Untyped λ-Calculus

Faridah Akinotcho, Lily Bryant, Junze Wu
Outline

- History
- Construction and reductions
- Encoding data types and structures
- Recursion and the Y combinator
- Variants and influence
Historical Figures

Alonzo Church (1903-1995)

Alan Turing (1912-1954)

Lambda Calculus  ↔  Turing Machines
Why \( \lambda \)?

\[
f(x) = 2x + 1 \\
\hat{x}.2x + 1 \\
\land x.2x + 1 \\
\lambda x.2x + 1
\]
Formal Description
Definition of $\lambda$-terms

Assume infinite set of variables $V = \{x, y, z, \ldots\}$.

Set $\Lambda$ of $\lambda$-terms:

(Variable) If $u \in V$, then $u \in \Lambda$.

(Application) If $M$ and $N \in \Lambda$, then $(M \ N) \in \Lambda$.

(Abstraction) If $u \in V$ and $M \in \Lambda$, then $(\lambda u.M) \in \Lambda$.

Example:

$[((\lambda x. x) \ (\lambda y. y \ z))]$
Terminology: Scoping and Boundness

**Free Variables**

\[ FV : \Lambda \rightarrow \mathcal{P}(V) \]

(Variable) \[ FV(x) = x \]

(Abstraction) \[ FV(\lambda x.M) = FV(M) \setminus \{x\} \]

(Application) \[ FV(M N) = FV(M) \cup FV(N) \]

⇒ A \(\lambda\)-term \(M\) is **closed** if \(FV(M) = \emptyset\).

\(M\) is also called a **combinator**.

**Bound Variables**

\[ BV : \Lambda \rightarrow \mathcal{P}(V) \]

(Variable) \[ BV(x) = \emptyset \]

(Abstraction) \[ BV(\lambda x.M) = \{x\} \cup BV(M) \]

(Application) \[ BV(M N) = BV(M) \cup BV(N) \]
Terminology: Substitution

Given a \( \lambda \)-term \( M \), we denote the substitution of a variable \( x \) in \( M \) by \( N \) as \( M[x:=N] \) such that:

- \( x[x:=N] = N \)
- \( y[x:=N] = y \) if \( x \neq y \)
- \( (PQ)[x:=N] = (P[x:=N])(Q[x:=N]) \)
- \( (\lambda y.P)[x:=N] = \lambda z.(P^y_z[x:=N]) \) where \( P^y_z \) stands for \( P \) with every instance of \( y \) renamed to \( z \), such that \( z \notin FV(N) \)
Terminology: Currying

A common shorthand is to use multi-argument functions. These can be rewritten as nested single-argument functions where each “level” takes an argument and returns another function.

Examples:

\[
(\lambda x \ y.(x \ y)) \equiv (\lambda x.(\lambda y.(x \ y)))
\]

\[
((\lambda x \ y.(x \ y)) \ 5) \equiv ((\lambda x.(\lambda y.(x \ y))) \ 5) \rightarrow^\beta (\lambda y.(5 \ y))
\]

“partial application”
Reductions

- $\alpha$-conversion: Argument renaming in functions
- $\beta$-reduction: Function application with substitution
- $\eta$-conversion: Function equivalence
Reductions: $\alpha$-conversion

Let $M \in \Lambda$.

$\alpha$-conversion performs the following renaming, provided $y \notin \text{FV}(M)$ and $y \notin \text{BV}(M)$:

$$\lambda x. M =_{\alpha} \lambda y. (M[x := y])$$

$\Rightarrow \lambda x. x$ and $\lambda z. z$ are $\alpha$-equivalent, but not syntactically-equivalent.
Reductions: $\beta$-reduction semantics

One-step $\beta$-reduction

1. $(\lambda x.M)N \rightarrow_\beta M[x := N]$

2. If $M \rightarrow_\beta N$, then $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LN$ and $\lambda x.M \rightarrow_\beta \lambda x.N$

$\beta$-reduction (zero-or-more-step)

