simply typed $\lambda$-calculus

logical verification

week 2

2004 09 15
newsflash

prime number theorem formalized

write $\pi(n)$ for the number of primes below $n$, then

$$\lim_{n \to \infty} \frac{\pi(n)}{n/\ln(n)} = 1$$

http://www.andrew.cmu.edu/user/avigad/isabelle/

- Jeremy Avigad
- Kevin Donelly
- David Gray
overview

last week

<table>
<thead>
<tr>
<th>Logic proofs</th>
<th>Type theory $\lambda$-terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>on paper</td>
<td></td>
</tr>
<tr>
<td>in Coq</td>
<td></td>
</tr>
</tbody>
</table>
why typed $\lambda$-calculus?

C program

```c
#include <math.h>

double findzero(double (*f)(double), double z) {
    double x, y;
    while (x = z, y = (*f)(x), z = x - y/((f)(x + y) - y)*y,
          fabs(z/x - 1) >= 1e-15);
    return z;
}

double sqrminus2(double x) { return x*x - 2; }

main() {
    printf("%.15g\n", findzero(&sqrminus2, 1));
}
```
programming styles

- imperative programming
  C
- object-oriented programming
  C++
  java
- logic programming
  prolog
- functional programming
  lisp
  ML ‘typed’
  haskell ‘lazy’ calculations with infinite data structures
functional programming

functional values become first class objects

no need to name functions anymore

\[
\text{findzero}( \ &\text{sqrmin}us2 \ , \ldots )
\]

\[
\downarrow
\]

\[
\text{findzero}( \ \lambda x.\ x^x - 2 \ , \ldots )
\]

functions also can return functional values

‘higher order’ functions
currying

\[ f : A \times B \rightarrow C \]

partial evaluation

\[ f(a, \cdot) : B \rightarrow C \]

curried version of the function:

\[ f : A \rightarrow (B \rightarrow C) \]

\[ f : A \rightarrow B \rightarrow C \]
the type of findzero

\[(\text{double} \to \text{double}) \times \text{double} \to \text{double}\]

curried:

\[(\text{double} \to \text{double}) \to \text{double} \to \text{double}\]

\[
\uparrow \quad \uparrow
\]

atomic type  function type
simply typed $\lambda$-calculus

types

- atomic types
  \[ A \, B \, C \, \ldots \]

- function types
  \[ A \rightarrow B \]
terms

- **variables**

  \[ x \ y \ z \ldots \]

- **lambda abstraction**

  \[ \lambda x : A. t \]

  the function that maps the variable \( x \) of type \( A \) to \( t \)

- **function application**

  \[ t \ u \]

  the result of applying the function \( t \) to the argument \( u \)
**parentheses**

- function types associate to the right
- application associates to the left

these conventions are natural for curried functions:

\[ f : A \to (B \to C) \]

\[ (f \ a) \ b \]

\[ \downarrow \]

\[ f : A \to B \to C \]

\[ f \ a \ b \]
simplest example

identity function on $A$

term $\lambda x : A. x$

type $A \rightarrow A$
example in the real numbers

term \( \lambda x : \mathbb{R}. x^2 - 2 \)

type \( \mathbb{R} \rightarrow \mathbb{R} \)

\[
(\lambda x : \mathbb{R}. x^2 - 2) \ 1 = 1^2 - 2 = -1
\]

\[
(\lambda x : \mathbb{R}. x^2 - 2) \ 2 = 2^2 - 2 = 2
\]

\(\uparrow\)

\(\beta\)-step
bigger example

term $\lambda x : (A \to B) \to C \to D. \lambda y : C. \lambda z : B. x (\lambda w : A. z) y$

type $((A \to B) \to (C \to D)) \to C \to B \to D$
type derivations

judgments

\[ \Gamma, x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash t : A \]

\( \Gamma \)
context

list of variable declarations
the three typing rules

variable rule

\[
\Gamma, x : A, \Gamma' \vdash x : A
\]
x does not occur in \(\Gamma'\)

abstraction rule

\[
\Gamma, x : A \vdash t : B \\
\Gamma \vdash (\lambda x : A.t) : (A \rightarrow B)
\]

application rule

\[
\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A \\
\Gamma \vdash t \ u : B
\]
type derivation for the example

\[ \vdash \lambda x : (A \to B) \to C \to D. \lambda y : C. \lambda z : B. x (\lambda w : A. z) y : \\
( (A \to B) \to (C \to D)) \to C \to B \to D \]
the Curry-Howard-de Bruijn isomorphism

recap minimal logic

- **formulas**
  - propositional variables
  - implication $A \rightarrow B$

- **rules**
  - implication introduction
  - implication elimination
recap example natural deduction

$((A \rightarrow B) \rightarrow (C \rightarrow D)) \rightarrow C \rightarrow B \rightarrow D$
implication introduction & the abstraction rule

\[ \boxed{A^x} \]

\[ \vdash \]

\[ \frac{B}{A \rightarrow B} \quad I[x] \rightarrow \]

\[ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x : A. t) : (A \rightarrow B)} \]
implication elimination & the application rule

\[
\begin{align*}
\vdash & \quad \vdash \\
A \to B & \quad A & \quad E \to \\
\hline
& B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : A \to B & \quad \Gamma \vdash u : A \\
\hline
\Gamma \vdash t \ u : B
\end{align*}
\]
isomorphism

propositional variable $\sim$ type variable
the connective $\rightarrow$ $\sim$ the type constructor $\rightarrow$
formula $\sim$ type

assumption $\sim$ variable
implication introduction $\sim$ lambda abstraction
implication elimination $\sim$ function application
proof $\sim$ term

provability $\sim$ ‘inhabitation’
proof checking $\sim$ type checking
BHK-interpretation

Brouwer, Heyting, Kolmogorov
intuitionistic logic

proof of $A \rightarrow B \sim$ function that maps proofs of $A$ to proofs $B$
proof of $\bot$ does not exist
proof of $A \land B \sim$ pair of a proof of $A$ and a proof of $B$
proof of $A \lor B \sim$ either a proof of $A$ or a proof of $B$
propostions as types

\[ \lambda x : A. \, x : A \to A \]

the function type \( A \to A \) represents a proposition
the term \( \lambda x : A. \, x \) represents a proof of that proposition

\( \lambda \)-terms are **proof objects**
term syntax

• x

• fun x : A => t

• t u
commands

- Check
  prints a term with its type

- Print
  print the term for a symbol with its type
example

fun x : A => x : A -> A

Coq as proof checker
'->' represents implication

Coq as functional programming language
'->' represents function type
proof objects

Lemma I : A -> A.
...
Qed.
Print I.
example

$((A \rightarrow B) \rightarrow (C \rightarrow D)) \rightarrow C \rightarrow B \rightarrow D$
summary

this week

<table>
<thead>
<tr>
<th>logic proofs (on paper)</th>
<th>type theory terms (in Coq)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>