[CSE301 / Lecture 3]
Lambda calculus and propositions-as-types

Noam Zeilberger

Ecole Polytechnique

20 September 2023

What is lambda calculus?

- A notation for defining functions and higher-order functions.
- The "original" functional programming language.
- An important tool in the study of programming languages, w/applications to category theory, linguistics, combinatorics...

What is propositions-as-types?

- A salient analogy between proving and programming.
- The retrospective observation that certain formal systems that were defined independently are in fact isomorphic.
- A guiding principle for designing the programming languages and the proof assistants of the future!

Agenda for today (aspirational)

We will go on a whirlwind tour of...

- some core concepts of lambda calculus
- how to program in untyped lambda calculus
- simply-typed λ-calculus and its link with natural deduction
- polymorphic lambda calculus
- the basic machinery of type inference
- a little bit of history

For a long and excellent treatment, see:

- Peter Selinger, *Lecture Notes on the Lambda Calculus*

Recall on algebraic notation

An advantage of usual algebraic notation for arithmetic expressions is that it avoids explicitly staging intermediate computations.

We write

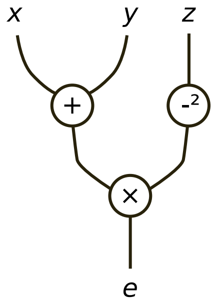
$$e = (x + y) \times z^2$$

rather than

$$\text{Let } w = x + y, \text{ then let } u = z^2, \text{ then let } e = w \times u.$$

Recall on algebraic notation

Diagrammatically, the expression « $e = (x + y) \times z^2$ » can be represented as a tree:



We don't need to label all the inner edges!

A notation for defining intermediate functions

Similarly, rather than writing

Let f be the function $x \mapsto x^2$. Then consider $e = f(5)$...

in the lambda calculus we just write

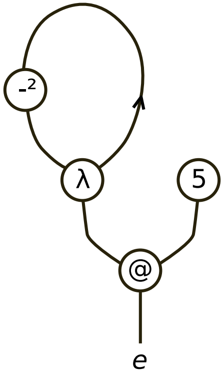
$$e = (\lambda x.x^2)(5)$$

The variable x is said to be **free** in the subexpression x^2, and **bound** in $\lambda x.x^2$. An expression like e with only bound variables is said to be **closed**.

Of course the variable name x is arbitrary, e.g., the expression $e' = (\lambda y.y^2)(5)$ is equivalent. (e and e' are “alpha equivalent”)

A notation for defining intermediate functions

Diagrammatically, the expression « $e = (\lambda x.x^2)(5)$ » can be represented as a graph:



Beta reduction

Application of a λ-expression to a value is defined by substitution.

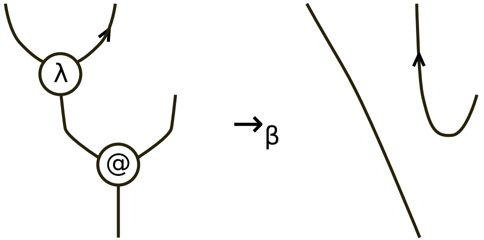
$$(\lambda x.e)(d) \rightarrow_\beta e[d/x]$$

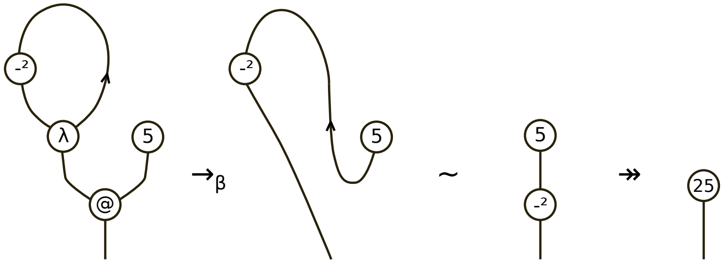
This rule is called **beta reduction**.

Example:

$$(\lambda x.x^2)(5) \rightarrow_\beta 5^2 \twoheadrightarrow 25$$

Beta reduction (visually)





Origins of lambda calculus

Invented by Alonzo Church, first published early 1930s.

Original goal: foundation for logic without free variables.
And it should be more natural than *Principia Mathematica*...

Just one minor defect: Church's original system was inconsistent!

Church (w/help from Kleene and Rosser) was able to salvage the situation by dividing the system in two parts: an **untyped** λ-calculus for computation, and a **typed** λ-calculus for logic.

Untyped lambda calculus

Three ways of building untyped λ-expressions:

$$\begin{aligned} e ::= {} & x && \text{(variable)} \\ \mid {} & e\ e' && \text{(application)} \\ \mid {} & \lambda x.e && \text{(abstraction)} \end{aligned}$$

... and nothing else!

Always consider expressions modulo α-equivalence.