$M \rightarrow_\beta N$, if $\exists n > 0$ and $M_0, \ldots, M_n$ such that $M_0 =_\alpha M$, $M_n =_\alpha N$ and

$$M_0 \rightarrow_\beta M_1 \rightarrow_\beta \ldots \rightarrow_\beta M_n$$
Reductions: \( \beta \)-reduction terminology

\[
\begin{align*}
\ldots \ldots ((\lambda x \cdot M)N) \ldots \ldots & \xrightarrow{\beta} \ldots \ldots (M[x := N]) \ldots \ldots \\
\end{align*}
\]

- The left hand side is called a *redex* and the right hand side is the *contractum*.
- If \( P \rightarrow_{\beta} L \), then \( P \) and \( L \) are said to be *\( \beta \)-equivalent*, denoted \( P =_{\beta} L \).
Reductions: $\beta$-reduction terminology

$\Rightarrow$ M is in $\beta$-normal form (or $\beta$-nf) if M does not contain a redex.

$\Rightarrow$ M is weakly-normalizable if there exists some N in $\beta$-nf where $M \rightarrow^{*}_{\beta} N$.

$\Rightarrow$ M is strongly-normalizable if there does not exist an infinite reduction path from M to N.
Reductions: $\beta$-reduction theorems

Church-Rosser Theorem

$\lambda$-calculus is confluent under $\beta$-reductions, i.e. supposing $M, M_1, M_2, L \in \Lambda$,

If $M \rightarrow_\beta M_1$ and $M \rightarrow_\beta M_2$, then there exists some $L$ such that $M_1 \rightarrow_\beta L$ and $M_2 \rightarrow_\beta L$.

Consequences:

$\Rightarrow$ A $\lambda$-term has at most one normal form.

$\Rightarrow$ Ordering of reduction choice does not matter (non-deterministic reduction!). This even applies in many typed versions of the calculus.
Reductions: $\beta$-reduction theorems

A set $L \subseteq \Lambda$ is closed under $\beta$-conversion if

$$\forall M, N \in \Lambda, ((M \in L \land M =_{\beta} N) \Rightarrow N \in L).$$

*Scott-Curry Theorem*

If two non-empty, disjoint sets of $\lambda$-terms are closed under $\beta$-conversion,

then they are recursively inseparable.

Consequence:

⇒ Showing two non-empty $\lambda$-terms are $\beta$-equivalent is undecidable.
Reductions: $\eta$-conversion

If function $F$ takes its argument and immediately applies the argument to $G$, then $F$ is *functionally-equivalent* to $G$.

$$\lambda x. (M \ x) \rightarrow^\eta M \quad \text{where } x \notin \text{FV}(M)$$

⇒ Intuitively: two functions are identical if they do the same thing.
Encoding data types and structures
FUNCTIONS

FUNCTIONS EVERYWHERE
Boolean Logic

Truth values

TRUE := λx y. x
FALSE := λx y. y

With these two terms, we can define:

NOT := λp. ((p FALSE) TRUE)
AND := λp q. ((p q) p)
OR := λp q. ((p p) q)

IFTHENELSE := λc t e. ((c t) e)
Numbers

Church numerals

ZERO := \( \lambda f \ x. \ x \)
ONE := \( \lambda f \ x. \ (f \ x) \)
TWO := \( \lambda f \ x. \ (f \ (f \ x)) \)
THREE := \( \lambda f \ x. \ (f \ (f \ (f \ x))) \)
...
\( n := \lambda f \ x. \ f^n x \)

Peano numbers

A NATURAL is one of:

- ZERO
- SUCC NATURAL

SUCC := \( \lambda n f x. \ (f \ ((n f) x)) \)
Arithmetic

Elements needed: successor, predecessor, addition, subtraction, multiplication, division, exponentiation.

