# A Simple Introduction to Lambda Calculus

Luca Abeni

November 28, 2022

The $\lambda$ calculus is a formalism (or, if we prefer to see it from a CS point of view, a programming language) which allows us to define the fundamental concepts of functional programming: functions, definition of functions and application of functions.

If we see $\lambda$ calculus as a programming language, we can notice how it introduces the basic mechanisms needed to write functional programs without introducing the abstractions that characterize higher-level functional programming languages. Thus, $\lambda$ calculus can be seen as the FP version of the Assembly language. It is interesting, however, to note that even lower-level functional languages exist, because they introduce even fewer abstractions (for example, we will see that the function definition construct can be omitted).

From a different point of view, the $\lambda$ calculus can be seen as the theoretical foundation for functional programming, as it can be proved to be turing-complete. This result has a remarkable importance, because it means that the functional programming paradigm allows to implement any computable algorithm (ie: it has the same expressive power as the imperative programming paradigm).

Summing up, the basic elements of $\lambda$ calculus are just names, the concept of abstraction (definition of functions), and function application. Therefore, higher-level concepts such as data types, global environment, and the like do not exist. Since there are no different types of data, the basic elements of the $\lambda$ calculation are generic "functions", which receive another function as an argument and generate a function as a result. The domain and codomain of these functions are generic expressions (better, $\lambda$-expressions) and are not explicitly specified.

We will see that there is a typed version of the $\lambda$ calculus, in which the type of a function is specified by the function's domain and codomain (as traditionally done in the various analysis or algebra courses). Paradoxically, however, this formalism loses the expressive power of the original $\lambda$ calculus and is no longer Turing-complete.

The basic idea of $\lambda$ calculus is to express algorithms (or to code programs) as expressions, called $\lambda$-expressions in the following. As we will see, the execution of a program then consists in evaluation of a $\lambda$-expression (using a simplification mechanism called "reduction"). So let's see how $\lambda$-expressions are composed The incredibly simple syntax reflects the fact that $\lambda$-expressions are built starting from the three simple concepts already mentioned:

1. Variables (which actually represent functions). They constitute the terminal elements of the language and are indicated by names (identifiers), which will be represented by single letters written in *italic* (for example, "$x$", "$y$" or "$f$" ) in the following

2. Abstractions, which allow you to specify that a variable "$x$" is an argument in the following expression. Technically, an abstraction is said to "bind" a variable into an expression (the reason for this terminology will become clear later). In practice, an abstraction "creates" (informally speaking) a function starting from a $\lambda$ expression, specifying the argument of the function (the bound variable)

3. Functions applications. They represent the inverse operation of abstraction and allow to transform an abstraction and a $\lambda$ expression into a single $\lambda$ expression by removing the bound variable (actually, the whole abstraction is removed)

Using BNF notation, the syntax of a $\lambda$-expression is:

```
<expression> ::= <name>                          ; lowercase letter in italic
               | ( λ<name>.<expression>)        ; abstraction
               | (<expression> <expression>)    ; function application
```

This is equivalent to the following inductive definition:

- A name / variable / function (indicated with a single italic letter below) is a $\lambda$ expression

- If $e$ is a $\lambda$ expression and $x$ is a name, then $(\lambda x.e)$ is a $\lambda$ expression
- If $e_1$ and $e_2$ are $\lambda$ expressions, then $(e_1 e_2)$ is a $\lambda$ expression

Based on the definitions above, the following are examples of $\lambda$ expressions:

- $x$ (variable / function)
- $(\lambda x.(xy))$ (abstraction: bind variable "$x$" in the expression "$xy$", transforming that expression into a function of variable "$x$")
- $((xy)z)$ (application: apply function "$x$" to "$y$", then apply the result to "$z$")
- $(\lambda x.(xy))z$ (more complex expression)

According to what explained so far, any abstraction or application of a function should be enclosed in parentheses (to make the syntax less ambiguous); actually, the following conventions are assumed to reduce the number of parentheses used:

- Function application is left-associative: $((xy)z)$ is therefore equivalent to $xyz$
- The "$\lambda$" operator is right-associative and has lower precedence than function application: $(\lambda x.(xy))$ is therefore equivalent to $\lambda x.xy$

Another convention often used in the literature is that the variables linked by several immediately successive abstractions are grouped together; for example, $\lambda x.\lambda y.f$ can be written as $\lambda xy.f$. However, this convention will not be used in the following and we will keep only one variable for $\lambda$.

