

Ordinals and Typed Lambda Calculus

Lambda Calculus

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2018-2

Introduction

Alonzo Church (1903 – 1995)*



*Figures sources: [History of computers](#), [Wikipedia](#) and [MacTutor History of Mathematics](#).

Introduction

Some remarks

- A formal system invented by Church around 1930s.
- The goal was to use the λ -calculus in the **foundation** of mathematics.
- Intended for studying **functions** and **recursion**.
- Computability model.
- A free-type functional programming language.
- λ -notation (e.g. anonymous functions and currying).

Application, Abstraction and Curryfication

Application

Application of the function M to argument N is denoted by $M N$ (juxtaposition).

Application, Abstraction and Curryfication

Application

Application of the function M to argument N is denoted by $M N$ (juxtaposition).

Abstraction

'If M is any formula containing the variable x , then $\lambda x[M]$ is a symbol for the function whose values are those given by the formula.' [Church 1932, p. 352]

Application, Abstraction and Curryfication

Application

Application of the function M to argument N is denoted by $M N$ (juxtaposition).

Abstraction

'If M is any formula containing the variable x , then $\lambda x[M]$ is a symbol for the function whose values are those given by the formula.' [Church 1932, p. 352]

Curryfication

'Adopting a device due to Schönfinkel, we treat a function of two variables as a function of one variable whose values are functions of one variable, and a function of three or more variables similarly.' [Church 1932, p. 352]

Lambda Terms

Definition

Let V be a denumerable set of variables. The set of **λ -terms**, denoted by Λ , is inductively defined by

$$x \in V \Rightarrow x \in \Lambda \quad \text{(variable)}$$

$$M, N \in \Lambda \Rightarrow (M N) \in \Lambda \quad \text{(application)}$$

$$M \in \Lambda, x \in V \Rightarrow (\lambda x.M) \in \Lambda \quad \text{(\lambda-abstraction)}$$

Lambda Terms

Remark

Usually, the set of λ -terms Λ is defined by the abstract grammar*

$\Lambda \ni t ::= x$	(variable)
$ \ t \ t$	(application)
$ \ \lambda x. t$	(λ -abstraction)

*See, e.g. [Pierce 2002].

Lambda Terms

Notation

The symbol ' \equiv ' denotes the syntactic identity.

Conventions

- λ -term **variables** will be denoted by x, y, z, \dots
- **λ -terms** will be denoted by M, N, P, Q, \dots

Lambda Terms

Conventions and syntactic sugar

- Outermost parentheses are not written.
- Application has higher precedence, that is,

$$\lambda x. M N := (\lambda x. (M N)).$$

- Application associates to the left, that is,

$$M N_1 N_2 \dots N_k := (\dots ((M N_1) N_2) \dots N_k).$$

- Lambda abstraction associates to the right, that is,

$$\begin{aligned} \lambda x_1 x_2 \dots x_n. M &:= \lambda x_1. \lambda x_2. \dots \lambda x_n. M \\ &:= (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. M) \dots))). \end{aligned}$$

Lambda Terms

Example

Using the conventions and syntactic sugar.

$(\lambda x y z. x z (y z)) u v w$	
$\equiv (\lambda x y z. (x z) (y z)) u v w$	(left-associative application)
$\equiv ((\lambda x y z. (x z) (y z)) u) v w$	(left-associative application)
$\equiv (((\lambda x y z. (x z) (y z)) u) v) w$	(left-associative application)
$\equiv (((\lambda x y z. ((x z) (y z))) u) v) w$	(application higher precedence)
$\equiv (((\lambda x. \lambda y. \lambda z. ((x z) (y z))) u) v) w$	(right-associative λ -abstraction)
$\equiv (((\lambda x. \lambda y. (\lambda z. ((x z) (y z)))) u) v) w$	(right-associative λ -abstraction)
$\equiv (((\lambda x. (\lambda y. (\lambda z. ((x z) (y z))))) u) v) w$	(right-associative λ -abstraction)
$\equiv (((((\lambda x. (\lambda y. (\lambda z. ((x z) (y z))))) u) v) w)$	(remove outermost parentheses)

Binding

Definition

A variable x occurs **free** in M if x is not in the scope of λx . Otherwise, x occurs **bound**.

