

logical verification lecture 7

2011 04 26

dependent types

overview: past present future

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

zeroes

definition in Coq:

```
Fixpoint zeroes (n:nat) : natlist :=
match n with
| 0 => nil
| S p => cons 0 (zeroes p)
end.
```

using Eval compute we find:

```
(zeroes 0) = nil : natlist
(zeroes 1) = cons 0 nil : natlist
(zeroes 2) = cons 0 (cons 0 nil) : natlist
```

the type of zeroes

the type of the function:

`zeroes : nat -> natlist`

use of the application rule (as before):

$$\frac{\text{zeroes} : \text{nat} \rightarrow \text{natlist} \quad 3 : \text{nat}}{\text{zeroes } 3}$$

the type of zeroes: more information

(zeroes 0) is a natlist of length 0

(zeroes 1) is a natlist of length 1

(zeroes 2) is a natlist of length 2

(zeroes n) is a natlist of length n

towards dependent types

types may depend on terms

type constructor for dependent lists: definition

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

type constructor for dependent lists: type

the type of the type constructor:

`natlist_dep : nat -> Set : Type`

use of the application rule (as before):

$$\frac{\textit{natlistdep} : \textit{nat} \rightarrow \textit{Set} \quad 3 : \textit{nat}}{\textit{natlistdep} 3 : \textit{Set}}$$

dependent zeroes: definition

```
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.
```

using Eval compute we find:

```
(zeroes_dep 0) = nil_dep : natlist_dep 0
(zeroes_dep 1) = cons_dep 0 0 nil_dep : natlist_dep 1
(zeroes_dep 2) = cons_dep 1 0 (cons_dep 0 0 nil_dep)
                : natlist_dep 2
```

dependent zeroes: type

the type of the function:

`zeroes_dep : forall n : nat, natlist_dep n`

(new) use of the application rule:

$$\frac{\text{zeroesdep} : \text{forall } n : \text{nat}, \text{natlistdep } n \quad 3 : \text{nat}}{\text{zeroesdep } 3 : \text{natlistdep } 3}$$

function types: non-dependent and dependent

non-dependent function type (as before):

```
nat -> natlist
```

dependent function type (new):

```
forall n : nat, natlist_dep n
```

uniform presentation for both:

```
forall n:nat, natlist
```

```
forall n : nat, natlist_dep n
```

length of dependent lists: definitions

an easy definition:

```
Definition length_dep (n : nat) (l : natlist_dep n)
  := n.
```

another definition:

```
Definition length_dep_b (n : nat) (l : natlist_dep n)
  : nat :=
  match l with
  | nil_dep => 0
  | cons_dep n h t => S n
  end.
```

append of dependent lists: definition

```
Fixpoint append_dep (n : nat) (k : natlist_dep n)
  (m : nat)(l : natlist_dep m)
  {struct k}
  : natlist_dep (n + m) :=
  match k in (natlist_dep n)
  return (natlist_dep (n + m)) with
  | nil_dep => l
  | cons_dep p h t =>
      cons_dep (p + m) h (append_dep p t m l)
  end.
```

we need to compute inside types

dependent types in Coq

datatype and (data)type constructor in Set:

```
natlist : Set
natlist_dep : nat -> Set
```

proposition and predicate ('proposition constructor') in Prop:

```
True : Prop
even : nat -> Prop
```

examples

```
O : nat : Set : Type
fun x:nat => x : nat -> nat : Set : Type
nil : natlist : Set : Type
nildep : natlistdep O : Set : Type
        natlistdep : nat -> Set : Type
```

lambda calculus with dependent types

- extension of simply typed lambda calculus
- terms and types are no longer separate worlds
cf. `natlistdep n` in Coq
- one constructor Π for (dependent) product
special case: the \rightarrow we had before
cf. `forall` in Coq

syntax of lambda P

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

product type $\Pi x : A. B$ can be written as $A \rightarrow B$ if $x \notin B$

lambda P versus Coq

*	~	Set
*	~	Prop
□	~	Type
x	~	x
$F\ N$	~	F N
$\lambda x:A. M$	~	fun x:A ⇒ M
$\prod x:A. M$	~	forall x:A, M

pseudo-terms of lambdaP

expressions we can form according to the grammar

examples:

\square

*

$\square \square$

$\lambda n : \text{nat}. \lambda x : n. x$

$\lambda n : \text{nat}. n n$

terms

all pseudo-terms that can be typed

examples:

□

*

$\lambda n : \text{nat}. n$

$\lambda f : \text{nat} \rightarrow \text{nat}. \lambda n : \text{nat}. f n$

typing system

selects the terms from the pseudo-terms

used to derive judgements of the form

$\Gamma \vdash M : N$

M is a term if we can derive

$\Gamma \vdash M : A$ or $\Gamma \vdash N : M$

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

we consider a few rules

we do not make complete derivations in lambdaP in the exam

typing system: application rule

new version—the product type may be dependent:

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

old version—the special case non-dependent function type λ^{\rightarrow} :

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

typing system: application rule

new version—the product type may be dependent:

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

compare with application in Coq:

$$\frac{\text{zeroesdep} : \text{forall} n : \text{nat}, \text{natlistdep } n \quad 3 : \text{nat}}{\text{zeroesdep } 3 : \text{natlistdep } 3}$$

typing system: abstraction rule

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

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

old version—the function type is itself not typed:

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

typing system: conversion rule

new rule—we may compute inside ‘types’:

$$\frac{\Gamma \vdash A : B' \quad \Gamma \vdash B : \star/\square}{\Gamma \vdash A : B} \quad \text{with } B =_{\beta} B'$$

use of this rule:

Poincaré principle

computational equalities do not require proof

cf in Coq