It is interesting to notice how the syntax of the $\lambda$-calculus allows to distinguish the definition of a function from its application: in the commonly used mathematical notation, the term "$f(x)$" is used both to indicate that the function "$f()$" is applied to the value "$x$" and to define this function (as in "$f(x) = x^2$"). In the $\lambda$-calculus, however, "$fx$" represents the application of "$f$" to "$x$", while "$\lambda x.f$" represents the definition of a function with argument "$x$".

Another interesting thing to note is that given the absence of a global environment it is not possible to *dynamically create* associations between $\lambda$ expressions and non-local names. In other words, not only the $\lambda$-calculus has no concept of "assignment" of values to a mutable variable, but it also lacks the equivalent of variable declaration (not even immutable variables). For convenience it is possible to use symbolic names for complex $\lambda$ expressions (see applied $\lambda$ calculus, below), but these are macro-like, static definitions (not bindings in a global environment that may vary dynamically at runtime).

As a result, only anonymous expressions and *anonymous functions* can be created in the $\lambda$-calculus (similar to what was done in standard ML with the `fn` construct, in Haskell with "\" or in C++ with lambda expressions "`[](...){...}`"). This could imply that $\lambda$-calculus does not allow defining recursive functions (and consequently is not Turing complete); we will see later how it is instead possible to define functions that require recursion by using the concepts of *fixed point* and *fixed point combinator*.

The only (name,value) bindings that can be created dynamically are the *local* bindings between formal parameters and actual parameters that are created during function application. This means that at least the concept of a local environment exists in the $\lambda$-calculus.

## 1 Semantic of the Lambda Calculus

As previously mentioned, $\lambda$-calculus allows programs to be encoded as expressions, which are "executed" by evaluating them via a process called reduction. Informally speaking, we can say that this process is based on the meanings that have been associated with the various basic elements of a $\lambda$ expression. To define the semantics of the $\lambda$-calculus in a more formal way, it is first necessary to introduce some basic concepts, such as "free variables" and "bound variables".

Intuitively, a variable "$x$" is bound by a construct "$\lambda x.E$" (where $E$ is a generic $\lambda$ expression), while it is free in an expression "$E$" if in "$E$" there is no $\lambda x$ abstraction that binds "$x$". To give a more formal definition, we must refer to the recursive definition of $\lambda$ expressions: in particular, if $\mathcal{B}(E)$ represents the set of bound variables in "$E$" and $\mathcal{F}(E)$ represents the set of free variables in " $E$", we can say that:

- For each variable "$x$", $\mathcal{F}(x) = \{x\}$ and $\mathcal{B}(x) = \emptyset$
- $\mathcal{F}(E_1 E_2) = \mathcal{F}(E_1) \cup \mathcal{F}(E_2)$; $\mathcal{B}(E_1 E_2) = \mathcal{B}(E_1) \cup \mathcal{B}(E_2)$

- $\mathcal{F}(\lambda x.E) = \mathcal{F}(E) - \{x\}$; $\mathcal{B}(\lambda x.E) = \mathcal{B}(E) \cup \{x\}$

Basically, this definition says that if an expression is composed of only one variable, that variable is free; composing two expressions (applying one expression to another) does not change the state of the variables (free variables remain free and bound variables remain bound) and the operator "$\lambda x.E$" binds the variable " $x$" in the expression "$E$" (it removes "$x$" from the set of free variables of "$E$" and adds it to the set of bound variables). The $\lambda$ operator is said to bind variable "$x$" in "$\lambda x.E$" because when the expression "$\lambda x.E$" is applied to an expression $E1$ a binding between $x$ and $E1$ is created in the local environment of $E$.

Based on this simple recursive definition it is possible to compute the set of free variables and bound variables for each $\lambda$ expression. A $\lambda$ expression which contains no free variables but is composed only of bound variables) is called a "combinator" and has the important property that the result of its evaluation only depends on the arguments (current parameters) used to evaluate it. More formally, a $\lambda$ expression $E$ is a combinator if $\mathcal{F}(E) = \emptyset$.

We can now define the concept of $\alpha$ equivalence between two $\lambda$ expressions. Informally, two $\lambda$ expressions $E_1$ and $E_2$ are $\alpha$ equivalent ($E_1 \equiv_\alpha E_2$) if they differ only in the parameters' names. This means that when defining a function the name of the function argument is not important (using a more familiar mathematical notation, $f_1(x) = x^2$ and $f_2(y) = y^2$ represent the same function); so, for example, $\lambda x.xy \equiv_\alpha \lambda z.zy$. The correct definition of $\alpha$ equivalence is obviously more complex, because, for example, $\lambda x.x\lambda x.xy$ is not $\alpha$ equivalent to $\lambda z.z.\lambda x.zy$ but is $\alpha$ equivalent to $\lambda z.z.\lambda x.xy$. Basically, $\lambda x.E$ is $\alpha$ equivalent to $\lambda z.E[x \to z]$, where $E[x \to z]$ represents the expression "$E$" with variable "$x$" is replaced by expression "$z$" only if it is free:

