# An Introduction to Lambda Calculus and Functional Programming

Rodrigo Machado
*Instituto de Informática*
*Universidade Federal do Rio Grande do Sul*
*Porto Alegre, Brasil*
*rma@inf.ufrgs.br*

***Abstract*—Lambda calculus is a formal system built around the concepts of function definition and function application. It is a minimalistic Turing-computable system that has a great influence on the design of functional programming. This paper is a tutorial on the untyped lambda calculus and its use as an idealised programming language.**

***Keywords*-Lambda Calculus, Functional Programming.**

## I. Introduction

The lambda calculus is a formal system based around the central ideas of function definition and function application. The pure, untyped version of the calculus was originally proposed by Church in the 1930s, and although very small, it is as expressive as Turing machines. Moreover, the system is quite influential among programming language design, inspiring several languages that follow the *functional programming paradigm* such as LISP, Ocaml, Haskell, among others. Due to its influence, it is important for computer scientists to be acquainted with its basic ideas.

This paper is based on a short course presented at the second edition of WEIT (Workshop-School on Theoretical Computer Science), which occurred in October 15-17th, 2013 in Rio Grande, Brasil. Its aim is to present an overview of the untyped lambda calculus, with focus on the concepts rather than on proofs. All the material covered here is standard and can be found in the literature. For the untyped version, the main reference is [1]. For typed variations of the calculus and their connection to mathematical logic, we recommend [3] and [4]. The relation between lambda calculus and programming languages is explored in more detail in [2]. The textbook [5] is a good introduction to functional programming.

The structure of the this paper is as follows: Section II introduces the syntactical aspects of lambda calculus. Section III explains the distinction between free and bound variables. Section IV explains the substitution operation and its subtleties. Section V describes $\alpha$-equivalence, and the concept of lambda terms. Section VI describes $\beta$-reduction and associated relations. Section VII discusses how to encode several elements of programming languages such as boolean values, natural numbers, pairs, lists and recursive definitions as lambda terms. Finally, Section VIII concludes this presentation.

## II. Syntax of lambda calculus

This section explain the syntax of lambda pre-terms. Initially, we must choose a countable infinite set $Var$ of names (also referred as variables). The choice of which names are in $Var$ is completely arbitrary, and we denote them by lowercase letters such as $x$, $y$ and $z$. Based on $Var$, we define a set of what we call *pre-terms*, as follows.

**Definition 1** (Pre-terms)**.** *Let $Var$ be a countable set of names. The* set of pre-terms $\Lambda^-$ *is the smallest set such that*

$$\frac{x \in Var}{x \in \Lambda^-} \quad (\text{VAR})$$

$$\frac{M \in \Lambda^- \qquad N \in \Lambda^-}{@(M,N) \in \Lambda^-} \quad (\text{APP})$$

$$\frac{x \in Var \qquad M \in \Lambda^-}{\lambda x.M \in \Lambda^-} \quad (\text{ABS})$$

*We read $\frac{\phi_1,\ldots,\phi_n}{\psi}$ as* if $\phi_1$ and $\phi_2$ and $\ldots$ and $\phi_n$ then $\psi$. *The uppercase letters $M$, $N$, $P$, $\ldots$ will be used to denote elements of the set $\Lambda^-$.*

**Example 1** (Pre-terms)**.** *Below we find some examples of pre-terms:*

| | | |
|---|---|---|
| $x$ | $\lambda x.y$ | $\lambda x.@(x,y)$ |
| $@(x,y)$ | $@((\lambda x.x),y)$ | $@(z,(\lambda x.y))$ |

A pre-term with the format $@(M,N)$ aims to mean "call function $M$ passing $N$ as its argument". The notation $\lambda x.M$ means "a function that receives one parameter, which will internally be called $x$, and which returns expression $M$ as result". For instance, if natural numbers and arithmetic operations were part of the language, we would write

$$\lambda x.x + 1$$

to denote the function that maps each natural number to its successor. We use

$$@(\ (\lambda x.x + 1),\ 3\ )$$

978-1-4799-3057-9/13 $31.00 © 2013 IEEE
DOI 10.1109/WEIT.2013.40

to denote the application of the successor function to the number 3. Notice that the version of the lambda calculus we are describing here does not contain primitive numbers and operations as we have used above. They, however, can be represented using terms, as it will be seen later in Section VII.

