

logical verification lecture 9

2011 05 03

second-order propositional logic

overview: past present future

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

prop2 and polymorphic types

- prop2
- polymorphic types in Coq
- $\lambda 2$: lambda calculus with polymorphic types
- correspondence prop2 and $\lambda 2$

formulas of prop1 (already seen)

$a \ b \ c \ p \ q$

$A \rightarrow B$

\perp

\top

$A \wedge B$

$A \vee B$

formulas of pred1 (already seen)

(using terms)

$a(\dots) b(\dots) c(\dots) p(\dots) q(\dots)$

$A \rightarrow B$

$\forall x. A$

\perp

\top

$A \wedge B$

$A \vee B$

$\exists x. A$

formulas of prop2 (new)

$a b c p q$

$A \rightarrow B$

$\forall a. A$

\perp

\top

$A \wedge B$

$A \vee B$

$\exists a. A$

examples

in prop1:

$$a \rightarrow a$$

in pred1:

$$\forall x. a(x) \rightarrow a(x)$$

in prop2:

$$\forall a. a \rightarrow a$$

higher-order

first order:

object

second order:

set of first-order objects

predicate on objects

third order:

set of second-order objects

predicate on predicates on objects

higher-order logic

first-order:

quantification over variables of order 1

$$a \rightarrow a$$

$$\forall x. a(x) \rightarrow a(x)$$

second-order:

quantification over variables of order 2

$$\forall a. a \rightarrow a$$

$$\forall a. \forall x. a(x) \rightarrow a(x)$$

$$\forall f. \forall x. a(f(x)) \rightarrow a(f(x))$$

third-order:

quantification over variables of order 3

$$\forall b. \forall f. b(f) \rightarrow \forall x. a(f(x))$$

second-order predicate logic: example

induction principle for natural numbers

$$\forall a(a(0) \rightarrow (\forall m. a(m) \rightarrow a(S(m)))) \rightarrow \forall n. a(n)$$

| | |
|-----|--------------------|
| m | 1st order variable |
| n | 1st order variable |
| 0 | 1st order constant |
| a | 2nd order variable |
| S | 2nd order constant |

second-order predicate logic: example

there is a sorting function

$\exists f : \text{natlist} \rightarrow \text{natlist}. \forall l : \text{natlist}. \text{sorted}(f(l)) \wedge \text{permutation}(l, f(l))$

| | |
|----------------------|--------------------|
| f | 2nd order variable |
| l | 1st order variable |
| sorted | 2nd order constant |
| permutation | 2nd order constant |

examples

| | |
|------------------------------|-----------------------------------------------|
| prop2 | pred2 |
| $\forall a. a \rightarrow a$ | $\forall p. \forall x. p(x) \rightarrow p(x)$ |
| prop1 | pred1 |
| $a \rightarrow a$ | $\forall x. p(x) \rightarrow p(x)$ |

proof rules for prop2

introduction rules

$I\top$

$I[x] \rightarrow$

$I\wedge$

$I\vee, Ir\vee$

$I\forall$

$I\exists$

elimination rules

$E\perp$

$E\rightarrow$

$EI\wedge, Er\wedge$

$E\vee$

$E\forall$

$E\exists$

universal quantification for prop2

\forall introduction:

$$\frac{A}{\forall a. A} \quad I\forall$$

variable condition: a not free in any open assumption

\forall elimination:

$$\frac{\forall a. A}{A[a := B]} \quad E\forall$$

existential quantification for prop2

\exists introduction:

$$\frac{A[a := B]}{\exists a. A} \text{I}\exists$$

\exists elimination:

$$\frac{\exists a. A \quad \forall a. A \rightarrow B}{B} \text{E}\exists$$

variable condition: a not free in B

examples

- $(\forall b. b) \rightarrow a$
- $a \rightarrow \forall b. (b \rightarrow a)$
- $a \rightarrow \forall b. ((a \rightarrow b) \rightarrow b)$
- $(\exists b. a) \rightarrow a$
- $\exists b((a \rightarrow b) \vee (b \rightarrow a))$

non-examples

- $a \rightarrow (\forall a. a)$
- $p(x) \rightarrow (\forall x. p(x))$
- $(\exists a. a) \rightarrow a$

impredicativity

the 'meaning' of $\forall a. B$

depends on the meaning of all formulas $B[a := A]$

(this includes $B[a := \forall a. B]$)

impredicativity

compare with Russell's paradox in set theory:

$$R = \{x \mid x \notin x\}$$

$$R \in R?$$

$$\Pi x: \star . x \rightarrow \text{bool}$$

impredicativity in Coq

Prop is impredicative

```
forall x : Prop, x          : Prop
forall x : Prop, x -> True : Prop
```

