overview

pred1

Coq

schema

pred1

Coq

first-order predicate logic (pred1)

predicate:
atomic formula is built from a predicate and zero one or more terms

first-order:
no quantification over formulas or predicates

first-order predicate logic (pred1)

syntax:

- terms (new compared to prop1)
- formulas
- judgements

proof rules:

- introduction rules
- elimination rules

terms: example

domain:	$\mathsf{nat} = \{0, 1, 2, 3, \ldots\}$
functions:	addition, division
terms:	3, 5, $3 + 5$

terms: definition

- a variable is a term
  x in Terms
- applying a function symbol to the right number of terms yields a term
  if $M_1, \ldots, M_n$ in Terms then
  $f(M_1, \ldots, M_n)$ in Terms

this is singly-sorted; we could work in a multi-sorted setting

predicates: example

$P(n)$ meaning n is a prime number

$E(n)$ meaning n is even

$Q(n, m)$ meaning n divides m

formulas: definition for prop1 (already seen)

$a\ b\ c\ p\ q$

$A \to B$

$\bot$

$\top$

$A \wedge B$

$A \vee B$

formulas: definition for pred1

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

$A \to B$

$\forall x.\, B$

$\bot$

$\top$

$A \wedge B$

$A \vee B$

$\exists x.\, A$

formulas: examples

in prop1:

$a \to a$

in pred1:

$\forall x.\, a(x) \to a(x)$

terminology: first-order

first order:
object

second order:
set of first-order objects
predicate on objects

first-order logic:
quantification over variables of order 1
$a \to a$
$\forall x.\, a(x) \to a(x)$

proof rules for universal quantification

$\forall$ introduction:

$$\frac{A}{\forall x.\, A}\ I\forall$$

variable condition: x not free in any open assumption

$\forall$ elimination:

$$\frac{\forall x.\, A}{A[x := M]}\ E\forall$$

forall: examples

tautology:
$(\forall x.\, P(x) \to Q(x)) \to (\forall x.\, P(x)) \to \forall y.\, Q(y)$

non-example: without the side condition we can prove
$\forall x.\, (P(x) \to \forall x.\, P(x))$

proof rules for existential quantification

$\exists$ introduction:

$$\frac{A[x := M]}{\exists x.\, A}\ I\exists$$

$\exists$ elimination:

$$\frac{\exists x.\, A \qquad \forall x.\, A \to B}{B}\ E\exists$$

variable condition: x not free in B

exists: examples

a tautology:
$(\exists x.\, P(x) \vee Q(x)) \rightarrow (\exists x.\, P(x)) \vee (\exists x.\, Q(x))$

non-example: without the side-condition we can prove
$\forall x.\, ((\exists x.\, P(x)) \rightarrow P(x))$

proof rules of pred1

introduction rules	elimination rules
$I\top$	
	$E\bot$
$I[x] \to$	$E \to$
$I\wedge$	$El\wedge$, $Er\wedge$
$Il\vee$, $Ir\vee$	$E\vee$
$I\forall$	$E\forall$
$I\exists$	$E\exists$

remark: empty domains

$I\exists$ is not valid if the domain is empty

$$\dfrac{\dfrac{}{\top}\, I\top}{\exists x.\top}\, I\exists$$

schema

pred1

Coq

terms in Coq: examples

in the exercise on the drinker:

```
Parameter D:Set.
Parameter d:D.
```

natural numbers (defined inductively):

```
nat : Set.
O : nat.
S O : nat.
S (S O) : nat.
```

predicates in Coq: examples

unary predicate

```
drinks : D -> Prop
```

unary predicate

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

binary predicate

```
le : nat -> nat -> Prop
```

formulas in Coq: examples

```
even 0 : Prop
even 1 : Prop

le 0 0 : Prop
le 1 0 : Prop

even 0 /\ le 0 0 : Prop
even 0 \/ le 1 0 : Prop

forall x:nat, even x : Prop
exists x:nat, even x : Prop
```

formulas: paper and Coq

$a(\ldots)$	`a ...`
$A \to B$	`A -> B`
$\forall x.\, B$	`forall x:Terms, B`
$\bot$	`False`
$\top$	`True`
$A \wedge B$	`A /\ B`
$A \vee B$	`A \/ B`
$\exists x.\, A$	`exists x:Terms, B`
	`{x:Terms | B }`