One important notational aspect is that mostly often white spaces are used to denote function application, i.e, we write $M\ N$ instead of $@(M, N)$. This way, the previous example of applying the successor function to 3 becomes

$$(\lambda x.x + 1)\ 3$$

This style, however, introduces ambiguity in the representation. For instance, now

$$M\ N\ P$$

could mean both $@(M, @(N, P))$ or $@(@(M, N), P)$. We solve this issue by stating that parentheses may be used to indicate the distinction, as in

$$(M(N\ P)) \neq ((M\ N)\ P)$$

and, in the case they are absent, we state that the following rules apply:

1) $\lambda$'s scope extend to the rightmost term, until "stopped" by parentheses. For example,

$$\lambda x.x\ y = \lambda x.(x\ y) \neq (\lambda x.x)\ y$$

2) The @ operator is left-associative. For example,

$$x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$$

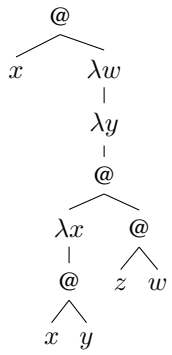
An additional shorthand notation allows us simplify a sequence of lambda abstractions, as shown in the example below.

$$\lambda x\ y\ z.M = \lambda x.\lambda y.\lambda z.M$$

**Example 2.** *According the the rules of notation, the pre-term written*

$$x\ \lambda w\ y.(\lambda x.x\ y)\ (z\ w)$$

*is actually the tree shown below:*




## III. Bound and free variables

The role of a variable within a pre-term depends on its relative position. This section distinguishes what we call *free* and *bound* variables. We start by characterizing the *scope* of a lambda abstraction.

**Definition 2** (Scope)**.** *Consider a pre-term with the format $\lambda x.M$. We say that $M$ is the* scope *of the formal parameter $\lambda x$.*

**Definition 3** (Bound and free variables)**.** *Every occurrence of a variable $x$ within the scope of $\lambda x$ is said to be a* bound occurrence*. Otherwise, the variable occurrence is said to be* free*.*

**Example 3** (Bound and free variables)**.**

| | | |
|---|---|---|
| *(a)* | $\lambda x.x\ y$ | *$x$ is bound, $y$ is free* |
| *(b)* | $\lambda x.z\ \lambda z.z\ x$ | *the leftmost $z$ is free, the rightmost $z$ and $x$ are bound* |
| *(c)* | $\lambda x.x\ \lambda x.x\ y$ | *both occurrences of $x$ are bound, $y$ is free* |

*Notice that a variable may occur both bound and free in the same term, as in item (b). If a variable $x$ occurs within the scope of two $\lambda x$'s, we say that it always binds to the innermost $\lambda x$ symbol (considering the term as a tree). For instance, in pre-term (c), the rightmost $x$ binds to the rightmost $\lambda x$ instead of the leftmost $\lambda x$.*

The following operation obtains the set of free variables of a pre-term.

**Definition 4.** *The function $FV : \Lambda^- \rightarrow Pow(Var)$ is defined as follows.*

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

**Definition 5.** *A pre-term $M$ where $FV(M) = \varnothing$ is said to be* closed*. Otherwise, it is* open*.*

It is important to notice that free and bound occurrences of variables have distinct meanings. Free variables in a pre-term may be interpreted as global, distinguishable names. Bound variables, on the other hand, may be seen as simply placeholders for actual arguments. For example, the following two expressions are clearly distinct, since the names $e$ and $i$ refer to a global interpretation (numeric constants).

$$42 + e^2$$
$$42 + i^2$$

In the following function definitions, however, $e$ and $i$ act as formal parameters, and thus they represent a reference for the value the function receives. Therefore, the same function

is being defined.

$$f(i) = 42 + i^2$$
$$f(e) = 42 + e^2$$

## IV. Substitution

The fundamental principle behind function application is the substitution of formal parameters by the actual parameters. For instance, the expected effect of the application $(\lambda x.x + 1)\ 3$ is to substitute $x$ for 3 within the scope (also called body) of the function, which results in $3 + 1$. The syntax

$$M[x := N]$$

means the result of substituting all *free occurrences* of $x$ inside $M$ by the pre-term $N$.

**Example 4** (Substitution)**.**

| | | | |
|---|---|---|---|
| *(1)* | $(\lambda x.x\ y)[x := w]$ | $=$ | $\lambda x.x\ y$ |
| *(2)* | $(\lambda x.x\ y)[y := w]$ | $=$ | $\lambda x.x\ w$ |
| *(3)* | $(\lambda x.x\ y)[z := w]$ | $=$ | $\lambda x.x\ y$ |
| *(4)* | $(z\ \lambda z.z)[z := w]$ | $=$ | $w\ \lambda z.z$ |
| *(5)* | $(z\ z)[z := \lambda z.z\ z]$ | $=$ | $(\lambda z.z\ z)(\lambda z.z\ z)$ |
| *(6)* | $(\lambda x.x\ y)[y := x]$ | $=$ | $(\lambda x.x\ x)?$ |

*Notice that in (1) the substitution did not cause any effect because $x$ does not occur free. The substitution (6) contains a question mark because, although mechanically it is the result of substituting a free occurrence of $y$ by the term $x$, in fact it modified the meaning of the original lambda abstraction, and it should not be considered to be a correct substitution.*

Let us focus on the substitution $(6)$ of the previous example:

$$\overbrace{(\lambda x.xy)}^{M}\ [y := \overbrace{x}^{N}]$$

Within $M$, notice that the name $x$ is bound in the position of variable $y$. In $N$, however, the name $x$ occurs free. When we substitute $y$ by $N$, we put a free $x$ in a position where the same name $x$ is bound. Since free and bound variables have distinct meanings, we are in fact transforming a free occurrence of $x$ into a bound occurrence of $x$. This problem is called *capture of free variable*, and the following formal definition of substitution successfully avoids it.