- If "$x$" and "$y$" are variables and $E$ is a $\lambda$ expression, $x[x \to E] = E$ and $y \neq x \Rightarrow y[x \to E] = y$
- Given two $\lambda$ expressions $E_1$ and $E_2$, $(E_1E_2)[x \to E] = (E_1[x \to E]E_2[x \to E])$
- If "$x$" and "$y$" are variables and $E$ is a $\lambda$ expression,
  - $y \neq x \wedge y \notin \mathcal{F}(E') \Rightarrow (\lambda y.E)[x \to E'] = \lambda y.(E[x \to E'])$
  - $y = x \Rightarrow (\lambda y.E)[x \to z] = \lambda y.E$

Looking back at the previous example, we can see how the rule "$y = x \Rightarrow (\lambda y.E)[x \to z] = \lambda y.E$" allows us to obtain the correct result: $\lambda x.x\lambda x.xy \equiv_\alpha \lambda z.(x\lambda x.xy)[x \to z] = \lambda z.x[x \to z](\lambda x.xy)[x \to z] = \lambda x.z\lambda x.xy$ as expected. It is also interesting to note that the rule "$y \neq x \wedge y \notin \mathcal{F}(E') \Rightarrow (\lambda y.E)[x \to E'] = \lambda y.(E[x \to E'])$" contains the condition "$y \notin \mathcal{F}(E')$": this condition is needed to avoid wrong substitutions like $(\lambda x.xy)[y \to x] = \lambda x.xx$ which would lead to $\alpha$ equivalences like $\lambda y.\lambda x.xy \equiv_\alpha \lambda x.\lambda x.xx$, clearly incorrect. This phenomenon, in which a free variable "$y$" in "$\lambda x.xy$" becomes bound after a substitution is called *variable capture* (because a simple substitution transforms a variable free in a bound variable) and should be avoided during substitutions. The substitution mechanism $E[x \to y]$ defined above is then called *capture-avoiding substitution* and can be used to formally define the $\alpha$ equivalence relation:

$$\lambda x.E \equiv_\alpha \lambda y.E[x \to y]$$

As suggested by the name, $\alpha$ equivalence is an equivalence relation: $E_1 \equiv_\alpha E_2$ is therefore a *symmetric*, *reflexive* and *transitive* relation between $\lambda$ expressions:

- $E \equiv_\alpha E$
- $E_1 \equiv_\alpha E_2 \Rightarrow E_2 \equiv_\alpha E_1$
- $E_1 \equiv_\alpha E_2 \wedge E_2 \equiv_\alpha E_3 \Rightarrow E_1 \equiv_\alpha E_3$

Capute-avoiding substitutions play a fundamental role in the $\lambda$-calculus , as they are used in the reduction mechanism to simplify $\lambda$ expressions as well as for $\alpha$ equivalences. Informally speaking, reducing a $\lambda$ expression consists in applying functions (removing abstractions) as in $(\lambda x.xy)z \to zy$. This procedure may appear simple, but it hides a series of complications; for example, the reduction $(\lambda x.(x\lambda y.xy))y \to y\lambda y.yy$ is clearly wrong, because the "$y$" variable is bound in the process (this is one of the reasons why the capture-avoiding substitution mechanism was defined earlier!). Thus, whenever you have an abstraction ("$\lambda x.E$") applied to an expression $E_1$, you can use a **capture-avoiding** substitution of $E_1$ in $E$ (replacing $x$) to reduce the $\lambda$ expression eliminating the abstraction.

More formally, a *redex* (*red*ucible *ex*pression) is defined as a $\lambda$ expression of the type $(\lambda x.E)E_1$ and $E[x \rightarrow E_1]$ is defined as is its reduced. Based on this, the $\beta$ reduction "$\rightarrow_\beta$" can be defined as the replacement of a redex by its reduction:

$$(\lambda x.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

It might appear that some redexes cannot be reduced because capture-avoiding reduction cannot be used (for example, a variable "$y$" is bound in "$E$" and "$y$" appears among the free variables of $E_1$ - in this case, a simple replacement would capture the free "$y$"). Consider the $\lambda$ expression $(\lambda y.\ lambdax.xy)(xz)$: this expression clearly represents a redex, so one could try to reduce it using the $\beta$ reduction mechanism, which would lead to $(\lambda x.xy)[y \rightarrow (xz)]$. Note however that $x \in \mathcal{F}(xz)$, so none of the rules presented in the definition of the capture-free substitution mechanism can be used (again: $(\lambda x.xy)[y \rightarrow (xz)] = \lambda x.x(xy)$ is not a capture-avoiding substitution, because it would capture the red "$x$"). How can this kind of redex be reduced, then? The concept of $\alpha$ equivalence comes to our aid, allowing us to rename the variables which are bound in $E$ so that they do not appear among the free variables of $E_1$. In the previous example:

$$(\lambda y.\lambda x.xy)(xz) \equiv_\alpha (\lambda y.\lambda k.ky)(xz) \rightarrow_\beta (\lambda k.ky)[y \rightarrow (xz)] = \lambda k.k(xz)$$

this time the reduction does not capture any free variable, and is correct (the capture-avoiding substitution $(\lambda k.ky)[y \rightarrow (xz)]$ is now possible because $k \notin \mathcal{F}(xz)$ ).

The $\beta$ reduction is not an equivalence relation, as it does not have the reflexive property: $E_1 \rightarrow_\beta E_2$ does not imply $E_2 \rightarrow_\beta E_1$. However, an equivalence relation (called $\beta$ equivalence "$\equiv_\beta$") can be created by computing the $\beta$ reduction reflexive and transitive closure. In practice, $E_1 \equiv_\beta E_2$ means that there is some chain of $\beta$ reductions which "connect" $E_1$ and $E_2$ ($\beta$ reducing $E_1$ and $E_2$ multiple times it is possible to arrive at the same expression $E$).

More formally, the $\beta$ equivalence $\equiv_\beta$ is defined as:

- $E_1 \rightarrow_\beta E_2 \Rightarrow E_1 \equiv_\beta E_2$
- $\forall E, E \equiv_\beta E$
- $\forall E_1, E_2 : E_1 \equiv_\beta E_2, E_2 \equiv_\beta E_1$
- $E_1 \equiv_\beta E_2 \wedge E_2 \equiv_\beta E_3 \Rightarrow E_1 \equiv_\beta E_3$

Finally, it is interesting to note that a generic $\lambda$ expression usually contains multiple redexes and the rules of the $\lambda$ computation do not define an order for applying the possible $\beta$ reductions. In this case it is possible to reduce the expression following any order for the $\beta$ reductions (provided that the parentheses and the rules of associativity and precedence between operators are respected).

The order in which to evaluate the various redexes is decided by defining an evaluation strategy in addition to the reduction rules of the $\lambda$-calculus (for example, proceed to the right starting from the leftmost redex, or start from the "innermost" redex, etc...). The various evaluation strategies (lazy vs eager, by name vs by value, etc...) used by higher level programming languages derive from this evaluation strategy.

An important theorem (the Church-Rosser Theorem) proves that if a $\lambda$ expression $E$ can be reduced to $E_1$ by 0 or more $\beta$ reductions and $E$ can be reduced to $E_2 \neq E_1$ by 0 or more $\beta$ reductions, then there exists $E_3$ such that both $E_1$ and $E_2$ can be reduced to $E_3$ by 0 or more $\beta$ reductions ($E \rightarrow_\beta ... \rightarrow_\beta E_1 \wedge E \rightarrow_\beta ... \rightarrow_\beta E_2 \Rightarrow \exists E_3 : E_1 \rightarrow_\beta ... \rightarrow_\beta E_3 \wedge E_2 \rightarrow_\beta ... \rightarrow_\beta E_3$).

An important corollary of this theorem is that if $E$ is reducible to a normal form ($\lambda$ expression which no longer contains any redex), then this normal form does not depend on the order of $\beta$ reductions. In other words, every $\lambda$ expression $E$ has at most 1 normal form. Note the use of the term "*at most*", as there are $\lambda$ expressions which cannot be reduced to a normal form (the reduction process never ends). A typical example is the combinator $\Omega = \omega\omega$, where $\omega = \lambda x.xx$:

$$\Omega = \omega\omega = (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (xx)[x \rightarrow (\lambda x.xx)] = (\lambda x.xx)(\lambda x.xx) = \omega\omega = \Omega$$

therefore, $\Omega \rightarrow_\beta \Omega$!!! This is the equivalent of an infinite loop in an imperative language, or an infinite recursion in a functional language. The existence of this type of expression (which generates a non-terminating sequence of reductions) is necessary for the Turing-completeness of the $\lambda$-calculus (the Turing machine allows to encode infinite computations; if the $\lambda$ calculus did not allow to encode infinite reductions, then it could not implement such Turing machine programs).