Set is not impredicative

```
forall x : Set, x          : Type
forall x : Set, x -> bool : Type
```

logical connectives in Coq

\rightarrow in the type theory

\forall in the type theory

\perp inductive type

\wedge inductive type

\vee inductive type

\exists inductive type

but another choice was possible: impredicative definitions

False: inductive definition in Coq

introduction rule: via the constructors (here: none)

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

elimination rule: via the induction principle

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

False: impredicative definition in prop2

$\perp := \forall a. a$

then we can prove the elimination rules

and: inductive definition in Coq

introduction rule: via the constructors

```
Inductive and (a b : Prop) : Prop :=  
  conj : a -> b -> (and a b).
```

elimination rule: via the induction principle

```
and_ind :  
  forall a b c : Prop,  
  (a -> b -> c) -> a /\ b -> c
```

and: impredicative definition in prop2

$$a \wedge b := \forall c. (a \rightarrow b \rightarrow c) \rightarrow c$$

then we can prove the elimination rules

or: inductive definition in Coq

introduction rule: via the constructors

```
Inductive or (a b : Prop) : Prop :=  
  | or_introl : a -> (or a b)  
  | or_intror : b -> (or a b) .
```

elimination rule: via the induction principle

```
or_ind :  
  forall a b c : Prop,  
  (a -> c) -> (b -> c) -> a \/ b -> c
```

or: impredicative definition in prop2

$$a \vee b := \forall c. (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow c$$

then we can prove the elimination rules

classical prop2

all formulas and proof rules of intuitionistic prop2
plus a classical axiom

- $\forall a. a \vee \neg a$
- $\forall a. \neg\neg a \rightarrow a$
- $\forall a. \forall b. ((a \rightarrow b) \rightarrow a) \rightarrow a$

example of a tautology:

$$\forall a. \forall b. ((a \rightarrow b) \vee (b \rightarrow a))$$

expressivity of classical prop2

- $\forall a. B$ is equivalent to $B[a := \perp] \wedge B[a := \top]$
- $\exists a. B$ is equivalent to $B[a := \perp] \vee B[a := \top]$

so classical prop1 is as expressive as classical prop2

natlists and boollists in Coq

```
Inductive natlist : Set :=  
  natnil : natlist  
| natcons : nat -> natlist -> natlist.
```

```
Inductive boollist : Set :=  
  boolnil : boollist  
| boolcons : bool -> boollist -> boollist.
```

polymorphic lists in Coq

definition:

```
Inductive polylist (A:Set) : Set :=  
  polynil   : (polylist A)  
| polycons  : A -> (polylist A) -> (polylist A).
```

types (use Check):

```
polylist : Set -> Set  
polynil   : forall A : Set, polylist A  
polycons  : forall A : Set,  
            A -> polylist A -> polylist A
```

instantiation by application

```
polylist nat : Set
polynil nat  : polylist nat
polycons nat : nat -> polylist nat -> polylist nat
```

example of the use of polymorphic lists

```
polycons nat 2 (polycons nat 1 (polynil nat))
```

```
polycons bool true  
  (polycons bool false (polynil bool))
```

slightly more efficient notation

Notation `ni := (polynil _)`.

Notation `co := (polycons _)`.

then we can write the following:

`Check (co 2 (co 1 ni))`.

`Check (co true (co false ni))`.

function on natlists in Coq

```
Fixpoint natlength (l:natlist) {struct l} : nat
:=
  match l with
  | natnil => 0
  | natcons h t => S (natlength t)
  end.
```

function on polymorphic lists in Coq

```
Fixpoint polylength
  (A:Set)(l:(polylist A)){struct l}
  : nat :=
  match l with
  | polynil => 0
  | polycons h t => S (polylength A t)
  end.
```

dependent types versus polymorphism

- **dependent types**
types may depend on terms
natlistdep n
- **polymorphic types**
terms may depend on types
polyid nat

dependent types versus polymorphism

- **dependent types**

functions and quantifications over elements of a type

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

$\prod x : \text{nat} . \text{natlistdep } x$

- **polymorphic types**

functions and quantification over types

$\lambda a : * . \lambda x : a . x$

$\prod a : * . a \rightarrow a$

(variety of) further reading

- about functional programming