β-reduction $(\lambda x.e)(d) \to e[d/x]$ is the only rule of computation.
(But it can be applied to any subexpression.)

Capture-avoiding substitution, formally

Definition of $e[d/x]$ by induction on e:

$$
\begin{aligned}
x[d/x] &= d \\
y[d/x] &= y && \text{if } x \neq y \\
(e_1\ e_2)[d/x] &= (e_1[d/x])\ (e_2[d/x]) \\
(\lambda x.e')[d/x] &= \lambda x.e' \\
(\lambda y.e')[d/x] &= \lambda y.(e'[d/x]) && \text{if } x \neq y \text{ and } y \notin \mathit{free}(d) \\
(\lambda y.e')[d/x] &= \lambda z.(e'\{z/y\}[d/x]) && \text{if } x \neq y, y \in \mathit{free}(d) \text{ and } z \text{ fresh}
\end{aligned}
$$

Some examples to understand the side-conditions:

$$
\begin{aligned}
(x\ (\lambda x.\lambda y.x))[d/x] &= d\ (\lambda x.\lambda y.x) \\
(\lambda y.x\ y)[(\lambda a.a)/x] &= \lambda y.(\lambda a.a)\ y \\
(\lambda y.x\ y)[(\lambda a.y\ a)/x] &= \lambda z.(\lambda a.y\ a)\ z
\end{aligned}
$$

Beta normalization

Normalization = repeatedly applying (β-)reductions until there are none left, i.e., the expression is in (β-)**normal form**.

Church-Rosser theorem (cf. Selinger §4.2–4.4):

$$\begin{tikzcd}
 & e \arrow[dl] \arrow[dr] & \\
d_1 \arrow[dr, dashed, two heads] & & d_2 \arrow[dl, dashed, two heads] \\
 & c &
\end{tikzcd}$$

Corollary: every expression has at most one normal form.

...But some have no normal form!

Beta normalization[1]

$$\begin{tikzcd}
 & (\lambda x.x)(\lambda y.(\lambda z.z)y) \arrow[dl] \arrow[dr] & \\
\lambda y.(\lambda z.z)y \arrow[dr] & & (\lambda x.x)(\lambda y.y) \arrow[dl] \\
 & \lambda y.y &
\end{tikzcd}$$

$$\omega = (\lambda x.x\,x)(\lambda x.x\,x) \longrightarrow (\lambda x.x\,x)(\lambda x.x\,x) \longrightarrow \dots$$

[1] A handy tool for visualizing lambda term reduction: `https://www.georgejkaye.com/lamviz/visualiser/`

Untyped lambda calculus

If a β-normal form exists, we can always find it by applying a "leftmost outermost" reduction strategy. But determining whether a λ-expression has a normal form is undecidable, as established by Church in 1936.[2]

His proof relied on first showing how to express a lot of arithmetic in untyped lambda calculus. This was kind of surprising, since pure λ-calculus does not have any built-in numbers or even booleans!

[2] This was actually the first example of an undecidable problem in print! Turing's paper on the Halting Problem came out a year later.

Church encoding: the booleans

Idea: "a boolean is a function that selects between two options."

$$\mathit{True} \stackrel{\mathrm{def}}{=} \lambda xy.x \quad \mathit{False} \stackrel{\mathrm{def}}{=} \lambda xy.y$$

Conditional expressions are easy in this representation:

$$\mathbf{if}\ b\ \mathbf{then}\ e\ \mathbf{else}\ e' \stackrel{\mathrm{def}}{=} b\ e\ e'$$

Negation, conjunction, disjunction:

$$\mathit{not} \stackrel{\mathrm{def}}{=} \lambda bxy.b\ y\ x \quad \mathit{and} \stackrel{\mathrm{def}}{=} \lambda bc.b\ c\ \mathit{False} \quad \mathit{or} \stackrel{\mathrm{def}}{=} \lambda bc.b\ \mathit{True}\ c$$

Church encoding: the natural numbers

Idea: "a natural number is a higher-order function sending any function to its *n*-fold composition."

$$0 \overset{\text{def}}{=} \lambda f.\lambda x.x \quad 1 \overset{\text{def}}{=} \lambda f.\lambda x.f(x) \quad 2 \overset{\text{def}}{=} \lambda f.\lambda x.f(f(x)) \quad \ldots$$

Successor function:

$$succ \overset{\text{def}}{=} \lambda n.\lambda f.\lambda x.f(n\, f\, x)$$

Addition and multiplication:

$$add \overset{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x) \qquad mult \overset{\text{def}}{=} \lambda mnfx.m\,(n\ f)\ x$$

Test for zero:

$$isZero \overset{\text{def}}{=} \lambda n.n\,(\lambda b.False)\,True$$

Church encoding other data types

Lists: $[a_1, \ldots, a_n] \stackrel{\text{def}}{=} \lambda f.\lambda x.f\,a_1\,(f\,a_2\,(\ldots(f\,a_n\,x)))$
(Does this remind you of anything?)