Definition

The **set of free variables in M** , denoted by $\text{FV}(M)$, is inductively defined by

$$\text{FV}(x) \quad := \{x\},$$

$$\text{FV}(M N) \quad := \text{FV}(M) \cup \text{FV}(N),$$

$$\text{FV}(\lambda x.M) \quad := \text{FV}(M) - \{x\}.$$

Substitution

Definition

The result of **substituting N for every free occurrence of x in M** , and changing bound variables to avoid clashes, denoted by $M[x \mapsto N]$, is defined by [Hindley and Seldin 2008, Definition 1.12]

$$x[x \mapsto N] := N;$$

$$y[x \mapsto N] := y, \quad \text{if } y \neq x;$$

$$(PQ)[x \mapsto N] := P[x \mapsto N]Q[x \mapsto N];$$

$$(\lambda x.P)[x \mapsto N] := \lambda x.P;$$

$$(\lambda y.P)[x \mapsto N] := \lambda y.P, \quad \text{if } y \neq x \text{ and } x \notin \text{FV}(P);$$

$$(\lambda y.P)[x \mapsto N] := \lambda y.P[x \mapsto N], \quad \text{if } y \neq x, x \in \text{FV}(P) \text{ and } y \notin \text{FV}(N);$$

$$(\lambda y.P)[x \mapsto N] := \lambda z.P[x \mapsto N][y \mapsto z], \quad \text{if } y \neq x, x \in \text{FV}(P) \text{ and } y \in \text{FV}(N);$$

where in the last equation, the variable z is chosen such that $z \notin \text{FV}(NP)$.

Substitution

Example

$(y (\lambda v. x v)) [x \mapsto (\lambda y. v y)] \equiv y (\lambda z. (\lambda y. v y) z)$ (with $z \neq v, y, x$).

Conversion Rules

Introduction

The functional behaviour of the λ -calculus is formalised through of their conversion rules:

$$\lambda x.N =_{\alpha} \lambda y.(N[x \mapsto y]) \quad (\alpha\text{-conversion})$$

$$(\lambda x.M) N =_{\beta} M[x \mapsto N] \quad (\beta\text{-conversion})$$

$$\lambda x.M x =_{\eta} M \quad (\eta\text{-conversion})$$

Alpha Congruence

Definition

A **changed of bound variables** in M is to replace a subterm $\lambda x.N$ of M by $\lambda y.(N[x \mapsto y])$ where y does not occur in N .

Definition

A λ -term M is **α -congruent** with N , denoted by $M \equiv_{\alpha} N$, iff N results from M by a finite (perhaps empty) series of changes of bound variables.

Example

Whiteboard.

Alpha Congruence

Theorem

The relation \equiv_α is an equivalence relation.*

Convention

Following Barendregt [(1981) 2004, Convention 2.1.12], we syntactically identified λ -terms that are α -congruent, that is,

$$M \equiv N := M \equiv_\alpha N.$$

*See, e.g. [Hindley and Seldin 2008, Lemma 1.19b].

Compatible Relations

Definition

A binary relation R on Λ is **compatible** iff*

$$(M, N) \in R \quad \Rightarrow \quad \begin{cases} (P M, P N) \in R, \\ (M P, N P) \in R, \\ (\lambda x.M, \lambda x.N) \in R. \end{cases}$$

*See, e.g. [Barendregt (1981) 2004, Definition 3.1.1i].

Beta Reduction

Definition

The binary relation β on Λ is defined by

$$\beta := \{ ((\lambda x.M) N, M[x \mapsto N]) \mid M, N \in \Lambda \}.$$

Beta Reduction

Definition

The binary relation **one step β -reduction** on Λ , denoted by \rightarrow_β , is the compatible closure of β .

The \rightarrow_β relation can be inductively defined by*

$$\frac{(M, N) \in \beta}{M \rightarrow_\beta N}$$
$$\frac{M \rightarrow_\beta N}{P M \rightarrow_\beta P N} \quad \frac{M \rightarrow_\beta N}{M P \rightarrow_\beta N P} \quad \frac{M \rightarrow_\beta N}{\lambda x. M \rightarrow_\beta \lambda x. N}$$

*See, e.g. [Barendregt (1981) 2004, Definition 3.1.5].