**Definition 6** (Substitution)**.**

$$x[y := P] = \begin{cases} P & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

$$(\lambda x.M)[y := P] = \begin{cases} \lambda x.M & \text{if } x = y \\ \lambda x.(M[y := P]) & \text{if } x \neq y \text{ and } x \notin FV(P) \end{cases}$$

$$(M\ N)[y := P] = M[y := P]\ \ N[y := P]$$

According to this definition of substitution, however, the result of $(\lambda x.x\ y)[y := x]$ is undefined.

## V. Alpha equivalence

This section focus on the notion of $\alpha$-equivalence. Let us start considering the following two pre-terms:

$$\lambda x.z\ x$$
$$\lambda y.z\ y$$

Notice that both terms specify the same effect over its parameter (it applies the name $z$ as a function over it). However, the two terms are syntactically distinct since their parameters are called, respectively, $x$ and $y$. The idea behind $\alpha$-equivalence is that terms which differ only by an arbitrary choice of *bound names* should be considered equivalent to each other. We write $M =_\alpha N$ to denote that $M$ and $N$ are $\alpha$-equivalent, and $M \neq_\alpha N$ to denote they are not.

**Definition 7** ($\alpha$-equivalence)**.** $=_\alpha$ *is the smallest equivalence relation on $\Lambda^-$ such that*

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

$$\frac{M =_\alpha M'}{M\ P =_\alpha M'\ P}$$

$$\frac{M =_\alpha M'}{P\ M =_\alpha P\ M'}$$

$$\frac{M =_\alpha M'}{\lambda x.M =_\alpha \lambda x.M'}$$

The axiom $(\alpha)$ states the idea of equivalence of terms by change of bound names. The other three rules state that $\alpha$-equivalence of subterms extend to the whole term.

**Example 5** ($\alpha$-equivalence)**.** *Below we find some examples and counterexamples of $\alpha$-equivalence.*

$$\lambda x.x\ y =_\alpha \lambda z.z\ y$$
$$\lambda x.x\ y \neq_\alpha \lambda y.y\ y$$
$$\lambda x.x\ x =_\alpha \lambda z.z\ z$$
$$\lambda y.z\ y \neq_\alpha \lambda y.x\ y$$

We now state that terms are actually the collection of all distinct but $\alpha$-equivalent pre-terms, i.e. the actual operation that the pre-terms are indicating, without the focus on the particular choice of parameter names.

**Definition 8** (Terms)**.** *A lambda* term *is an equivalence class of $\alpha$-equivalent pre-terms. We will write $\{M\}_\alpha$ to denote the term containing the pre-term $M$. We denote by $\Lambda$ the set of all lambda terms.*

**Example 6** (Terms)**.** *Below we see the definition of some common terms.*

$$\begin{aligned} \boldsymbol{x} &= \{x\} \\ \boldsymbol{I} &= \{\ \lambda x.x,\ \lambda y.y,\ \lambda z.z, \dots\ \} \\ \boldsymbol{\omega} &= \{\ \lambda a.a\ a,\ \lambda b.b\ b,\ \lambda c.c\ c \dots\ \} \end{aligned}$$

Operations defined on pre-terms can be extended to terms. For instance, the following rule states how substitution on pre-terms defines substitution on terms.

**Definition 9** (Substitution on terms)**.** *Let $\{M\}_\alpha$ and $\{N\}_\alpha$ be terms, and $x \in Var$ be a variable. If*

$$M[\ x := N\ ] = P$$

*for some pre-terms $M$, $N$ and $P$, then*

$$\{M\}_\alpha[\ x := \{N\}_\alpha\ ] = \{P\}_\alpha$$

The effect of this definition is that substitution on pre-terms, which was partial to avoid capture of free variables, becomes total when defined on terms, since it always possible to find an $\alpha$-equivalent term for a given substitution that avoids a fixed set of free variables.

**Example 7.** *Consider the following $\alpha$-equivalent pre-terms:*

$$(\lambda x.x\ y) =_\alpha (\lambda a.a\ y)$$

*Notice that the following substitution is defined:*

$$(\lambda a.a\ y)[y := x] = (\lambda a.a\ x)$$

*Therefore, the following substitution on terms is also defined:*

$$\{\lambda x.x\ y\}_\alpha[y := \{x\}_\alpha] = \{\lambda a.a\ x\}_\alpha$$

An important remark is that the distinction between pre-terms and terms is formally important but is rather technical, and several authors (in particular [1]) choose to use a representative pre-term $M$ to refer to the term $\{M\}_\alpha$ itself. This convention actually helps to keep the definitions clear, and from now on we will follow it, i.e., we assume there are implicit $\{\ \}_\alpha$ around pre-terms $M$, $N$, ... and we refer to them as *terms*.

