# Ordinals and Typed Lambda Calculus

## Lambda Calculus

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2018-2

(Last modification: 28th December 2024)

# 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 $\lambda$-calculus in the foundation of mathematics.
- Intended for studying functions and recursion.
- Computability model.
- A free-type functional programming language.
- $\lambda$-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.' [Chu1932, 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.' [Chu1932, 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.' [Chu1932, p. 352]

# Lambda Terms

## Definition

Let $V$ be a denumerable set of variables. The set of **$\boldsymbol{\lambda}$-terms**, denoted by $\Lambda$, is inductively defined by

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

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

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

# Lambda Terms

## Remark

Usually, the set of $\lambda$-terms $\Lambda$ is defined by the abstract grammar*

$$\begin{aligned} \Lambda \ni t ::= {} & x && \text{(variable)} \\ \mid {} & t\, t && \text{(application)} \\ \mid {} & \lambda x.t && \text{($\lambda$-abstraction)} \end{aligned}$$

*See, e.g. [Pie2002].

# Lambda Terms

## Notation

The symbol '$\equiv$' denotes the syntactic identity.

## Conventions

- $\lambda$-term variables will be denoted by $x, y, z, \ldots$.
- $\lambda$-terms will be denoted by $M, N, P, Q, \ldots$.

# 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,

$$M\, N_1\, N_2 \ldots N_k := (\ldots ((M\, N_1)\, N_2) \ldots N_k).$$

- Lambda abstraction associates to the right, that is,

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

# Lambda Terms

## Example

Using the conventions and syntactic sugar.

$$
\begin{aligned}
&(\lambda\, x\, y\, z.\, x\, z\, (y\, z))\, u\, v\, w \\
&\equiv (\lambda\, x\, y\, z.\, (x\, z)\, (y\, z))\, u\, v\, w && \text{(left-associative application)} \\
&\equiv ((\lambda\, x\, y\, z.\, (x\, z)\, (y\, z))\, u)\, v\, w && \text{(left-associative application)} \\
&\equiv (((\lambda\, x\, y\, z.\, (x\, z)\, (y\, z))\, u)\, v)\, w && \text{(left-associative application)} \\
&\equiv (((\lambda\, x\, y\, z.\, ((x\, z)\, (y\, z)))\, u)\, v)\, w && \text{(application higher precedence)} \\
&\equiv (((\lambda\, x.\, \lambda\, y.\, \lambda\, z.\, ((x\, z)\, (y\, z)))\, u)\, v)\, w && \text{(right-associative } \lambda\text{-abstraction)} \\
&\equiv (((\lambda\, x.\, \lambda\, y.\, (\lambda\, z.\, ((x\, z)\, (y\, z))))\, u)\, v)\, w && \text{(right-associative } \lambda\text{-abstraction)} \\
&\equiv (((\lambda\, x.\, (\lambda\, y.\, (\lambda\, z.\, ((x\, z)\, (y\, z)))))\, u)\, v)\, w && \text{(right-associative } \lambda\text{-abstraction)} \\
&\equiv ((((\lambda\, x.\, (\lambda\, y.\, (\lambda\, z.\, ((x\, z)\, (y\, z)))))\, u)\, v)\, w) && \text{(remove outermost parentheses)}
\end{aligned}
$$

# Binding

### Definition

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

### Definition

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

$$\begin{aligned}
\mathrm{FV}(x) &\coloneqq \{x\}, \\
\mathrm{FV}(M\,N) &\coloneqq \mathrm{FV}(M) \cup \mathrm{FV}(N), \\
\mathrm{FV}(\lambda x.M) &\coloneqq \mathrm{FV}(M) - \{x\}.
\end{aligned}$$

# Substitution

## Definition

The result of **substituting** $\boldsymbol{N}$ **for every free occurrence of** $\boldsymbol{x}$ **in** $\boldsymbol{M}$, and changing bound variables to avoid clashes, denoted by $M[\,x \mapsto N\,]$, is defined by [HS2008, Definition 1.12]

$$
\begin{aligned}
x[\,x \mapsto N\,] &:= N; && \\
y[\,x \mapsto N\,] &:= y, && \text{if } y \not\equiv x; \\
(P\,Q)[\,x \mapsto N\,] &:= P[\,x \mapsto N\,]\,Q[\,x \mapsto N\,]; && \\
(\lambda x.P)[\,x \mapsto N\,] &:= \lambda x.P; && \\
(\lambda y.P)[\,x \mapsto N\,] &:= \lambda y.P, && \text{if } y \not\equiv x \text{ and } x \notin \mathrm{FV}(P); \\
(\lambda y.P)[\,x \mapsto N\,] &:= \lambda y.P[\,x \mapsto N\,], && \text{if } y \not\equiv x, x \in \mathrm{FV}(P) \text{ and } y \notin \mathrm{FV}(N); \\
(\lambda y.P)[\,x \mapsto N\,] &:= \lambda z.P[\,x \mapsto N\,][\,y \mapsto z\,], && \text{if } y \not\equiv x, x \in \mathrm{FV}(P) \text{ and } y \in \mathrm{FV}(N);
\end{aligned}
$$

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

# Substitution

## Example

$(y\,(\lambda\,v.\,x\,v))[\,x \mapsto (\lambda\,y.\,v\,y)\,] \equiv y\,(\lambda\,z.\,(\lambda\,y.\,v\,y)\,z)$ (with $z \not\equiv v, y, x$).

