Lambda-Calculus and Type Theory

ISR 2024
Obergurgl, Austria

Herman Geuvers & Niels van der Weide

Radboud University Nijmegen, The Netherlands

Lecture 1

Introduction; Syntax and Semantics of Untyped Lambda Calculus

First half of the previous century

- Untyped lambda calculus (Church, Curry, Turing)
  - What is (machine) computation? What is computability?
  - Untyped lambda calculus as a model for computation, equivalent to Turing Machines
  - Undecidable problems
  - Untyped lambda calculus is not a good basis to do logic
- Type Theory (Russell and Whitehead)
  - Ramified Theory of Types
  - A formal system for mathematics that avoids paradoxes
- Typed Lambda Calculus (Church)
  - Simple Theory of Types
  - Define higher order logic
  - Use lambda calculus to be clear about variable binding renaming bound variables, substitution, comprehension

Later Developments

Curry-Howard, De Bruijn, Girard, Martin-Löf.

- Interpreting formulas-as types and proofs-as-terms
- Dependent Types
- Polymorphic and Higher Order Types
- Inductive Types

Type theory as

- a language for describing proofs (deductions) as terms
- a basis for proof checking ($\rightarrow$ proof assistants)
- a formalism to define the provable total functions in arithmetic
- a foundadtion for (constructive) mathematics

And apart from that: typed lambda calculus as a basis for functional programming (Turner)

Type Theory now

- Theoretical basis of proof assistants
  - formulas are types
  - proofs (deductions) are terms
  - proof checking $=$ type checking
  - proving $=$ interactively constructing a term of a given type
- Theoretical basis for (functional) programming languages
  - types are specifications
  - terms are programs (with anotations)
  - compiler checks types $\rightarrow$ guarantees (partial) correctness
- Foundation of Mathematics: Constructive Mathematics; Homotopy Type Theory.

Contents

We assume familiarity with logic and natural deduction; formal languages; some functional programming

- Untyped Lambda Calculus (crash course)
- Type theory: simple types, dependent types, polymorphic types, higher order types, inductive types
- The Coq proof assistant
- Some meta-theory of type systems: confluence of reduction (Church-Rosser), normalization of reduction
- Functional programmer's view on type theory: principal types
- Homotopy Type Theory

Proof Assistants, the general picture

What are Proof Assistants for?

- Precise mathematical modelling (defining)
- Verification of properties of systems (proving)

Computer supports in these activities:

- Checking correctness of definitions
- Take care of the bookkeeping
- Do some computation
- Do some proving for us

What have PAs ever done for us?

Does the Proof Assistant do all the proving?
No . . .
It is undecidable in general whether a formula is true or not.

Automated Theorem Provers	Proof Assistants
Specific domains	Generally applicable
Massage your problem	Modelling is direct
False or True (or Don't Know)	Interactive, user guided
No proofs	complete, checkable proofs

Use of PAs

Who is using Proof Assistants and what for?
Computer Scientists for

- Modelling and specifying systems
- Proving the correctness of models / software /systems

Mathematicians for

- Building up theories
- Verifying proofs

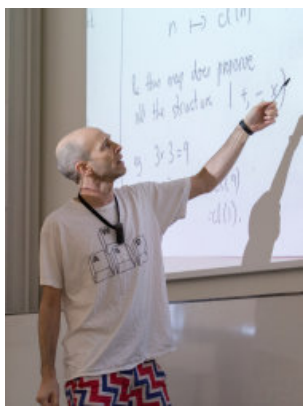
Mathematicians are not (yet) big users of Proof Assistants

- Mechanically verifying a proof takes too much time. (Too much idiosyncracy, not enough automation.)
- We don't need computers to verify proofs! We are much better at it!

Mathematical users of Proof Assistants

Gradually, more mathematicians are getting interested, young mathematicians are less afraid of computers.

- Store formalized mathematics on a computer and make large repositories of formal mathematics actively available.
- Various mathematicians observe that the proofs in their field are becoming too long, complex, abstract that one can only trust them if they are machine verified.
- Kevin Buzzard: Mathlib
  a user maintained library for the
  Lean theorem prover



Computer Science users of Proof Assistants

Compcert (Leroy et al.)