## VI. Beta reduction

The notion of beta-reduction formalizes the mechanical aspect of function application by means of substitution.

**Definition 10** (Redex)**.** *A* redex *(reducible expression) is a subterm of a term $M$ with the format*

$$(\lambda x.P)\ Q$$

*for which the respective* contractum *is*

$$P[x := Q]$$

*When relevant, we will mark redexes inside terms by putting a solid line over them.*

A redex represents a part of a term where we have an abstraction being applied to another term, and therefore substitution can occur.

**Definition 11** (Normal form)**.** *A term that does not contain any redex is a* normal form*. Otherwise, we say the term is* reducible*.*

**Example 8** (Redexes and normal forms)**.**

| | | |
|---|---|---|
| $(a)$ | $\overline{(\lambda y.\lambda x.y\ x)\ \overline{((\lambda y.x)\ y)}}$ | *contains two redexes* |
| $(b)$ | $\lambda x.\lambda y.\ x\ x$ | *is a normal form* |

We say a term $M$ $\beta$-reduces to $N$ if $N$ is the result of substituting a redex in $M$ by its respective contractum, which we write $M \to_\beta N$.

**Definition 12** ($\beta$-reduction)**.** *$\beta$-reduction ($\to_\beta$) is the smallest relation on $\Lambda$ such that*

$$(\lambda x.P)\ Q \to_\beta P[x := Q] \qquad (\beta)$$

$$\frac{M \to_\beta M'}{P\ M \to_\beta P\ M'}$$

$$\frac{M \to_\beta M'}{M\ P \to_\beta M'\ P}$$

$$\frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

**Example 9** ($\beta$-reduction)**.** *Below there are some examples of $\beta$-reduction.*

$$\begin{aligned} (\lambda a.a\ b)\ x &\to_\beta x\ b \\ (\lambda a.a\ a)\ (\lambda y.y) &\to_\beta (\lambda y.y)\ (\lambda y.y) \end{aligned}$$

The $\beta$-reduction relation may be seen as a "one-step" evaluation. If we want to focus on the overall evaluation of a term, we use the "multistep" reduction $\twoheadrightarrow_\beta$.

**Definition 13.** *$\twoheadrightarrow_\beta$ is the smallest relation on terms such that*

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

$$M \twoheadrightarrow_\beta M$$

$$\frac{M \to_\beta N \qquad N \to_\beta P}{M \twoheadrightarrow P}$$

*In other words, $\twoheadrightarrow_\beta$ is the reflexive and transitive closure of $\to_\beta$.*

Another relation based on $\beta$-reduction is $\beta$-equivalence, which identifies terms that have confluent rewritings.

**Definition 14** ($\beta$-equivalence)**.** *$=_\beta$ is the smallest relation on terms such that*

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

**Example 10** ($\beta$-equivalence)**.** *The following terms are beta-equivalent:*

$$M_1 = (\lambda z.\lambda x.x\ z)\ y$$
$$M_2 = (\lambda u.\lambda x.x\ y)\ (t\ t)$$
$$M_3 = \lambda x.x\ y$$

*Notice, however, that $M_1 \not\twoheadrightarrow_\beta M_2$ and $M_2 \not\twoheadrightarrow_\beta M_1$.*

We can interpret $\beta$-equivalence as a notion of "same value". For instance, the arithmetic expression $1 + (2 + 3)$ can be evaluated as follows

$$(1 + (2 + 3)) \Rightarrow (1 + 5) \Rightarrow 6$$

The idea is that the intermediate expressions are equivalent to the value they evaluate to. Following this analogy, normal forms would pose as values, and reducible terms as numeric expressions.

**Definition 15.** *A term $M$ has a normal form when exists $P$ such that $M =_\beta P$ and $P$ is a normal form.*

Notice that reducible terms may not have normal forms:

- $x\ z$ is a normal form, therefore it has a normal form.
- $(\lambda a.a)\ x$ has a normal form since it evaluates to $x$.
- $(\lambda a.a\ a)\ (\lambda a.a\ a)$ does not have a normal form, since it $\beta$-reduces to itself.

An arbitrary term $P$ may have several redexes, which allows it to be $\beta$-reduced to distinct terms.

**Example 11.** *Consider $\boldsymbol{I} = \lambda x.x$ and $\omega = \lambda x.x\ x$. The term $\omega\ (\omega\ \boldsymbol{I})$ has two redexes and may be reduced as follows:*

$$\omega\ \underline{(\omega\ \boldsymbol{I})} \ \to_\beta \ \omega\ (\boldsymbol{I}\ \boldsymbol{I})$$
$$\underline{\omega\ (\omega\ \boldsymbol{I})} \ \to_\beta \ (\omega\ \boldsymbol{I})\ (\omega\ \boldsymbol{I})$$

**Definition 16.** *An* evaluation strategy *is a function that chooses a single redex for every reducible term.*

Usually evaluation strategies select redexes based on their relative position in the term. The two classic strategies are *lazy evaluation* and *strict evaluation*.