Beta Reduction

Definition

The binary relation **β -reduction** on Λ , denoted by \rightarrow_β , is the reflexive and transitive closure of \rightarrow_β .

The \rightarrow_β relation can be inductively defined by*

$$\frac{M \rightarrow_\beta N}{M \twoheadrightarrow_\beta N}$$
$$\frac{}{M \twoheadrightarrow_\beta M} \qquad \frac{M \twoheadrightarrow_\beta N \quad N \twoheadrightarrow_\beta P}{M \twoheadrightarrow_\beta P}$$

*See, e.g. [Barendregt (1981) 2004, Definition 3.1.5].

Beta Equality or Beta Convertibility

Definition

The binary relation **β -equality** (or **β -convertibility**) on Λ , denoted by $=_\beta$, is the equivalence relation generated by \rightarrow_β .

The $=_\beta$ relation can be inductively defined by*

$$\frac{M \rightarrow_\beta N}{M =_\beta N}$$
$$\frac{M =_\beta N}{N =_\beta M} \qquad \frac{M =_\beta N \quad N =_\beta P}{M =_\beta P}$$

*See, e.g. [Barendregt (1981) 2004, Definition 3.1.5].

Normal Forms

Definition

A **β -redex** is a λ -term of the form $(\lambda x.M) N$.

Definition

A λ -term which contains no β -redex is in **β -normal form** (β -nf).

Definition

A λ -term N is a **β -nf of M** (or M **has the β -nf M**) iff N is a β -nf and $M =_{\beta} N$.

Example

Whiteboard.

Normal Forms

Remark

Church [1935, 1936] proved that the set

$$\{ M \in \Lambda \mid M \text{ has a } \beta\text{-normal form} \}$$

is not computable* (i.e. undecidable). This was the **first** undecidable set ever.[†]

*We use the term ‘computable’ rather than ‘recursive’ following to [Soare 1996].

[†]See also [Barendregt 1990].

Combinators

Definition

A **combinator** (or **closed λ -term**) is a λ -term without free variables.

Convention

A combinator called for example `pred` will be denoted by `pred`.

Combinators

Example

Some common combinators.

$B := \lambda f g x. f (g x)$	(a composition combinator)
$B' := \lambda f g x. g (f x)$	(a reversed composition combinator)
$C := \lambda x y z. x z y$	(a permuting combinator)
$I := \lambda x. x$	(an identity combinator)
$K := \lambda x y. x$	(a projection combinator)
$M := \lambda x. x x$	(a doubling combinator)
$S := \lambda f g x. f x (g x)$	(a stronger composition combinator)
$T := \lambda x y. y x$	(a permuting combinator)
$V := \lambda x y z. z y x$	(a permuting combinator)
$W := \lambda f x. f x x$	(a doubling combinator)

Combinators

Remark

The programs in a programming language based on λ -calculus are combinators.

Remark

The combinators **K** and **S** (i.e. the combinatory logic) are a Turing-complete language.

Fixed-Point Combinators

Definition

A **fixed-point combinator** is any combinator **fix** such that for all terms M ,

$$\text{fix } M =_{\beta} M (\text{fix } M).$$

Theorem

The combinator $\mathbf{Y} := \lambda f. V V$, where $V \equiv \lambda x. f (x x)$, is a fixed-point combinator.*

Theorem

The combinator $\mathbf{U} \mathbf{U}$, where $\mathbf{U} := \lambda u x. x (u u x)$, is a fixed-point combinator.†

*According to [Hindley and Seldin 2008, p. 36], this combinator was hinted by Curry in 1929 and first published by Rosenbloom [1950]. See also [Barendregt (1981) 2004, Corollary 6.1.3].

†Defined by Turing [1937]. See, also [Barendregt (1981) 2004, Definition 6.1.4].