- verifying an optimizing compiler from C to x86/ARM/PowerPC code
- implemented using Coq's functional language
- verified using using Coq's proof language

Xavier Leroy

why?

- your high level program may be correct, maybe you've proved it correct ...
- ... but what if it is compiled to wrong code?
- compilers do a lot of optimizations: switch instructions, remove dead code, re-arrange loops, ...
- for critical software the possibility of miscompilation is an issue

C-compilers are generally not correct

Csmith project *Finding and Understanding Bugs in C Compilers*, X. Yang, Y. Chen, E. Eide, J. Regehr, University of Utah.

> *... we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools.*
> *Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs.*
>
> *As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.*

Some history of Proof Assistants

- Church 1940: λ-calculus, simple type theory, higher order logic
- Curry Howard (De Bruijn): Formulas-as-Types
  Interpret formulas as types,
  Encode proofs as terms
  Proof-checking = Type-checking
- Automath (De Bruijn 1970s): first implementation of these ideas
- LCF (Milner), ML
- Coq, Hol, Isabelle, Lean, Agda, Mizar, PVS, ACL2, ...

Typed λ calculus as the basis for a Proof Assistant

Typed λ calculus forms the basis for a variety of proof Assistants, e.g. Coq (and Lean, Agda, Nuprl, Matita).

λ-term	type
program	specification
proof	formula

Integrated system for proving and programming

Types are not sets

Types are a bit like sets, but types give syntactic information, e.g.

$$3 + (7 \times 8)^5 \ : \ \mathsf{nat}$$

whereas sets give semantic information, e.g.

$$3 \ \in \ \{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)\}.$$

- $3 + (7 \times 8)^5$ is of type nat because $3, 7, 8$ are natural numbers and $\times$, $+$ and power are operations on natural numbers.
- $3 \in \{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)\}$ because there are no positive x, y, z such that $x^3 + y^3 = z^3$, which is an instance of Fermat's last Theorem, proved by Wiles.
- To establish that 3 is an element of the given set, we need a proof, we can't just read it off from the components of the statement.
- To establish $3 + (7 \times 8)^5 : \mathsf{nat}$ we don't need a proof but a simple computation (the "reading the type of of the term").

Decidability of :, undecidability of $\in$

- Membership is undecidable in set theory, as it requires a proof to establish $a \in A$.
- Type checking is decidable: Verifying whether M is of type A requires purely syntactic methods, which can be cast into a typing algorithm.

$$3 + (7 \times 8)^5 : \mathsf{nat} \qquad \text{versus} \qquad \frac{1}{2} \sum_{n=0}^{\infty} 2^{-n} \in \mathbb{N}$$

Question: Can we turn (e.g.)

$$\{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^{+} (x^n + y^n \neq z^n)\}$$

into a (syntactic) type, with decidable type checking?

Phrased differently: can we talk about this set as a "subtype of nat"?

Formulas are also types; proofs are terms

$$\{n \in \mathsf{nat} \mid \forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)\}$$

is a type.
Its terms are pairs $\langle n, p \rangle$ where

- n : nat
- p : $\forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)$

So p is a proof, and we view the formula
$\forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)$ as the type of its proofs.

If we have decidable proof checking, then it is decidable whether a given pair $\langle n, p \rangle$ is typable with the above type or not.

We summarize:

- proof checking = type checking,
- type checking is decidable (so proof checking is decidable),
- proof finding is not decidable (proof finding is required to check an $\in$-judgment).

Next

- Untyped lambda calculus crash course

λ-abstraction

Defining a function

$$\begin{aligned} f(x) &:= x^2 + 2 \\ f &: x \mapsto x^2 + 2 \\ g(x, y) &:= x^2 + y + 2 \end{aligned}$$

In λ-calculus we use λ-abstraction:

$$\begin{aligned} f &:= \lambda x.\, x^2 + 2 \\ g &:= \lambda x.\, \lambda y.\, x^2 + y + 2 \end{aligned}$$

- distinguish between term with a variable $x^2 + 2$ and the function $\lambda x.\, x^2 + 2$ that sends x to $x^2 + 2$.
- make explicit which variables are abstracted over.
- clearly distinguish between free and bound (occurrences) of variables.