Immediately, we can define:

PLUS := λm n. ((m SUCC) n)
MULT := λm n. ((m (PLUS n)) ZERO)

Alternatively, MULT := λm n f. (m (n f))

EXP := λm n. (n m)
Pairs and Lists

For predecessor and subtraction, we first need to introduce pairs:

Encoding pairs and lists

\[ \text{CONS} := (\lambda a \ b \ s.((s \ a) \ b)) \]

\[ \text{CAR} := (\lambda p.(p \ \text{TRUE})) \]

\[ \text{CDR} := (\lambda p.(p \ \text{FALSE})) \]

\[ \text{NIL} := \text{FALSE} \]
The “wisdom tooth trick”:

\[ T := (\lambda p.((\text{CONS} (\text{SUCC} (\text{CAR} p))) (\text{CAR} p)))) \]

\[ \text{PRED} := (\lambda n. (\text{CDR} ((n T) ((\text{CONS} \text{ZERO}) \text{ZERO})))) \]

\[ \text{SUB} := (\lambda m \ n. (n \ \text{PRED}) \ m) \]

Note: \text{PRED} \ \text{ZERO} \ \text{is} \ \text{ZERO}
Predicates

ZERO? := λn.((n (λx. FALSE)) TRUE)
LEQ? := λm n.ZERO?((SUB m) n)
GT? := λm n.NOT ((LEQ? m) n)
EQ? := λm n.((AND ((LEQ? m) n)) ((LEQ? n) m)))
Named Constants

We can introduce let binding, with the following relation:

\[ \text{LET } x = n \text{ IN } m \equiv ((\lambda x.m) n) \]
Division is a **recursive** procedure:

```
(define (divide n m)
  (if (>= n m)
      (+ 1 (divide (- n m) m))
      0))
```

But we don’t have a way to do recursion yet...
Recursion
Y Combinator

\[ \lambda f. (\lambda x. (f (x x))) \lambda x. (f (x x)) \]
An example, the factorial function

```racket
#lang racket
(define factorial
  (λ (n)
    (if (zero? n) 1 1 (* n (factorial (sub1 n))))))

;; Sadly, we can’t do this in λ-calculus
```
An example, the factorial function

```racket
#lang racket
(define almost-factorial
  (λ (f)
    (λ (n)
      (if (zero? n)
          1
          (* n (f (sub1 n)))))))
```

What function should we pass in for \( f \) in order to get the actual factorial function back?

It is *the actual factorial function* that we are trying to define!
An example, the factorial function

So the $f$ that we are looking for is a “fixed-point” of almost-factorial:

$$(\text{almost-factorial } f) = f$$
A function has a fixed point at $a$ if $f(a) = a$. Function $f$ applied to $a$ returns $a$ again: “Fixed” by $f$.

Examples (functions defined on real numbers):

$f(n) = n^2$ has two fixed points: 0 and 1

$f(n) = n + 1$ has no fixed points

$$f(f(\ldots f(f(a))\ldots)) = a$$
Fixed-point theorem

For all $L \in \Lambda$ there is $M \in \Lambda$ such that $(L \ M) =_{\beta} M$.

Proof:

For given $L$, define $M := (((\lambda x. (L(x \ x))) \ (\lambda x. (L(x \ x)))) \ (\lambda x. (L(x \ x))))$.

This $M$ is a redex, so we have:

$M \equiv (((\lambda x. L(x \ x))) \ (\lambda x. (L(x \ x)))) \rightarrow_{\beta} L(((\lambda x. L(x \ x))) \ (\lambda x. L(x \ x)))) \equiv (L \ M)$.

Hence, $(L \ M) =_{\beta} M$. 
Recursion: Y combinator

Fixed-point combinator:

\[ Y := \lambda f. (\lambda x. (f \ (x\ x)) \ \lambda x. (f \ (x\ x))) \]

\( Y \) is a higher order function that computes the fixed-point of the argument function. For any \( \lambda \)-term \( f \), \( (Y \ f) \) is a fixed point of \( f \):

\[ (Y \ f) = (f \ (Y \ f)) \]
An example, the factorial function

```scheme
#lang lazy
(define Y (\( f \) ((\( x \) (f (x x)))) (\( x \) (f (x x)))))
(define factorial (Y
    (\( f \)
        (\( n \)
            (if (zero? n)
                1
                (* n (f (sub1 n)))))))
```

An example, the factorial function

```scheme
#lang lazy
(define Y (λ (f) ((λ (x) (f (x x))) (λ (x) (f (x x))))))
(define FACTORIAL (Y
    (λ (f)
        (λ (n)
            (((ZERO? n) ONE)
             (((MULT n) (f (PRED n)))))
        )))
))
```
Division