- *lazy evaluation* always choose the leftmost, outermost (closer to the root) redex of a term. It is associated with the idea of applying a function without evaluating its arguments.
- *strict evaluation* always choose the leftmost, innermost (closer to the leaves) redex of a term. It is associated with the idea of applying a function only after evaluating its arguments.

Those two distinct strategies may lead to a distinct behaviour when applied to the same term. For instance, consider the term $\mathbf{K}\ \mathbf{I}\ \Omega$, where $\mathbf{I} = \lambda x.x$, $\mathbf{K} = \lambda x\ y.x$ and $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$. Below we have its evaluation using the lazy strategy, which finishes in two steps.

$$(\mathbf{K}\ \mathbf{I}\ \mathbf{\Omega}) \to_\beta \ (\lambda y.\mathbf{I})\ \mathbf{\Omega} \to_\beta \mathbf{I}$$

However, if we try the strict strategy, we end up with a non-terminating evaluation.

$$(\mathbf{K}\ \mathbf{I}\ \mathbf{\Omega}) \ \to_\beta \ (\lambda y.\mathbf{I})\ \mathbf{\Omega} \ \to_\beta \ (\lambda y.\mathbf{I})\ \mathbf{\Omega} \ \to_\beta \ \cdots$$

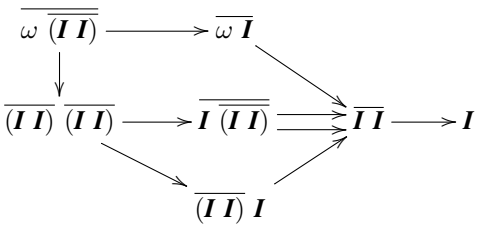
At first sight it may seen that the lambda calculus is excessively non-deterministic, since we can $\beta$-reduce the same term in more than one way. However, one of the first results of the theory actually shows that the evaluation can always converge.

**Theorem 1** (Confluence of $\beta$-reduction)**.** *if $M \to_\beta N$ and $M \to_\beta N'$ then exists $P$ such that $N \twoheadrightarrow_\beta P$ and $N' \twoheadrightarrow_\beta P$*

*Proof:* See Chapter 3, Section 3.2 of [1]. ■

This result, also known as Church-Rosser property, has a very important consequence: it is not possible for the same term to have more than one distinct normal form. Either a term does not have a normal form, or it has exactly one. For instance, this allows the safe use of normal forms to encode values such as numbers, without the risk of introducing nondeterminism in arithmetic operations.

**Example 12** (Confluence of $\beta$-reduction)**.** *Consider $\boldsymbol{I} = \lambda x.x$ and $\omega = \lambda x.x\ x$. Below we see all possibilities for $\beta$-reducing the term $\omega\ (\boldsymbol{I}\ \boldsymbol{I})$, where each arc represents the reduction of a single redex. Notice that its normal form is $\boldsymbol{I}$.*




Another important property of $\beta$-reduction is the fact that, if a term has a normal form, there is a strategy that is guaranteed to achieve it.

**Theorem 2.** *If term $M$ has normal form $P$, then lazy evaluation strategy always reach $P$ from $M$ using a finite number of $\beta$-reductions.*

*Proof:* See Chapter 1, Section 1.5 of [3]. ■

## VII. Lambda calculus and functional programming languages

The lambda calculus was a major influence on the *functional programming paradigm*, where primitive functions are the basic building blocks, and function composition is the main *program composition mechanism*. Basic elements of this style are the prevalence of recursion over iteration (as loop mechanism) and the pervasive use of inductively defined data structures such as lists. Roughly speaking, modern functional programming languages such as Haskell and Ocaml can be seen as lambda calculus + an evaluation strategy + lots of additional primitive values (integers, boolean values, strings, ...) and operations (arithmetic, relational, string manipulation, ...).

Although very small, the untyped lambda is is actually as powerful as any modern programming language regarding Turing-computability. This section will describe how it is possible to encode constructions usually found in general programming languages using only lambda terms. For example, we will see how to represent boolean values, arithmetic operations, data structures and functions defined recursively. As a rule, both values and operations will be encoded using *closed terms* to avoid dependence on a particular set of variable names.

### A. Logic operations

Let us start considering how to encode the boolean values *true* and *false*:

**Definition 17** (Church booleans)**.**

$$\textbf{\textit{true}} = \lambda x\ y\ .\ x$$
$$\textbf{\textit{false}} = \lambda x\ y\ .\ y$$

Both boolean values are encoded as selectors: terms that receive two values in sequence, and return one of them. The encoding of *true* returns the first value, while the encoding of *false* returns the second value. This choice is convenient because it makes the implementation of the *if-then-else* construction as simple as applying the boolean value over the *then* and *else* subterms.