## 2 Encoding High-Level Languages

After seeing the most important definitions of the $\lambda$-calculus and the details of the reduction mechanism, it is quite difficult to understand how such a simple and seemingly inexpressive formalism can be Turing complete. In fact, it might seem that the $\lambda$ computation could only be useful for manipulating functions or the like.

Making a parallelism with imperative programming, we can remember that programs written in a high-level languages are transformed into Assembly (by a compiler or an interpreter) to be executed by a physical CPU. Just as the $\lambda$-calculus allows you to work only with functions (and to perform relatively simple reduction operations on expressions composed only of functions, abstractions and applications), the Assembly language also allows you to operate only on binary numbers (stored in CPU registers or RAM) and has no concept of datatypes or global environment. Yet we have no problem in thinking that a program written in a high-level language with a global environment and strictly typed variables is converted to Assembly: we “just” need to implement all the high-level concepts based on Assembly instructions that operate on registers or memory. Similarly, the same high-level concepts can be implemented by using just functions, abstractions, and function applications.

In general, the various high-level abstractions we will be encoded by using combinators (which, as already said, are $\lambda$ expressions in which no free variables appear). This is because the encoding must not depend on the context (hence, it must not refer to any symbol that is not a formal parameter / argument of the expression). As an example, some notable combinators known in the literature are:

- The combinator representing the identity function: $I = \lambda x.x$
- The combinator representing function composition: $B = \lambda f.\lambda g.\lambda x.f(gx)$
- $K = \lambda x.\lambda y.x$
- $S = \lambda f.\lambda g.\lambda x.fx(gx)$
- $\omega = \lambda x.xx$
- $\Omega = \omega\omega$
- $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

The importance of the $Y$ combinator is fundamental for the $\lambda$-calculus, as it is a so-called *fixed point combinator* (this will be discussed later). The $K$ and $S$ combinators are interesting as they allow to define a subset of the $\lambda$ calculus (called “SK calculus” or “SKI calculus”) in which the abstraction operator $\lambda$ is not explicitly used (!!!) (this will also be explained later) Before going on, note that the “=” symbol that has been informally used above to define the various combinators represents some kind of equality / equivalence (indicating, for example, that writing “$I$ ” is equivalent to writing “$\lambda x.x$”) and is not a formally defined construct in the $\lambda$-calculus. Remember that the $\lambda$-calculus does not allow to modify some kind of non-local environment or to create bindings between names and symbols (again: in the $\lambda$ calculation, there is no construct that allows you to create links between names and symbols at a global level!).

After these necessary premises, we can start to see how to implement the high-level constructs we are interested in using the $\lambda$ calculus. The first thing to do is find a “$\lambda$ encoding” for the natural numbers (and for the operations that can be performed on them). These encodings can then be used to encode integers (natural numbers with sign), rational numbers, real numbers, and so on.

Then, $\lambda$ expressions will be used to also represent boolean values, basic logical operations and the so-called “arithmetic if” (which allows to evaluate an expression $E_1$ or an expression $E_2$ depending on the truth value of a boolean expression). Finally, more complex data structures can be implemented to show how higher-level data types can be represented by $\lambda$ expressions.

Recalling that the $\lambda$-calculus is a functional formalism, it is quite clear that it will be necessary to use an inductive definition of the natural numbers, similar to Peano’s:

- 0 is a natural number
- Given a natural number $n$, the successor of $n$ (computable as $succ(n)$) is a natural number

The idea is therefore to use a $\lambda$ expression to represent the natural number 0 and define a combinator which, applied to the representation of a natural number $n$, computes the representation of $n + 1$. The *Church numerals* implement this idea:

- 0 is represented by the $\lambda$ expression $\lambda f.\lambda x.x$
- The *succ*() function (which calculates the successor of a natural number $n$) is represented by the $\lambda$ expression $\lambda n.\lambda f.\lambda x.f(nfx)$

This encoding has the interesting property that the natural number $n \in N$ is represented by the function $f$ applied $n$ times to x: $n \equiv \lambda f.\lambda x.\overbrace{f(\ldots f}^{n}(x)\ldots)$ (to simplify the notation, the expression "$\overbrace{f(\ldots f}^{n}(x)\ldots)$" is sometimes written as "$f^n(x)$").