proof rules: paper and Coq

$I[x] \to \quad I\forall$	intro
$E \to \quad E\forall$	apply
$E\bot \quad El \wedge \quad Er \wedge \quad E \vee \quad E\exists$	elim
$I\wedge$	split
$Il\vee$	left
$Ir\vee$	right
$I\exists$	exists
$I\top$	exact I

overview

towards dependent types

lambda-calculus with dependent types

logical frameworks

from arrow to P

we have seen the extension from prop1 to pred1

we consider the extension from $\lambda \rightarrow$ to λP

(a fragment of) λP corresponds to minimal pred1

overview

towards dependent types

lambda-calculus with dependent types

logical frameworks

the three universes of Coq

`Set`
intuitively the universe of sets or datatypes
we have seen the declaration `Parameter Terms : Set.`

`Prop`
intuitively the universe of propositions or types
we have seen the declaration `Parameter A : Prop.`

`Type`
intuitively the universe of classes (sets of sets)

typing the universes

in Coq everything is typed; use `Check` to see the type

`Set : Type`

`Prop : Type`

`Type : Type`

the latter seems to lead to paradoxes
but in fact it is an infinite hierarchy

universes in type theory

for both `Set` and `Prop` we have $*$

for `Type` we have $\Box$

typing relation (rules follow): $* : \Box$

(there is no infinite hierarchy here)

simple types in lambdaP

in $\lambda \rightarrow$ types are inductively defined by means of a grammar

in λP "is a type" is derived in the typing system

assuming $\texttt{nat} : *$, that is, "$\texttt{nat}$ is a type"

we derive $\texttt{nat} \rightarrow \texttt{nat} : *$, that is, "$\texttt{nat} \rightarrow \texttt{nat}$ is a type"

dependent type: example

define for every n : nat a set S_n as follows:
$S_n := \{0, n, 2n, 3n, \ldots\}$ so
$S_0 = \{0\}$
$S_1 = \{0, 1, 2, 3, \ldots\}$
$S_2 = \{0, 2, 4, 6, \ldots\}$
$S_3 = \{0, 3, 6, 9, \ldots\}$

for every n : nat we have $S_n : *$ (in `Set`)
S_n is a type depending on a term

λn : nat. S_n : nat $\to *$ is a type constructor

dependent type: example

define for every n : nat a proposition P_n as follows:
$P_n := n$ is a prime number so
$P_0 = 0$ is a prime number
$P_1 = 1$ is a prime number
$P_2 = 2$ is a prime number
$P_3 = 3$ is a prime number

for every n : nat we have $P_n : *$ (in `Prop`)
P_n is a formula (type) depending on a term

λn : nat. P_n : nat $\to *$ is a type constructor

dependent type: example (programming)

define for every n : nat a set $\texttt{list}_n$ as follows:
$\texttt{list}_n$ are the natlists consisting of n elements so
$\texttt{list}_0$ is the type of the empty list (of natural numbers)
$\texttt{list}_1$ is the type of the lists of length 1
$\texttt{list}_2$ is the type of the lists of length 2

consider a function ones as follows:
$\text{ones}(0) = []$
$\text{ones}(1) = [1]$
$\text{ones}(2) = [1, 1]$
$\text{ones}(3) = [1, 1, 1]$
$\vdots$

consider the types

dependent type: example (programming)

define for every n : nat a set $\texttt{list}_n$ as follows:
$\texttt{list}_n$ are the natlists consisting of n elements so
$\texttt{list}_0$ is the type of the empty list (of natural numbers)
$\texttt{list}_1$ is the type of the lists of length 1
$\texttt{list}_2$ is the type of the lists of length 2

consider a function ones as follows:
$\text{ones}(0) = [] : \text{list}_0$
$\text{ones}(1) = [1] : \text{list}_1$
$\text{ones}(2) = [1, 1] : \text{list}_2$
$\text{ones}(3) = [1, 1, 1] : \text{list}_3$
$\vdots$

consider the types

example: application with instantiation of type

$$\frac{\mathsf{ones} : \Pi n : \mathsf{nat}.\, \mathsf{list}_n \qquad 0 : \mathsf{nat}}{\mathsf{ones}\, 0 : \mathsf{list}_0}$$

$$\frac{\mathsf{ones} : \Pi n : \mathsf{nat}.\, \mathsf{list}_n \qquad 1 : \mathsf{nat}}{\mathsf{ones}\, 1 : \mathsf{list}_1}$$