Application

We have seen the functions f and g:

$$\begin{aligned} f &:= \lambda x.\, x^2 + 2 \\ g &:= \lambda x.\, \lambda y.\, x^2 + y + 2 \end{aligned}$$

Application:

$$f(3) \quad \text{no!} \qquad f\,3 \quad \text{or} \quad f \cdot 3 \quad \text{or} \quad (f\,3)$$

$\Rightarrow$ application is a binary operator which is usually not written.

Giving two arguments:

$$(g\,3)\,4 \qquad \text{or just} \quad g\,3\,4$$

because we omit brackets by associating them to the left.

Untyped λ-calculus

Untyped λ-calculus = Variables + λ-abstraction + application

$$\Lambda ::= \mathrm{Var} \mid (\Lambda\,\Lambda) \mid (\lambda \mathrm{Var}.\Lambda)$$

Notation

$M\,N\,P$	denotes	$(M\,N)\,P$	(so not $M\,(N\,P)$)
$\lambda xyz.M$	denotes	$\lambda x.\lambda y.\lambda z.M$	(or more precisely $\lambda x.(\lambda y.(\lambda z.M))$.)

Examples:
- $\mathbf{I} := \lambda x.x$
- $\mathbf{K} := \lambda x\,y.x$
- $\mathbf{S} := \lambda x\,y\,z.x\,z(y\,z)$
- $\omega := \lambda x.x\,x$
- $\Omega := \omega\,\omega$

Computing with λ-terms

Computation is done via the β-rule

$$(\lambda x.x^2 + 2)\, 3 \to_\beta 3^2 + 2$$

DEFINITION β-equality, written as $=_\beta$ is the term reduction generated from the β-rule:

$$\boxed{(\lambda x.M)\, P \to_\beta M[x := P]}$$

where $M[x := N]$ denotes the substitution of N for all occurrences of x in M.

That $=_\beta$ is a term reduction means that it is closed under the term-forming-operators. More precisely we have

$$\frac{M \to_\beta M'}{M\, P \to_\beta M'\, P} \qquad \frac{P \to_\beta P'}{M\, P \to_\beta M\, P'} \qquad \frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

Examples

Remember $\mathbf{I} := \lambda x.x$, $\mathbf{K} := \lambda x\, y.x$, $\mathbf{S} := \lambda x\, y\, z.x\, z(y\, z)$, $\omega := \lambda x.x\, x$, $\Omega := \omega\, \omega$.

$$\begin{array}{rcl} \mathbf{I}\, P & \to_\beta & P \\ \mathbf{K}\, P\, Q & \to_\beta & \ldots \to_\beta P \\ \Omega & \to_\beta & \Omega \end{array}$$

$$\begin{array}{rcl} (\lambda x\, y.y\, x)\, P & \to_\beta & \lambda y.y\, P \\ (\lambda x\, y.y\, x)\, y & \overset{??}{\to}_\beta & \lambda y.y\, y \end{array}$$

No!

$\lambda y.M$ binds all occurrences of y in M. We cannot just substitute a term with a free y inside M.

Free and bound variables, alpha-equivalence

- $\lambda y.M$ binds all occurrences of y in M.
- We distinguish bound variables and free variables in a term: BV(M) and FV(M). (Better bound and free occurrences of variables.)
- We consider term modulo renaming of bound variables (also called "modulo α-equality"):

$$\lambda x.M \equiv \lambda y.M[x := y]$$

if y does not occur in M.

A more precise definition of $\rightarrow_\beta$:

$$(\lambda x.M)\, P \rightarrow_\beta M[x := P]$$

where the substitution $M[x := P]$ is defined by: (1) rename the bound variables in M that occur free in P, obtaining M'; (2) replace all free occurrences of x by P.