**Definition 18** (Conditional operator)**.**

$$\textbf{\textit{if}} = \lambda b\ v\ w\ .\ b\ v\ w$$

**Example 13.** *Below there is an example of evaluation of the conditional operator using the lazy strategy.*

$$\begin{aligned}
(\textbf{\textit{if false}}\ \Omega\ \textbf{\textit{I}}) &= (\lambda b\ v\ w\ .\ b\ v\ w)\ \textbf{\textit{false}}\ \Omega\ \textbf{\textit{I}} \\
&\to_\beta (\lambda v\ w\ .\ \textbf{\textit{false}}\ v\ w)\ \ \Omega\ \textbf{\textit{I}} \\
&\to_\beta (\lambda w\ .\ \textbf{\textit{false}}\ \Omega\ w)\ \textbf{\textit{I}} \\
&\to_\beta \textbf{\textit{false}}\ \Omega\ \textbf{\textit{I}} \\
&= (\lambda x\ y\ .\ y)\ \Omega\ \textbf{\textit{I}} \\
&\to_\beta (\lambda y\ .\ y)\ \textbf{\textit{I}} \\
&\to_\beta \textbf{\textit{I}}
\end{aligned}$$

The boolean operations **not** and **and** can be easily defined by means of the conditional operator, as the next definition shows. The other boolean operations such as **or** and **xor** can be defined in terms of **not** and **and** in the traditional way.

**Definition 19** (Boolean operations)**.**

$$\textbf{\textit{not}} = \lambda a\ .\ \textbf{\textit{if}}\ a\ \textbf{\textit{false true}}$$
$$\textbf{\textit{and}} = \lambda a\ b\ .\ \textbf{\textit{if}}\ a\ b\ \textbf{\textit{false}}$$

### B. Numbers and arithmetic

Although there are many ways of encoding natural numbers as lambda terms, the following is the "standard".

**Definition 20** (Church numerals)**.**

$$\begin{aligned}
\textbf{\textit{0}} &= \lambda f\ x\ .\ x \\
\textbf{\textit{1}} &= \lambda f\ x\ .\ f\ x \\
\textbf{\textit{2}} &= \lambda f\ x\ .\ f\ (f\ x) \\
&\vdots \\
\textbf{\textit{n}} &= \lambda f\ x\ .\ \overbrace{f\ (f\ (f\ \ldots (f}^{n}\ x) \ldots))
\end{aligned}$$

A natural number $n$ is encoded as a term that receives two arguments $f$ and $x$, and apply the $f^{(n)}$ over $x$. Notice that the only two ways to actually use a number is to call it as a function, or to pass it to another function. These will be the building blocks for defining all numeric operations. For instance, consider the definition of the successor function:

**Definition 21** (Successor)**.**

$$\textbf{\textit{succ}} = \lambda n\ p\ q\ .\ p\ (n\ p\ q)$$

The rationale behind the structure of **succ** is the following:

1) Its receives a Church numeral $n$:
$$n = \lambda f\ x\ .\ \overbrace{f\ (f\ (f\ \ldots (f}^{n}\ x) \ldots)$$
2) The application $(n\ p\ q)$ changes $f$'s into $p$'s, and $x$ into $q$.
$$(n\ p\ q) = \overbrace{p\ (p\ (p\ \ldots (p}^{n}\ q) \ldots)$$
3) An extra $p$ is applied over $(n\ p\ q)$, obtaining
$$p\ (n\ p\ q) = \overbrace{p\ (p\ (p\ \ldots (p}^{n+1}\ q) \ldots)$$
4) Finally $p$ and $q$ are bound, resulting in
$$\lambda p\ q\ .\ p\ (n\ p\ q) = \lambda p\ q\ .\ \overbrace{p\ (p\ (p\ \ldots (p}^{n+1}\ q) \ldots)$$
which is the Church numeral that encodes $n + 1$.

All arithmetic operations follow a pattern more or less similar to **succ**. The next definition presents the terms that

encode addition, multiplication, exponentiation, and test of zero.

**Definition 22** (Arithmetic operations and comparisons)**.**

$$\begin{aligned} \textbf{\textit{add}} &= \lambda\ m\ n\ p\ q\ .\ m\ p\ (n\ p\ q) \\ \textbf{\textit{mult}} &= \lambda\ m\ n\ p\ q\ .\ m\ (n\ p)\ q \\ \textbf{\textit{exp}} &= \lambda\ m\ n\ .\ n\ m \\ \textbf{\textit{isZero}} &= \lambda n\ .\ n\ (\lambda x.\ \textbf{\textit{false}})\ \textbf{\textit{true}} \end{aligned}$$

*C. Structured data: pairs and lists*

There are also ways to encode data structures such as tuples and lists as lambda terms. Let us start with pairs.