# Conversion Rules

## Introduction

The functional behaviour of the $\lambda$-calculus is formalised through of their conversion rules:

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

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

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

# 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 $\lambda$-term $M$ is **$\alpha$-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 $\equiv_\alpha$ is an equivalence relation.*

## Convention

Following Barendregt [Bar2004, Convention 2.1.12], we syntactically identified $\lambda$-terms that are $\alpha$-congruent, that is,

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

*See, e.g. [HS2008, Lemma 1.19b].

# Compatible Relations

## Definition

A binary relation $R$ on $\Lambda$ 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. [Bar2004, Definition 3.1.1i].

# Beta Reduction

## Definition

The binary relation $\beta$ on $\Lambda$ 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 $\beta$-reduction** on $\Lambda$, denoted by $\to_\beta$, is the compatible closure of $\beta$.

The $\to_\beta$ relation can be inductively defined by*

$$\frac{(M, N) \in \beta}{M \to_\beta N}$$

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

*See, e.g. [Bar2004, Definition 3.1.5].

# Beta Reduction

## Definition

The binary relation **$\beta$-reduction** on $\Lambda$, denoted by $\twoheadrightarrow_\beta$, is the reflexive and transitive closure of $\rightarrow_\beta$.

The $\twoheadrightarrow_\beta$ relation can be inductively defined by*

$$\frac{M \rightarrow_\beta N}{M \twoheadrightarrow_\beta N}$$

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

*See, e.g. [Bar2004, Definition 3.1.5].

# Beta Equality or Beta Convertibility

## Definition

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

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

$$\frac{M \twoheadrightarrow_\beta N}{M =_\beta N}$$

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

*See, e.g. [Bar2004, Definition 3.1.5].

# Normal Forms

## Definition

A $\boldsymbol{\beta}$**-redex** is a $\lambda$-term of the form $(\lambda x.M)\,N$.

## Definition

A $\lambda$-term which contains no $\beta$-redex is in $\boldsymbol{\beta}$**-normal form** ($\beta$-nf).

## Definition

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

## Example

Whiteboard.

# Normal Forms

## Remark

Church [Chu1935; Chu1936] 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 [Soa1996].

†See also [Bar1992].

# Combinators

## Definition

A **combinator** (or **closed $\boldsymbol{\lambda}$-term**) is a $\lambda$-term without free variables.

## Convention

A combinator called for example $\mathrm{pred}$ will be denoted by pred.

# Combinators

## Example

Some common combinators.

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

# Combinators

### Remark

The programs in a programming language based on $\lambda$-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 $\mathsf{fix}$ such that for all terms $M$,

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

## Theorem

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

## Theorem

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

*According to [HS2008, p. 36], this combinator was hinted by Curry in 1929 and first published by Rosenbloom [Ros1950]. See also [Bar2004, Corollary 6.1.3].

†Defined by Turing [Tur1937]. See, also [Bar2004, Definition 6.1.4].

# Recursion Using Fixed-Points

## Example

An informal example using the factorial function [Pey1987, § 2.4.1].

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

# Recursion Using Fixed-Points

## Example

An informal example using the factorial function [Pey1987, § 2.4.1].

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

Now, we can redefine the factorial function using fix.

$$\mathsf{h} := \lambda\, f.\, \lambda\, n.\, (\ldots f \ldots) \qquad \text{(non-recursive combinator)}$$

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

(continued on next slide)

# Recursion Using Fixed-Points

Example (continuation)

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

# References

[Bar2004] H. P. Barendregt. The Lambda Calculus. Its Syntax and Semantics. Revised edition, 6th impression. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier, 2004 (1981) (cit. on pp. 17, 18, 20–22, 28).

[Bar1992] Henk Barendregt. Functional Programming and Lambda Calculus. In: Handbook of Theoretical Computer Science. Volume B. Formal Models and Semantics. Ed. by J. van Leeuwen. Second impression. MIT Press, 1992 (1990). Chap. 7. DOI: 10.1016/B978-0-444-88074-1.50012-3 (cit. on p. 24).

[Chu1932] Alonzo Church. A Set of Postulates for the Foundation of Logic. Annals of Mathematics 33.2 (1932), pp. 346–366. DOI: 10.2307/1968337 (cit. on pp. 4–6).

[Chu1935] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. Preliminar Report (Abstract). Bulletin of the American Mathematical Society 41.5 (1935), pp. 332–333. DOI: 10.1090/S0002-9904-1935-06102-6 (cit. on p. 24).

[Chu1936] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58.2 (1936), pp. 345–363. DOI: 10.2307/2371045 (cit. on p. 24).

# References


[HS2008] J. Roger Hindley and Jonathan P. Seldin. Lambda-Calculus and Combinators. An Introduction. Cambridge University Press, 2008 (cit. on pp. 13, 17, 28).

[Pey1987] Simon L. Peyton Jones. The Implementation of Functional Programming Languages. Series in Computer Sciences. Prentice-Hall International, 1987 (cit. on pp. 29, 30).

[Pie2002] Benjamin C. Pierce. Types and Programming Languages. MIT Press, 2002 (cit. on p. 8).

[Ros1950] Paul C. Rosenbloom. The Elements of Mathematical Logic. Dover Publications, 1950 (cit. on p. 28).

[Soa1996] Robert I. Soare. Computability and Recursion. The Bulletin of Symbolic Logic 2.3 (1996), pp. 284–321. DOI: 10.2307/420992 (cit. on p. 24).

[Tur1937] A. M. Turing. The $\mathfrak{p}$-Function in $\lambda$-$K$-Conversion. The Journal of Symbolic Logic 4.2 (1937), p. 164. DOI: 10.2307/2268281 (cit. on p. 28).