Alpha equivalence

Two terms M, N are α-equal, $M \equiv N$, in case they can be obtained from eachother via renaming bound variables.

Examples

$$\begin{array}{lcl} \lambda x.\lambda y.x\,y & \overset{??}{\equiv} & \lambda y.\lambda x.y\,x \\ \lambda x.\lambda y.x\,y & \overset{??}{\equiv} & \lambda x.\lambda y.y\,x \\ \lambda x.\lambda y.x\,y & \overset{??}{\equiv} & \lambda x.\lambda y.y\,y \\ \lambda x.\lambda x.x\,x & \overset{??}{\equiv} & \lambda x.\lambda y.y\,y \end{array}$$

Multi-step reduction and β-equality

- $\twoheadrightarrow_\beta$ is the transitive reflexived closure of $\rightarrow_\beta$.
  So $M \twoheadrightarrow_\beta P$ iff M β-reduces to P in 0 or more steps.
- $=_\beta$ is the transitive, reflexive, symmetric closure of $\rightarrow_\beta$.
  So $=_\beta$ is the least congruence obtained from $=_\beta$.

EXAMPLES of reductions:

$$
\begin{array}{rcl}
\mathbf{I}\,P & \rightarrow_\beta & P \\
\mathbf{K}\,P\,Q & \twoheadrightarrow_\beta & P \\
\mathbf{K}\,\mathbf{I}\,P\,Q & \twoheadrightarrow_\beta & Q \\
\mathbf{S}\,\mathbf{K}\mathbf{K} & \twoheadrightarrow_\beta & \mathbf{I} \\
\Omega & \rightarrow_\beta & \Omega
\end{array}
$$

Is λ-calculus consistent?

Why does λ-calculus "make sense"?
Could it be the case that $M =_\beta P$ for all M, P? (Then λ-calculus would be inconsistent...)

THEOREM λ-calculus satisfies the Church-Rosser property.

COROLLARY $\mathbf{K} \neq_\beta \mathbf{I}$ and so λ-calculus is consistent.

The computational power of λ-calculus

Untyped λ-calculus is Turing complete
Its power lies in the fact that you can solve recursive equations:
Is there a term M such that

$$M\,x =_\beta x\,M\,x?$$

Is there a term M such that

$$M\,x =_\beta \textbf{if}\,(\mathsf{Zero}\,x)\,\textbf{then}\,1\,\textbf{else}\,\mathsf{Mult}\,x\,(M\,(\mathsf{Pred}\,x))?$$

Yes, because we have a fixed point combinator:
- $\mathbf{Y} := \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$

Property:

$$\boxed{\mathbf{Y}\,f =_\beta f(\mathbf{Y}\,f)}$$

Untyped λ-calculus (ctd.)

Solving recursive equations using the fixed point combinator:

- For M a λ-term, $\mathbf{Y}\,M$ is a fixed point of M, that is

$$M\,(\mathbf{Y}\,M) =_\beta \mathbf{Y}\,M$$

- As a consequence, a question like "Is there a λ-term P such that $P\,x =_\beta x\,P\,x\,P$ (for all x)?" can be answered affirmative:

Representing data in λ-calculus

Booleans

$$\begin{aligned} \textbf{true} &:= \lambda x\, y.\, x \\ \textbf{false} &:= \lambda x\, y.\, y \\ \textbf{if } M \textbf{ then } P \textbf{ else } Q &:= M\, P\, Q \end{aligned}$$

Natural Numbers via the so-called Church Numerals

$$\begin{aligned} c_0 &:= \lambda f\, x.x \\ c_1 &:= \lambda f\, x.f\, x \\ c_2 &:= \lambda f\, x.f(f\, x) \\ \ldots & \\ c_n &:= \lambda f\, x.f^n\, x \end{aligned}$$

where $f^n\, x$ is an n-times application of f on x.

Then, e.g.

$$\begin{aligned} \textbf{Succ} &:= \lambda n\, f\, x.f(n\, f\, x) \\ \text{Zero} &:= \lambda n.n\,(\lambda y.\, \textbf{false}\,)\ \textbf{true} \end{aligned}$$