Pairs: $(u, v) \stackrel{\text{def}}{=} \lambda f.f\,u\,v$

et cetera

Defining the predecessor function (slightly apocryphal anecdote)

From Henk Barendregt's essay, "Gems of Corrado Böhm":

> *At first neither Church nor his students could find a way to lambda define the predecessor function. At the dentist's office Kleene did see how to simulate recursion by iteration and could in that way construct a lambda term defining the predecessor function, [Cro75]. (I believe Kleene told me it was under the influence of laughing gas, N2O, used as anesthetic.) When Church saw that result he stated "Then all intuitively computable functions must be lambda definable."*

Defining recursive functions

Key idea: first define a **fixed point operator** in untyped LC.

Here is one:[3] $\Theta \stackrel{\text{def}}{=} (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

We have $\Theta\, f \twoheadrightarrow f\,(\Theta\, f)$ for any f, meaning that

$$f\,(\Theta\, f) =_\beta \Theta\, f\,.$$

Thus Θ finds a fixed point of f!

[3] Due to Alan Turing, who used it to prove the equivalence between the expressive power of untyped λ-calculus and Turing machines. Another famous fixed point operator is Haskell Curry's $\mathsf{Y} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$.

Defining recursive functions

Fixed point operators can be used to define recursive functions.

For example:

$$\mathit{fact} \stackrel{\text{def}}{=} \Theta(\lambda f n.\mathit{isZero}\ n\ 1\ (\mathit{mult}\ n\ (f\ (\mathit{pred}\ n))))$$

We have that

$$\mathit{fact}\ n \twoheadrightarrow \mathit{isZero}\ n\ 1\ (\mathit{mult}\ n\ (\mathit{fact}\ (\mathit{pred}\ n)))$$

(Aside: Alonzo Church as a teacher)

From “Indiscrete Thoughts” by Gian-Carlo Rota:

> *He looked like a cross between a panda and a large owl. He spoke slowly in complete paragraphs which seemed to have been read out of a book, evenly and slowly enunciated, as by a talking machine. When interrupted, he would pause for an uncomfortably long period to recover the thread of the argument. He never made casual remarks: they did not belong in the baggage of formal logic. For example, he would not say: "It is raining." Such a statement, taken in isolation, makes no sense. (Whether it is actually raining or not does not matter; what matters is consistency.) He would say instead: "I must postpone my departure for Nassau Street, inasmuch as it is raining, a fact which I can verify by looking out the window." (These were not his exact words.) [...]*

Simply-typed lambda calculus

In Church's original formulation (1940), every variable is annotated with a type, and expressions are built up in a way that ensures that every well-typed expression has a unique type.

For example,

$$\lambda x_\iota.\lambda y_\iota.x_\iota \quad \text{and} \quad \lambda x_\iota.\lambda y_\iota.y_\iota$$

are well-typed expressions of type $\iota \to \iota \to \iota$, and

$$\lambda f_{\iota\to\iota\to\iota}.\lambda x_\iota.\lambda y_\iota.f\ y\ x$$

is a well-typed expression of type $(\iota \to \iota \to \iota) \to (\iota \to \iota \to \iota)$.

Typing contexts and typing rules

A more modern convention is to consider expressions in a *typing context* for their free variables.

We write $\Gamma \vdash e : B$ to mean that e is an expression of type B in a context $\Gamma = x_1 : A_1, \dots, x_n : A_n$ of types for its free variables.

Typing rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\,e' : B} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}$$

Type safety and strong normalization

Preservation: if $e : A$ and $e \to e'$ then $e' : A$.

Progress: if $e : A$ then either e is β-normal or $\exists e'.e \to e'$.

Termination: if $e : A$ then $e \twoheadrightarrow v$ to some (unique) β-normal v.

Corollary: ω and Θ can't be typed!

Product and sum types

Easy to extend STLC with products.

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \pi_1 e : A} \quad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \pi_2 e : A}$$

$$\pi_1(e_1, e_2) \to e_1 \qquad \pi_2(e_1, e_2) \to e_1$$

Can similarly add typing rules and reductions for sum types.

Type safety and strong normalization are preserved.

A surprising correspondence

STLC turns out to be isomorphic to (a subsystem of) a proof system for logic introduced earlier by Gerhard Gentzen, called *natural deduction*.

A key idea of ND is that every logical connective is governed by “introduction” and “elimination” rules.

Natural deduction

For example, conjunction defined by the following rules:

$$\frac{A \quad B}{A \wedge B}\ {\wedge I} \qquad \frac{A \wedge B}{A}\ {\wedge E_1} \quad \frac{A \wedge B}{B}\ {\wedge E_2}$$

And universal quantification by the following rules:

$$\frac{A[a/x]}{\forall x.A}\ {\forall I} \qquad \frac{\forall x.A}{A[t/x]}\ {\forall E}$$

(where *a* is a fresh variable not appearing in *A*, and *t* is any term.)