**Definition 23** (Pairs)**.** *An ordered pair $(M, N)$ is represented by* ***pair*** $M\ N$ *where*

$$\textbf{\textit{pair}} = \lambda\ m\ n\ b\ .\ b\ m\ n$$

The **pair** constructor receives two arguments $m$ and $n$ to be "stored", and it returns a term which is a lambda abstraction. For instance,

$$\textbf{pair 0 true} = \lambda b.\ b\ \textbf{0 true}$$

The elements of the pair can be extracted by passing either **true** or **false** to it as a parameter. For example,

$$\begin{aligned} (\textbf{pair 0 true})\ \textbf{true} &\twoheadrightarrow_\beta \textbf{0} \\ (\textbf{pair 0 true})\ \textbf{false} &\twoheadrightarrow_\beta \textbf{true} \end{aligned}$$

This leads to straightforward implementation of the projections.

**Definition 24** (Pair projections)**.**

$$\begin{aligned} \textbf{\textit{fst}} &= \lambda p.p\ \textbf{\textit{true}} \\ \textbf{\textit{snd}} &= \lambda p.p\ \textbf{\textit{false}} \end{aligned}$$

**Example 14** (Predecessor function)**.** *Pairs make possible a description of the numeric* predecessor function $pred : \mathbb{N} \to \mathbb{N}$*:*

$$pred(n) = \begin{cases} 0 & \textit{if} \quad n \leq 1 \\ n-1 & \textit{otherwise} \end{cases}$$

*For such, we require an auxiliary term* ***shiftInc*** *that operates on pairs of numbers.*

$$shiftInc(a, b) = (b, b+1)$$

*The term* ***shiftInc*** *has the following behavior: it copies the value on the second position of the pair to the first position, and then increments the number on the second position. If we apply the term* ***shiftInc*** $n$ *times over* (***pair 0 0***)*, and extract the first position of the pair, we obtain $n-1$. The encoding of these operations as lambda terms is presented below.*

$$\begin{aligned} \textbf{\textit{shiftInc}} &= \lambda p.\ (\textbf{\textit{pair}}\ (\textbf{\textit{snd}}\ p)\ (\textbf{\textit{succ}}\ (\textbf{\textit{snd}}\ p))) \\ \textbf{\textit{pred}} &= \lambda n.\ \textbf{\textit{fst}}\ (n\ \textbf{\textit{shiftInc}}\ (\textbf{\textit{pair 0 0}})) \end{aligned}$$

Notice that is possible to combine natural numbers and pairs to represent other numerical domains. For example, integers can be encoded by pairs $(s, n)$ where $s$ codifies the signal (positive or negative) and $n$ codifies the magnitude of the number.

Lists are fundamental in functional programming languages. They consist of ordered sequences of values of arbitrary (but finite) size. The inductive definition of lists consists of two basic operators: *empty*, which does not have parameters and represent the empty list, and *cons*, which receives a value $v$ and a list $l$, and returns $l$ prefixed with $v$. For example, the list of numbers $1, 2, 3$ is represented by means of these constructors as

$$(\texttt{cons 1}\ (\texttt{cons 2}\ (\texttt{cons 3 empty})))$$

Here we have the encoding of the constructors as terms.

**Definition 25** (List constructors)**.**

$$\begin{aligned} \textbf{\textit{empty}} &= \lambda x.\textbf{\textit{true}} \\ \textbf{\textit{cons}} &= \textbf{\textit{pair}} \end{aligned}$$

There are two basic operations for lists: `first`, which returns the starting element of the list, and `rest`, which removes the first element and returns the remaining list. There is one basic test, `isEmpty`, which receives a list $l$ and tests if if does not contain elements. It is considered an error to apply `first` or `rest` to the empty list. Here we have some examples of the result of list operations.

$$\begin{aligned} \texttt{first}\ (\texttt{cons 1}\ (\texttt{cons 2 empty})) &= \texttt{1} \\ \texttt{rest}\ (\texttt{cons 1}\ (\texttt{cons 2 empty})) &= (\texttt{cons 2 empty}) \\ \texttt{isEmpty}\ (\texttt{cons 1}\ (\texttt{cons 2 empty})) &= \texttt{false} \end{aligned}$$

The following terms encode the previous operations as lambda terms.

**Definition 26** (List operations)**.**

$$\begin{aligned} \textbf{\textit{isEmpty}} &= \lambda p.p\ (\lambda x\ y.\ \textbf{\textit{false}}) \\ \textbf{\textit{first}} &= \textbf{\textit{fst}} \\ \textbf{\textit{rest}} &= \textbf{\textit{snd}} \end{aligned}$$

*D. Recursive definitions*

Let us consider the problem of defining a term based on a recursive definitions, such as the factorial function:

$$fact(n) = \begin{cases} 1 & \text{if} \quad n \leq 1 \\ n * fact(n-1) & \text{otherwise} \end{cases}$$

We could attempt to translate this definition to a lambda term, arriving at the following equation:

$$\begin{aligned} \textbf{fact} = \lambda n.\textbf{if}\ &(\textbf{isZero}\ (\textbf{pred}\ n)) \\ &\textbf{1} \\ &(\textbf{mult}\ n\ (\textbf{fact}\ (\textbf{pred}\ n))) \end{aligned}$$

In a conventional programming language this definition would pose no problem, since the name **fact** would be registered in a global environment, and at each occurrence of the name of the function, such as in (**fact** (**pred** $n$))), a new call would be invoked with the new parameters. However, in lambda calculus there is no environment, only a term that reduces until it arrives at a normal form (or continues to reduce indefinitely). This definition, therefore, is circular and cannot be directly used to encode the factorial function as a lambda term.