$$\frac{\mathsf{ones} : \Pi n : \mathsf{nat}.\, \mathsf{list}_n \qquad 2 : \mathsf{nat}}{\mathsf{ones}\, 2 : \mathsf{list}_2}$$

lambda calculus with dependent types

extension of simply typed lambda calculus

terms and types are no longer separate worlds

one constructor Π for (dependent) product

we get $\rightarrow$ back as a special case of Π

overview

towards dependent types

lambda-calculus with dependent types

logical frameworks

syntax of lambda P

sorts	lambda abstraction
$*$, $\Box$	$\lambda x : A.\ N$
variables	dependent product
$x, y, z, \ldots$	$\Pi x{:}A\,.\,N$
	function application
	$F\ N$

product type $\Pi x{:}A\,.\,B$ generalizes $A \to B$
it can be written as $A \to B$ if $x \notin B$

lambda P and Coq

$*$	$\sim$	Set
$*$	$\sim$	Prop
$\Box$	$\sim$	Type
x	$\sim$	x
$F\ N$	$\sim$	F N
$\lambda x : A.\ M$	$\sim$	fun x:A $\Rightarrow$ M
$\Pi x{:}A\ .\ M$	$\sim$	forall x:A, M

pseudo-terms of lambdaP

expressions we can form according to the grammar

examples:

$\Box$

$*$

$\Box\,\Box$

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

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

terms

all pseudo-terms that can be typed

examples:

$\Box$

$*$

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

$\lambda f : \mathsf{nat} \to \mathsf{nat}.\ \lambda n : \mathsf{nat}.\ f\ n$

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 $\Box$)
- variable rule
- product rule
- abstraction rule
- application rule

and two more rules:

- weakening rule
- conversion rule

typing system

we consider a few rules

typing system: application rule

new version–the product type may be dependent:

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

old version–the special case non-dependent function type $\lambda^{\rightarrow}$:

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

typing system: abstraction rule

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

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

old version–the function type is itself not typed:

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

typing system: conversion rule

new rule–we may compute inside 'types':

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

use of this rule:
Poincaré principle
computational equalities do not require proof

cf in Coq

overview

towards dependent types

lambda-calculus with dependent types

logical frameworks

logical framework

we can define logics in λP

example in the practical work: prop1 in λP

prop1 in λP

$$
\begin{array}{rcl}
\mathsf{prop} & : & \mathsf{Set} \\
\mathsf{imp} & : & \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop} \\
\mathsf{T} & : & \mathsf{prop} \to \mathsf{Prop} \\
 & & \\
\mathsf{intro} & : & \Pi p{:}\mathsf{prop}\,.\,\Pi q{:}\mathsf{prop}\,.\,(\mathsf{T}\,p \to \mathsf{T}\,q) \to \mathsf{T}\,(\mathsf{imp}\,p\,q) \\
\mathsf{elim} & : & \Pi p{:}\mathsf{prop}\,.\,\Pi q{:}\mathsf{prop}\,.\,\mathsf{T}\,(\mathsf{imp}\,p\,q) \to \mathsf{T}\,p \to \mathsf{T}\,q
\end{array}
$$

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)
```

typing system for lambdaP

$$\frac{}{\vdash * : \Box}\ \textit{start}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \textit{var}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ \textit{weak}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x{:}A \,.\, B : s}\ \textit{prod}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x{:}A \,.\, B : s}{\Gamma \vdash \lambda x : A.\, M : \Pi x{:}A \,.\, B}\ \textit{abs}$$

$$\frac{\Gamma \vdash F : \Pi x{:}A \,.\, B \qquad \Gamma \vdash M : A}{\Gamma \vdash (F\ M) : B[x := M]}\ \textit{app}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}\ \textit{conv} \qquad \text{with } B =_\beta B'$$

examples

$\mathsf{nat} : *, n : \mathsf{nat} \vdash n : \mathsf{nat}$

$\mathsf{nat} : * \vdash \Pi n : \mathsf{nat}.\, \mathsf{nat} : *$

$\mathsf{nat} : * \vdash \lambda n : \mathsf{nat}.\, n : \Pi n : \mathsf{nat}.\, \mathsf{nat}$

decidability issues

- type inhabitation problem (TIP)
  $\Gamma \vdash ? : A$
  undecidable in lambda P

- type checking problem (TCP)
  $\Gamma \vdash P : A?$
  decidable in lambda P

- type synthesis problem (TSP) or typability problem
  $\Gamma \vdash P :?$
  decidable in lambda P