As an exercise, we can try to compute the representation of the natural number 1 as $1 = succ(0)$:

$$1 = succ(0) = (\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x)$$

$$(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x) \rightarrow_\beta (\lambda f.\lambda x.f((nf)x))[n \rightarrow (\lambda f.\lambda x.x)] = \lambda f.\lambda x.f(((\lambda f.\lambda x.x)f)x) \rightarrow_\beta$$
$$\rightarrow_\beta \lambda f.\lambda x.f(((\lambda x.x)[f \rightarrow f])x) = \lambda f.\lambda x.f((\lambda x.x)x) \rightarrow_\beta \lambda f.\lambda x.f((x)[x \rightarrow x]) = \lambda f.\lambda x.f(x)$$

Similarly, it is possible to compute the encoding of 2:

$$2 = succ(1) = (\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.f(x))$$

$$(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.f(x)) \rightarrow_\beta (\lambda f.\lambda x.f((nf)x))[n \rightarrow (\lambda f.\lambda x.f(x))] = \lambda f.\lambda x.f(((\lambda f.\lambda x.f(x))f)x) \rightarrow_\beta$$
$$\rightarrow_\beta \lambda f.\lambda x.f(((\lambda x.f(x))[f \rightarrow f])x) = \lambda f.\lambda x.f((\lambda x.f(x))x) \rightarrow_\beta \lambda f.\lambda x.f(f(x))$$

Without pretending to give rigorous proofs, let's try to better understand the encoding of *succ*(): informally speaking, *succ* must transform $\lambda f.\lambda x.f^n(x)$ in $\lambda f.\lambda x.f(f^n(x))$. This can be done:

1. By somehow "removing" the two abstractions $\lambda f.\lambda.x.$ from the encoding of $n$
2. Then applying $f$ to the expression obtained above
3. Then adding back the two abstractions $\lambda f.\lambda.x.$ removed in step 1
4. And finally abstracting everything from the number $n$

The first step (removal of the abstractions $\lambda f.\lambda.x.$) can easily be accomplished by applying the natural number encoding to $f$ and $x$: in fact, $(\lambda f.\lambda x.f^n(x))f \rightarrow_\beta \lambda x.f^n(x)$ and $((\lambda f.\lambda x.f^n(x))f)x \rightarrow_\beta (\lambda x.f^n(x))x \rightarrow_\beta f^n(x)$. Thus, if the function $n$ represents the encoding of a natural, then $(nf)x$ is an expression containing f applied $n$ times to $x$. As mentioned (step 2), $f$ must be applied to this expression once again, obtaining $f((nf)x)$; after step 3 we obtain $\lambda f.\lambda x.f((nf)x)$ and abstracting everything with respect to $n$ (so that $n$ is an argument of the combinator *succ* and not a free variable) we get $\lambda n.\lambda f.\lambda x.f((nf)x)$ which is just the encoding of *succ* presented above.

Based on the definitions of Church numerals, it is possible to define the encoding of the various operations on the natural numbers. For example, the sum can be encoded using a combinator which, applied to the encodings of two natural numbers $n$ and $m$, generates the encoding of their sum. The expression of this combinator is $\lambda m.\lambda n.\lambda f.\lambda x.(mf)((nf)x)$ and can be obtained in this way:

1. First, we apply $n$ to $f$ and $x$ to remove the abstractions $\lambda f.\lambda x.$, similar to what we did for encoding *succ*
2. Next, apply $m$ to $f$ to remove the abstraction $\lambda f.$
3. Next, applying the result of $mf$ (that is, "$m$" without "$\lambda f$") to the result of $((nf)x)$ (which is "$f^n x$"), we add $m$ "$f($" to the left of "$f^n x$". The result is "$f^{n+m}x$"
4. As done for *succ*, we abstract again with respect to $f$ and $x$ to add the $\lambda f.\lambda x.$ removed in step 1
5. Finally, we abstract with respect to $n$ and $m$, in order to obtain a combinator

It is then possible to define the encoding of the other operations on natural numbers, but the the details are omitted here for the sake of brevity. Coding a "*pred*" operator (which calculates the predecessor of a natural number) is possible, but not easy (and there are strange anecdotes involving Alonso Church - inventor of the $\lambda$ calculus - one of his PhD students - who solved the *pred* encoding problem - and a barber). Without going into details, the encoding of this operator involves transforming the encoding of $n$ into a pair containing the encoding of $n$ and the encoding of $n-1$, and then taking the second element of the pair. The encoding of $(n, n-1)$ is generated from the encoding of $n$ by starting from the encoding of $(0,0)$ and iterating $n$ times a function $\hat{f}$ which transforms the pair $(n, m)$ into $(n+1, n)$. Now, remembering that the encoding of $n$ is a combinator that applies $n$ times its first argument to its second argument, it becomes clear that $(n, n-1)$ can be obtained applying $n$ to $\hat{f}$ and then applying the resulting function to the encoding of $(0,0)$. At this point, as mentioned, the encoding of $n-1$ can be obtained by applying to the result a function that returns the second element of a pair. As usual, everything must be abstracted from $n$. In light of this, it is therefore important to understand how to encode pairs using the *lambda*-calculus.