A solution to this dilemma comes if we visualize the definition above in another way. Imagine the right-hand side of the equation as a transformation over the left-hand side:

$$\begin{aligned}\textbf{fact} = \lambda n.&\textbf{if } (\textbf{isZero } (\textbf{pred } n))\\ &\textbf{1}\\ &(\textbf{mult } n\ (\textbf{fact } (\textbf{pred } n)))\end{aligned}$$

We can give this transformation a name,

$$\begin{aligned}\mathbf{S} = \lambda M.\lambda n.&\textbf{if } (\textbf{isZero } (\textbf{pred } n))\\ &\textbf{1}\\ &(\textbf{mult } n\ (M\ (\textbf{pred } n)))\end{aligned}$$

which allows us to define the original equation as

$$\textbf{fact} = \mathbf{S}\ \textbf{fact}$$

Notice that the function we want to define, **fact**, is actually what we call a *fixed point* of the combinator **S**, i.e., a term which does not change when applying **S** over it. This is a consequence of the fact that a recursive function may be seen as an infinite sequence of chained `if-then-else`s, and thus should not be altered by adding an additional `if-then-else` step.

Now the question becomes if it is possible to obtain a term which is a fixed point of the combinator $\mathbf{S} : \Lambda \to \Lambda$. The good news is that there are terms that *construct* fixed points based on an arbitrary combinator.

**Definition 27** (Y combinator)**.** *The term below is called the **Y** combinator.*

$$\boldsymbol{Y} = \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$$

**Theorem 3.** *The **Y** combinator produces a fixed point for its argument.*

*Proof:*

$$\begin{aligned}\mathbf{Y}F &\to_\beta\\ (\lambda x.F(x\ x))\ (\lambda x.F(x\ x)) &\to_\beta\\ F\ (\lambda x.F(x\ x))\ (\lambda x.F(x\ x)) &=_\beta\\ F(\mathbf{Y}F)&\end{aligned}$$

Since $\mathbf{Y}S =_\beta S(\mathbf{Y}S)$, we have that $\mathbf{Y}S$ is a fixed-point of the term $S$ (under $\beta$-equivalence). ■

Using **Y** we can construct terms based on *recursive definitions*. We first construct a combinator based on the recursive definition, and then we pass it to the **Y** combinator as a parameter to obtain the required function.

**Example 15** (Definition of factorial)**.** *Considering **S** to be the combinator of factorial previously defined, we have*

$$\boldsymbol{fact} = \boldsymbol{Y}\ \boldsymbol{S}$$

One important remark is that *lazy evaluation* is required for the evaluation of terms based on the **Y** combinator, since the evaluation of **Y** itself diverges under *strict evaluation*.

## VIII. Conclusion

This paper presented a conceptual overview of the untyped lambda calculus and its use as an idealized programming language. The main elements of the theory, such as $\alpha$-equivalence and $\beta$-reduction were explained with the help of examples, and the consequences of some results (such as confluence of $\beta$-reduction) were discussed. We have also reviewed the traditional encodings for natural arithmetic, boolean operations, conditional expressions, pairs, lists and recursive definitions, showing the connection the calculus has with functional programming.

Due to the introductory nature of this text, several ideas were not mentioned. For instance, extensionality ($\eta$-reduction), impure systems ($\delta$-reduction), typed lambda calculi, the relationship between (typed) lambda calculus and mathematical logic, among others. Those topics are available in the references for the interested reader.

## Acknowledgment

I would like to acknowledge the organizers of the second WEIT (Workshop-Escola de Informática Teórica) for the invitation to present the course on lambda calculus. In particular, I would like to thank Graçaliz Dimuro for her helpfulness and assistance during the preparation of this paper.

## References

[1] H. P. Barendregt, *The Lambda Calculus: its Syntax and Semantics*, North Holland, 1984.

[2] G. Michaelson, *An Introduction to Functional Programming Through Lambda Calculus*, Addison-Wesley, 1989.

[3] M. H. Sørensen, P. Urzyczyn, *Lectures on the Curry-Howard Isomorphism*, Elsevier, 2006.

[4] H. P. Barendregt, *Lambda Calculi with Types, in Handbook of Logic in Computer Science*, Oxford University Press, 1992.

[5] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, *How to Design Programs: An Introduction to Computing and Programming*, MIT Press, 2003. Available online at http://www.htdp.org