P x) \rightarrow \Pi y:\text{Terms} . A \rightarrow (P y)$

inhabitant:

$\lambda u : (A \rightarrow \Pi x:\text{Terms} . P x). \lambda y : \text{Terms} . \lambda a : A. u a y$

example two: λP and minimal pred1

formula:

$A \rightarrow \forall x. A$

type:

$A \rightarrow \Pi x:\text{Terms} . A$

inhabitant:

$\lambda a : A. \lambda x : \text{Terms} . a$

logical framework

prop1 in λP

we can define logics in λP

example in the practical work: prop1 in λP

prop : Set
imp : prop \rightarrow prop \rightarrow prop
T : prop \rightarrow Prop

intro : $\Pi p:\text{prop} . \Pi q:\text{prop} . (T p \rightarrow T q) \rightarrow T (\text{imp } p q)$
elim : $\Pi p:\text{prop} . \Pi q:\text{prop} . T (\text{imp } p q) \rightarrow T p \rightarrow T q$

prop1 in λP

where are dependent types needed?

to see that the types of the proof rules are ok

prop1 in λP : example

formula:

```
forall p : prop, T (p => p)
```

inhabitant:

```
fun p : prop => imp_introduction p p (fun u : T p => u)
```

finding inhabitants: non-dependent

```
Definition prop1 := (*! term *)
(* fun (x : ?) (y : ?) (z : ?) => x z (y z). *)
```

finding inhabitants: dependent

```
Definition pred1 := (*! term *)
(* fun (l : Set -> Set) (A : ?) (B : ?)
      (f : l A -> l B) (x : ?)
      => f x. *)
```

dependent types in programming: further reading

- [Epigram](#) (Conor McBride, James McKinna)
- [Dependent ML](#) (Frank Pfenning and Hongwei Xi)
- [Dependent types in Haskell](#)