Natural deduction

Implication defined by:

$$\dfrac{\begin{matrix} {}^{x}A \\ \vdots \\ B \end{matrix}}{A \supset B}\;{\supset}I^{x} \qquad \dfrac{A \supset B \quad A}{B}\;{\supset}E$$

Note the special form of the intro: "Suppose we can derive B from the assumption of A, then $A \supset B$ holds."

The $\supset I$ rule is said to "discharge" the assumption labelled x.

A proof with no undischarged assumptions is said to be *closed*.

Natural deduction

Example of a closed proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{{}^{x}B \supset C \qquad \dfrac{{}^{y}A \supset B \quad {}^{z}A}{B}\,{\supset}E}{C}\,{\supset}E}{A \supset C}\,{\supset}I^{z}}{(A \supset B) \supset (A \supset C)}\,{\supset}I^{y}}{(B \supset C) \supset (A \supset B) \supset (A \supset C)}\,{\supset}I^{x}$$

ND $\sim$ STLC

We can alternatively describe rules $\supset I$ and $\supset E$ using explicit contexts of assumptions, and using lambda notation for the proofs. While we're at it, let's write $\supset$ as $\rightarrow$:

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \rightarrow B}\ {\scriptstyle \rightarrow I} \qquad \frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B}\ {\scriptstyle \rightarrow E}$$

But these are the typing rules of simply-typed lambda calculus!

Indeed, every simply-typed λ-term may be seen as a proof in (intuitionistic[4], implicational) natural deduction, and vice versa.

[4]Intuitionistic logic (also called constructive logic) is a generalization of classical logic where one does not accept non-constructive principles such as the law of excluded middle $A \vee \neg A$ and double-negation elimination $\neg\neg A \supset A$.

Polymorphism

The identity function $\lambda x.x$ can be given type $A \to A$ for any A.

Intuitively, it works "the same way" regardless of the choice of A, but there is no way of expressing that in STLC.

Using **polymorphic quantification** we can say $\lambda x.x$ has type

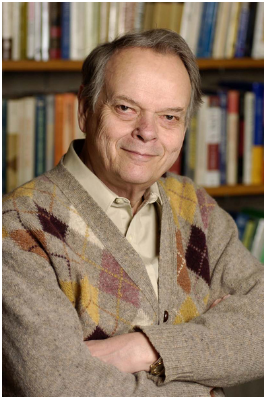
$$\forall a.a \to a$$

where a is a type variable. (Logically, $\forall a$ may be seen as quantifying over formulas.)

Polymorphic lambda calculus

Invented independently by a logician and a computer scientist:

Jean-Yves Girard



John C. Reynolds

Polymorphic lambda calculus

One nice property: typed Church encodings!

$$\begin{aligned} \mathit{Bool} &= \forall a.\, a \to a \to a \\ \mathit{Nat} &= \forall a.\, (a \to a) \to (a \to a) \\ \mathit{List}\, a &= \forall b.\, (a \to b \to b) \to (b \to b) \\ \mathit{Both}\, a\, b &= \forall c. (a \to b \to c) \to c \end{aligned}$$

But note we still have strong normalization. (So still can't type Θ)

Let's try this out in Agda and Haskell...

Polymorphism in Haskell and OCaml

In Girard and Reynold's calculus, applying a polymorphic function requires explicitly passing a type argument.

In Haskell and OCaml, types are inferred automatically through **Hindley-Milner type inference**, as long as you stick to *prenex polymorphism* of the form $\forall a_1 \ldots \forall a_n.A$ (with A monomorphic).

Both also include some support for *higher-rank polymorphism* (polymorphic types as an argument to a function), but type inference becomes more difficult or undecidable.

Hindley-Milner type inference

Basically: examine the program to generate a lot of typing constraints, and try to solve them!

(Let's try some some examples by hand and by ghci...)

```
ghci> :t (\f x -> f (f x))
  ...
ghci> :t (\x y z -> x (y z))
  ...
```

Hindley-Milner type inference

It works very well when it works, but sometimes gives misleading error messages...

```
ghci> foldl (flip (++)) [] "hello"

<interactive>:128:22-28: error:
    * Couldn't match type 'Char' with '[a]'
      Expected type: [[a]]
        Actual type: [Char]
    * In the third argument of 'foldl', namely '"hello"'
      In the expression: foldl (flip (++)) [] "hello"
      In an equation for 'it': it = foldl (flip (++)) [] "hello"
    * Relevant bindings include
        it :: [a] (bound at <interactive>:128:1)
ghci> foldl (flip (:)) [] "hello"
"olleh"
```