Introduction to Lambda Calculus

Santiago Palacio Gómez

Universidad EAFIT

20 de marzo de 2013

Table of Contents

λ-calculus is a collection of formal systems based on a notation invented by Alonzo Church in the 1930s. They are used to describe how operators and functions can be combined to create other functions.

In order to provide a little intuitive concept of the notation, think of the common mathematical expression "$x - y$". This can be thought as a function $f(x)$ or $g(y)$:

$$f(x) = x - y \qquad\qquad g(y) = x - y$$

In order to provide a little intuitive concept of the notation, think of the common mathematical expression "$x - y$". This can be thought as a function $f(x)$ or $g(y)$:

$$f(x) = x - y \qquad g(y) = x - y$$

So there's a need for a notation to name those functions systematically. Church included an auxiliary symbol λ and wrote:

$$f = \lambda x.x - y \qquad g = \lambda y.x - y$$

In order to provide a little intuitive concept of the notation, think of the common mathematical expression "$x - y$". This can be thought as a function $f(x)$ or $g(y)$:

$$f(x) = x - y \qquad\qquad g(y) = x - y$$

So there's a need for a notation to name those functions systematically. Church included an auxiliary symbol λ and wrote:

$$f = \lambda x.x - y \qquad\qquad g = \lambda y.x - y$$

then $f(0)$ or $g(0)$ would become $(\lambda x.x - y)(0)$ and $(\lambda y.x - y)(0)$ respectively.

This notation can be extended to represent functions of more than 1 variable. Say we define

$$h(x, y) = x - y$$

This notation can be extended to represent functions of more than 1 variable. Say we define

$$h(x, y) = x - y$$

then we would have

$$h = \lambda xy.x - y$$

This notation can be extended to represent functions of more than 1 variable. Say we define

$$h(x, y) = x - y$$

then we would have

$$h = \lambda xy.x - y$$

However this can be rewritten in terms of previous notation using:

$$h^* = \lambda x.(\lambda y.x - y)$$

and this is what we currently name **currying**.

Table of Contents

Initially, we assume there is an **infinite** set of expressions $v_0, v_1, v_2, \ldots$ called *variables*, and a finite, infinite or empty set of expressions called *atomic constants* (Note that the term *atom* here has a different meaning from what it has in logic programming). When the sequence of *atomic constants* is empty, the system will be called *pure*, otherwise *applied*. The set of expressions called λ-terms is defined inductively:

- all variables and atomic constants, called *atoms* are λ-terms

Initially, we assume there is an **infinite** set of expressions $v_0, v_1, v_2, \ldots$ called *variables*, and a finite, infinite or empty set of expressions called *atomic constants* (Note that the term *atom* here has a different meaning from what it has in logic programming). When the sequence of *atomic constants* is empty, the system will be called *pure*, otherwise *applied*. The set of expressions called λ-terms is defined inductively:

- all variables and atomic constants, called *atoms* are λ-terms
- if M and N are λ-terms, then (MN) is a λ-term (this is called an *application*).

Initially, we assume there is an **infinite** set of expressions $v_0, v_1, v_2, \ldots$ called *variables*, and a finite, infinite or empty set of expressions called *atomic constants* (Note that the term *atom* here has a different meaning from what it has in logic programming). When the sequence of *atomic constants* is empty, the system will be called *pure*, otherwise *applied*. The set of expressions called λ-terms is defined inductively:

- all variables and atomic constants, called *atoms* are λ-terms
- if M and N are λ-terms, then (MN) is a λ-term (this is called an *application*).
- if M is a λ-term and x is a variable, then $(\lambda x.M)$ is a λ-term (this is called an *abstraction*).

- Expressions like $(x(\lambda x.(\lambda x.x)))$ are, albeit possible, discouraged since there are several occurrences of λx in one term.

- Expressions like $(x(\lambda x.(\lambda x.x)))$ are, albeit possible, discouraged since there are several occurrences of λx in one term.
- Application is left-associative. This is, writing $M_1 M_2 \ldots M_n$ is equivalent to writing $(((\ldots(M_1 M_2) M_3) \ldots M_n)$.

- Expressions like $(x(\lambda x.(\lambda x.x)))$ are, albeit possible, discouraged since there are several occurrences of λx in one term.
- Application is left-associative. This is, writing $M_1 M_2 \ldots M_n$ is equivalent to writing $(((\ldots(M_1 M_2) M_3) \ldots M_n)$.
- Writing $\lambda x_1, x_2, \ldots, x_n.M$ is equivalent to writing $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots)))$.