(define divide
  (Y (λ (f)
       (λ (n)
         (λ (m)
           (if (>= n m)
               (add1 ((f (- n m)) m))
               0))))))
Variants and Influence
Variants in Programming
Reduction Strategies

We define a strategy as a function $S$,

$S: \Lambda \rightarrow \Lambda$

If $S(M) = N$, then $M \rightarrow_{\beta} N$. 
Reduction Strategies

Full Beta Reductions: Any redex can be reduced at any time

Applicative Order Reduction: The rightmost, innermost term is always evaluated first, i.e. the arguments to a function are always evaluated first. It is a call-by-value evaluation strategy.

\[ ((\lambda x. M) (\lambda y. P)) \]
Reduction Strategies

Normal Order Reduction: The leftmost, outermost redex is always reduced first, i.e. whenever possible the arguments are substituted into the body of an abstraction before they are reduced.

$$(((\lambda x. M) N) (\lambda y. P))$$

It is similar to call-by-name strategy except reductions can happen in the body of an abstraction, ex:

$$\lambda x.(\lambda x.x)x$$ is in normal form according to the call-by-name strategy, but not the normal order since it contains the redex $$(\lambda x.x)x$$
Reduction Strategies: an example

We define omega, a combinator $\Omega=((\lambda x.(xx))(\lambda x.(xx)))$

Let $M:=(\lambda u.v)\Omega$

There is an infinite reduction by contracting $\Omega$ and a one step reduction, in other words $M$ is weakly normalizing.
**Typed \( \lambda \)-calculus**

- **(var)** \( \Gamma \vdash x : \sigma \) if \( x : \sigma \in \Gamma \)

- **(appl)**
  \[
  \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}
  \]

- **(abst)**
  \[
  \frac{\Gamma, \ x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \rightarrow \tau}
  \]
Functional Programming

- $\lambda$-calculus and its variants provide the theoretical foundation for functional languages such as Lisp, ML, and Scheme.

- *System F* is a typed $\lambda$-calculus that formed the basis for some typed functional languages, such as Haskell and OCaml, and define *parameter polymorphism*: the idea allowing of generic type variables.

- Anonymous functions and other aspects of functional programming have also been introduced into many non-functional languages, such as Java, C, and MATLAB.
Variants in Logic
De Bruijn Indices

Alternative to Church’s notation which removes the overhead of naming variables.

Each variable becomes a natural number equal to the number of variables/lambdas introduced between its definition and usage (inclusive).

Example:

\[(\lambda x.\lambda y.\lambda z. \ w \ y \ x \ z) \ (\lambda u.\lambda v. v \ u)) \quad ((\lambda \lambda \lambda 4 \ 2 \ 3 \ 1) \ (\lambda \lambda 1 \ 2))\]

\(w\) is a free variable

\(\Rightarrow\) \(\alpha\)-equivalent (renamed) terms are now syntactically-equivalent
SKI calculus

S: \((S \times y z) \equiv (\lambda x y z.((x z) (y z))) \rightarrow_\beta (x z (y z))\)

K: \((K \times y) \equiv (\lambda x y. x) \rightarrow_\beta x\)

I: \((I \times) \rightarrow \eta ((S K K) x) \rightarrow_\beta x\)

⇒ For all \(M \in \Lambda\), there exists a combination of \(S\) and \(K\) that is functionally-equivalent to \(M\). \(S\) and \(K\) form the basis of all \(\lambda\)-terms.

⇒ Essentially, a tiny Turing-complete language!

⇒ Combinatory logic was developed around the same time as \(\lambda\)-calculus and works with this idea of combinators (without abstraction) as a basis for reasoning.
Conclusion

$\lambda$-calculus is..

- A simple way of capturing computability while still providing many powerful results
- Very helpful for building up the logical basis of functional reasoning
- Less helpful for finding intuition on very high level descriptions of algorithms
References

(Book) Church, A. (1941). The calculi of lambda-conversion.


(Screencast) Lambda Calculus: PyCon 2019 Tutorial
Some people prefer not to commingle the functional, lambda-calculus part of a language with the parts that do side effects. It seems they believe in the separation of Church and state.

— Guy Steele —