The "$(a, b)$" pair can be encoded as $\lambda z.zab$ and in general the function that generates the encoding of the "$(a, b)$" pair from "$a$" and "$b$"is $\lambda x.\lambda y.\lambda z.zxy$. Given the encoding of a pair, it is possible to obtain the first element using the function "first" = "$\lambda z.z(\lambda x.\lambda y.x)$" and the second element using the function ' 'second" = "$\lambda z.z(\lambda x.\lambda y.y)$".

As already mentioned, in addition to natural numbers and arithmetic operations, the $\lambda$-calculus allows you to encode everything needed to implement any algorithm. It is hence important to encode the boolean values `true` and `false`, and the selection operation (arithmetic if). A simple encoding for `true` could be "$\lambda t.\lambda f.t$", while `false` could be encoded as "$\lambda t.\lambda f.f$": informally speaking, `true` and `false` are encoded as $\lambda$ expressions with two arguments, which return the first or second argument.

The selection function (arithmetic if), on the other hand, can be encoded as "$\lambda c.\lambda a.\lambda b.cab$": it is a $\lambda$-expression that receives 3 arguments "$c$", "$a$" and "$b$", where "$c$" is the encoding of a boolean value. If "$c$" is the encoding of `true`, then the expression evaluates to "$a$", otherwise it evaluates to "$b$":

$$(\lambda c.\lambda a.\lambda b.cab)(\lambda t.\lambda f.t) \to_\beta \lambda a.\lambda b.(\lambda t.\lambda f.t)ab \to_\beta \lambda a.\lambda b.(\lambda f.a)b \to_\beta \lambda a.\lambda b.a$$

And

$$(\lambda c.\lambda a.\lambda b.cab)(\lambda t.\lambda f.f) \to_\beta \lambda a.\lambda b.(\lambda t.\lambda f.f)ab \to_\beta \lambda a.\lambda b.(\lambda f.f)b \to_\beta \lambda a.\lambda b.b$$

Based on these encodings it is then possible to implement the boolean operators `and` ($\lambda p.\lambda q.pqp$), `or` ($\lambda p.\lambda q.ppq$), and so on[1].

It is then possible to encode boolean predicates such as "is zero" (which receives the encoding of a natural number as an argument and evaluates to `true` if the number is 0), "less than" , "equal" and similar.

Although the encodings of values and operations presented so far allow to implement generic functions (a mechanism to implement / encode recursion or iteration is still missing, but will be shown shortly), the resulting lambda expressions risk to be too complex. For example, the simple arithmetic expression "$2+3$" is encoded as "$2+3 \equiv (\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$"!!! And the encoding of the function "$f(a) = a+2$" is "$\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$" . To simplify the expressions, it is possible to use a notation sometimes known as "applied lambda calculus", in which the encodings of the various values and operations are replaced with more common mathematical symbols. Thus, "+" is a synonym for "$(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))$", "2" is a synonym for "$(\lambda f.\lambda x.f(fx))$" and so on. It then becomes possible to write "$\lambda a.a+2$" instead of the lambda expression mentioned above.

At this point, the most important thing that seems to be missing is a loop construct (or rather, recursion, since we are talking about functional programming!). As already mentioned, the lack of a global environment seems to make it impossible to implement recursion. But the latest surprise of $\lambda$-calculus, the concept of fixed point combinator, comes to our aid.

As known, if the "imperative" version of an algorithm contains a loop, its implementation according to the functional paradigm is based on recursion: in other words, the implementation of a function calls the function itself (the typical example is the factorial). But the $\lambda$-calculus allows you to define only anonymous functions ($\lambda$ abstractions) and without a global environment, a function cannot call itself, as it has no name. In other words, a recursive function contains at least one free variable (therefore it is not a combinator), which indicates the name of the function itself, to be called recursively. The first step to implement recursion in the $\lambda$-calculus is therefore to eliminate this free variable (thus transforming the recursive function into a combinator). This can be done by passing the name of the function as an argument. Therefore, if $f = E$ is an expression that recursively calls $f$ (itself), it is transformed into $f_c = \lambda f.E$, binding the variable $f$, which then becomes the first argument of $f_c$.