- Expressions like $(x(\lambda x.(\lambda x.x)))$ are, albeit possible, discouraged since there are several occurrences of λx in one term.
- Application is left-associative. This is, writing $M_1 M_2 \ldots M_n$ is equivalent to writing $(((\ldots(M_1 M_2) M_3) \ldots M_n)$.
- Writing $\lambda x_1, x_2, \ldots, x_n.M$ is equivalent to writing $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots)))$.
- Application has higher precedence than abstraction, so $\lambda x.PQ \equiv \lambda x.(PQ)$, not $(\lambda x.P)Q$. (the symbol $\equiv$ means **Syntactic equivalence**).

In general, (MN) is interpreted as applying M to the argument N. Another common notation for this is $M(N)$ however (MN) is the standard way in λ-calculus.

In general, (MN) is interpreted as applying M to the argument N. Another common notation for this is $M(N)$ however (MN) is the standard way in λ-calculus.

The terms in the form $(\lambda x.M)$ represents the function whose value with an argument N is calculated by substituting N for x in M.

In general, (MN) is interpreted as applying M to the argument N. Another common notation for this is $M(N)$ however (MN) is the standard way in λ-calculus.

The terms in the form $(\lambda x.M)$ represents the function whose value with an argument N is calculated by substituting N for x in M.

For example, $(\lambda x.x)(a)$ is the result of applying the function identity to the atom a.

Table of Contents

For any M, N, x, we define $[N/x]M$ to be the result of substituting N for every **free** occurrence of x in M, and changing bound variables to avoid clashes. This definition can be extended to several **simultaneous** substitution $[N_1/x_1, N_2/X_2 \ldots N_n/X_n]$, but it must not be confused with several consecutive substitutions.

For any M, N, x, we define $[N/x]M$ to be the result of substituting N for every **free** occurrence of x in M, and changing bound variables to avoid clashes. This definition can be extended to several **simultaneous** substitution $[N_1/x_1, N_2/X_2 \dots N_n/X_n]$, but it must not be confused with several consecutive substitutions.

This definition is similar to that given in course, however the notation is opposite, here $[N/x]$ is similar to what $[x/N]$ is in the course.

The precise inductive definition is:

$$[N/x]x \equiv N$$

The precise inductive definition is:

$$
\begin{aligned}
[N/x]x &\equiv N \\
[N/x]a &\equiv a && \text{for all atoms } a \not\equiv x
\end{aligned}
$$

The precise inductive definition is:

$$
\begin{array}{lll}
[N/x]x & \equiv N & \\
[N/x]a & \equiv a & \text{for all atoms } a \not\equiv x \\
[N/x](PQ) & \equiv ([N/x]P[N/x]Q) &
\end{array}
$$

The precise inductive definition is:

$$
\begin{array}{lll}
[N/x]x & \equiv N & \\
[N/x]a & \equiv a & \text{for all atoms } a \not\equiv x \\
[N/x](PQ) & \equiv ([N/x]P[N/x]Q) & \\
[N/x](\lambda x.P) & \equiv (\lambda x.P) &
\end{array}
$$

The precise inductive definition is:

$$
\begin{array}{lll}
[N/x]x & \equiv N & \\
[N/x]a & \equiv a & \text{for all atoms } a \not\equiv x \\
[N/x](PQ) & \equiv ([N/x]P[N/x]Q) & \\
[N/x](\lambda x.P) & \equiv (\lambda x.P) & \\
[N/x](\lambda y.P) & \equiv (\lambda y.P) & \text{if } x \notin FV(P)
\end{array}
$$

The precise inductive definition is:

$$
\begin{array}{lll}
[N/x]x & \equiv N & \\
[N/x]a & \equiv a & \text{for all atoms } a \not\equiv x \\
[N/x](PQ) & \equiv ([N/x]P[N/x]Q) & \\
[N/x](\lambda x.P) & \equiv (\lambda x.P) & \\
[N/x](\lambda y.P) & \equiv (\lambda y.P) & \text{if } x \notin FV(P) \\
[N/x](\lambda y.P) & \equiv (\lambda y.[N/x]P) & \text{if } x \in FV(P) \text{ and } y \notin FV(N)
\end{array}
$$