Recursion Using Fixed-Points

Example

An informal example using the factorial function [Peyton Jones 1987, § 2.4.1].

$$\begin{aligned}\text{fac} &:= \lambda n. \text{if } (n == 0) \text{ then } 1 \text{ else } n * \text{fac } (n - 1) && \text{(combinator)} \\ &\equiv \lambda n. (\dots \text{fac} \dots) && \text{(recursive combinator)} \\ &\equiv (\lambda f. \lambda n. (\dots f \dots)) \text{fac} && \text{(\lambda-abstraction on fac)}\end{aligned}$$

Recursion Using Fixed-Points

Example

An informal example using the factorial function [Peyton Jones 1987, § 2.4.1].

$$\begin{aligned}\text{fac} &:= \lambda n. \text{if } (n == 0) \text{ then } 1 \text{ else } n * \text{fac } (n - 1) && \text{(combinator)} \\ &\equiv \lambda n. (\dots \text{fac} \dots) && \text{(recursive combinator)} \\ &\equiv (\lambda f. \lambda n. (\dots f \dots)) \text{fac} && \text{(\lambda-abstraction on fac)}\end{aligned}$$

Now, we can redefine the factorial function using **fix**.

$$\text{h} := \lambda f. \lambda n. (\dots f \dots) \quad \text{(non-recursive combinator)}$$

$$\text{fac} := \text{fix h} \quad \text{(\text{fac} is a fixed-point of h)}$$







(continued on next slide)

Recursion Using Fixed-Points






Example (continuation)

$$\begin{aligned}\text{fac } 1 &\equiv \text{fix } h \ 1 \\ &=_{\beta} h \ (\text{fix } h) \ 1 \\ &\equiv (\lambda f. \lambda n. (\dots f \dots)) \ (\text{fix } h) \ 1 \\ &\rightarrow_{\beta} \text{if } (1 == 0) \text{ then } 1 \text{ else } 1 * (\text{fix } h \ 0) \\ &\rightarrow_{\beta} 1 * (\text{fix } h \ 0) \\ &=_{\beta} 1 * (h(\text{fix } h) \ 0) \\ &\equiv 1 * ((\lambda f. \lambda n. (\dots f \dots)) \ (\text{fix } h) \ 0) \\ &\rightarrow_{\beta} 1 * (\text{if } (0 == 0) \text{ then } 1 \text{ else } 1 * (\text{fix } h \ (-1))) \\ &\rightarrow_{\beta} 1 * 1 \\ &\rightarrow_{\beta} 1\end{aligned}$$

References

-  Barendregt, H. P. [1981] (2004). The Lambda Calculus. Its Syntax and Semantics. Revised edition, 6th impression. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier (cit. on pp. 17, 18, 20–22, 28).
-  Barendregt, Henk (1990). Functional Programming and Lambda Calculus. In: Handbook of Theoretical Computer Science. Ed. by van Leeuwen, J. Vol. B. Formal Models and Semantics. MIT Press. Chap. 7. DOI: 10.1016/B978-0-444-88074-1.50012-3 (cit. on p. 24).
-  Church, Alonzo (1932). A Set of Postulates for the Foundation of Logic. Annals of Mathematics 33.2, pp. 346–366. DOI: 10.2307/1968337 (cit. on pp. 4–6).
-  — (1935). An Unsolvable Problem of Elementary Number Theory. Preliminar Report (Abstract). Bulletin of the American Mathematical Society 41.5, pp. 332–333. DOI: 10.1090/S0002-9904-1935-06102-6 (cit. on p. 24).
-  — (1936). An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58.2, pp. 345–363. DOI: 10.2307/2371045 (cit. on p. 24).
-  Hindley, J. Roger and Seldin, Jonathan P. (2008). Lambda-Calculus and Combinators. An Introduction. Cambridge University Press (cit. on pp. 13, 17, 28).

References

-  Peyton Jones, Simon L. (1987). The Implementation of Functional Programming Languages. Series in Computer Sciences. Prentice-Hall International (cit. on pp. 29, 30).
-  Pierce, Benjamin C. (2002). Types and Programming Languages. MIT Press (cit. on p. 8).
-  Rosenbloom, Paul C. (1950). The Elements of Mathematical Logic. Dover Publications (cit. on p. 28).
-  Soare, Robert I. (1996). Computability and Recursion. The Bulletin of Symbolic Logic 2.3, pp. 284–321. DOI: [10.2307/420992](https://doi.org/10.2307/420992) (cit. on p. 24).
-  Turing, A. M. (1937). The μ -Function in λ - K -Conversion. The Journal of Symbolic Logic 4.2, p. 164. DOI: [10.2307/2268281](https://doi.org/10.2307/2268281) (cit. on p. 28).