The precise inductive definition is:

$$
\begin{array}{lll}
[N/x]x & \equiv N & \\
[N/x]a & \equiv a & \text{for all atoms } a \not\equiv x \\
[N/x](PQ) & \equiv ([N/x]P[N/x]Q) & \\
[N/x](\lambda x.P) & \equiv (\lambda x.P) & \\
[N/x](\lambda y.P) & \equiv (\lambda y.P) & \text{if } x \notin FV(P) \\
[N/x](\lambda y.P) & \equiv (\lambda y.[N/x]P) & \text{if } x \in FV(P) \text{ and } y \notin FV(N) \\
[N/x](\lambda y.P) & (\lambda z.[N/x][z/y]P) & \text{if } x \in FV(P) \text{ and } y \in FV(N)
\end{array}
$$

where $FV(P)$ is the set of free variables present in P.

We call an α-conversion of P to be the change of a bound variable in P. For example, if we have the expression $\lambda x.M$, and let $y \notin FV(M)$. Then $\lambda y.[y/x]M$ is called an α-conversion.
Iff P can be changed to Q by a finite series α-conversions then we say P is **congruent** to Q, or P α-converts to Q, which is equivalent to write

$$P \equiv_\alpha Q$$

This relation results being reflexive, transitive and symmetric.

A term of the form $(\lambda x.M)N$ represents an operator $.M$ *applied* to an argument N. Informally, it the value that results of substituting N for x in M, so $(\lambda x.M)N$ can be 'simplified' to $[N/x]M$.

Any term in the form

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

is called a β-*redex*, and the corresponding term

$$[N/x]M$$

is called its *contractum*.

If a term P contains a β-redex and we replace it to its contractum, getting the result P', then we say that we have *contracted* the redex-occurrence in P, and P *β-contracts* to P', and we note:

$$P \rhd_{1\beta} P'$$

If P can be changed to Q by a finite series of β-contractions and changes of bound variables, we say P *β-reduces* to Q, or

$$P \rhd_{\beta} Q$$

Note that the result of a β-contraction does not always "simplifies" the expression.

Note that the result of a β-contraction does not always "simplifies" the expression.

- $(\lambda x.xx)(\lambda x.xx)$

Note that the result of a β-contraction does not always "simplifies" the expression.

- $(\lambda x.xx)(\lambda x.xx)$
- $(\lambda x.xxy)(\lambda x.xxy)$

Note that the result of a β-contraction does not always "simplifies" the expression.

- $(\lambda x.xx)(\lambda x.xx)$
- $(\lambda x.xxy)(\lambda x.xxy)$

In the first example, the result of applying the β-contraction results is the same expression. In the second example we observe this pattern

$$\begin{aligned}(\lambda x.xxy)(\lambda x.xxy) &\triangleright_\beta (\lambda x.xxy)(\lambda x.xxy)y \\ &\triangleright_\beta (\lambda x.xxy)(\lambda x.xxy)yy \\ &\ldots etc.\end{aligned}$$

So the 'simplification' process might actually complicate the expression.

β-normal form

β-nf

A term Q which contains no β-redexes is called a *β-normal form*, or a β-nf. The class of all β-normal forms is called β-nf or $\lambda\beta$-nf. If a term P β-reduces to a term Q in β-nf, then Q is called a β-normal form of P.
As we saw before, not every expression has a β-nf.

The Church-Rosser theorem states that if $P \triangleright_\beta M$ and $P \triangleright_\beta N$ then there exists a term T such that $M \triangleright_\beta T$ and $N \triangleright_\beta T$. In general, this property is called *confluence*, so this statement can be reduced to *β-reduction is confluent.*
This theorems proves that if any statement P has a normal form, then it is unique modulo $\equiv_\alpha$.

β-equality

We say P is β-equal or β-convertible to Q (noted $P =_\beta Q$) iff Q can be obtained from P by a finite sequence of β-contractions, reversed β-contractions and changes of bound variables. That is, $P =_\beta Q$ iff $\exists(P_1, P_2, \ldots, P_n)$ such that

$$P_1 = P$$

$$P_n = Q$$

$$P_i \rhd_b P_{i+1} \vee P_{i+q} \rhd_\beta P_i \vee P_1 \equiv_\alpha P_{i+1}$$

Table of Contents

All the information was taken from the book:
Lambda-Calculus and Combinators: An Indtroduction, by
J.Roger Hindley and Jonathan P. Seldin
Cambridge