In other words, $f$ can be seen as the result obtained by passing $f$ as an argument to $f_c$: $f = f_c f$, where in this case "=" means " $\equiv_\beta$" ($\beta$ equivalent). This is not simply a syntactic trick, but allows us to reformulate our problem as an equation whose solution $f$ is the recursive function we are looking

[1]The reader can verify the correctness of the encodings of `and` and `or` as a simple exercize.

```
unsigned int factorial(unsigned int n)
{
  unsigned int i res = 1;

  for (i = 2; i <= n; i++) {
    res = res * i;
  }

  return res;
}
```

Figure 1: Iterative implementation of the `factorial()` function.

```
unsigned int factorial(unsigned int n)
{
  if (n == 0) return 1;
  return n * fattoriale(n - 1);
}
```

Figure 2: Recursive implementation of the `factorial()` function.

for. Such an equation $f \equiv_\beta f_c f$ is solved by finding the "fixed point" of $f_c$. The existence of *fixed point combinators* (combinators that given a function $f_c$ compute its fixed point $f = f_c f$) shows us that recursion is implementable in $\lambda$-calculus, even if only anonymous functions exist.

The most famous fixed point combinator is $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

As an example, let's see how to use the fixed point combinator Y to calculate the factorial function. A possible imperative implementation of the factorial is shown in Figure 1, while the traditional recursive implementation is shown in Figure 2. Figure 3 shows a "more functional" implementation of this function. A first attempt (not too successful, actually) of conversion to $\lambda$-calculus could be

$$\texttt{factorial} = \lambda n.\texttt{cond}\ (n = 0)1(n * (\texttt{factorial}\ \ (\text{pred}\ n))$$

Note that this function, which might look strange when talking about pure $\lambda$-calculus (as it contains functions like "`pred`",predicates like $n = 0$, and expressions like $n - 1$, which are not part of the pure $\lambda$-calculus), was written using the applied $\lambda$-calculus. Remember that the selection "`cond`" (arithmetic if) can be replaced with the $\lambda$-expression $\lambda c.\lambda a.\lambda b.cab$, the predicate "$n = 0$" can be replaced with the $\lambda$-expression encoding of "is zero" and `pred` can be replaced with its encoding mentioned earlier.

As already noted several times, this is a strange form of definition because it requires the presence of a binding for its own name in the global environment. This problem is solved by defining the function $f_c$ as

$$\lambda f.\lambda n.\texttt{cond}(n = 0)1(n * (f(\texttt{pred}\ \ n))$$

and finding the function `factorial` such that `factorial`$= f_c$`factorial` (fixed point of $f_c$). This function $f$ can be calculated using the fixed point combinator $Y$: `factorial`$= Y f_c$.

Finally, it should be noted that the encodings of data types[2] and high-level functions presented above are not unique. For example, using Church numerals it is possible to define operations such as addition or predecessor in different ways (of course all the definitions will be functionally equivalent).

Going further, it can be seen that Church encoding is only one of the possible way to encode high-level data structures and functions and other alternative encodings are possible. For example, the so-called *Scott's numerals* propose an alternative encoding of natural numbers (and operations on them) to that of Church:

- The natural number 0 is represented by the $\lambda$ expression $\lambda f.\lambda x.f$
- The function `succ` which calculates the next of a natural number $n$ is encoded by the $\lambda$ expression $\lambda n.\lambda f.\lambda x.xn$

[2]Actually, only the encoding of natural numbers has been presented... But it is possible to encode in $\lambda$-calculus any type of data.

```
unsigned int factorial(unsigned int n)
{
  return (n == 0) ? 1 : n * fattoriale(n - 1);
}
```

Figure 3: Functional implementation of the `factorial()` function.

Although Scott's coding is less known (and less used!) than Church's, in some cases it has advantages (for example, it allows simplifying the definition of the predecessor function `pred` which is codable as $\lambda n.n(0)(\lambda x.x)$).

## 3 Removing Abstractions

As seen, the main constructs of the $\lambda$-calculus are abstraction (definition of functions) and function application. Actually, it is possible to define a minimal functional programming language even without using the abstraction mechanism, provided that an adequate set of predefined functions is provided. What you get in this way is a "*combinatory calculus*", so called because of the predefined combinators (the set of predefined functions mentioned above) on which it is based.

The syntax of an expression of this type of calculus can be defined as:

```
<expression> ::= <name> ; lowercase letter
               | <Combinator> ; default function
               | (<expression> <expression>) ; application
```

where `<name>` is the name of a variable and `<Combinator>` is a built-in function (with a well-defined behavior). This is equivalent to the following inductive definition:

- A name / variable (denoted by a single lowercase letter) is an expression of combinatory calculus
- A combinator (denoted by a single uppercase letter) is an expression of combinatory calculus
- If $e_1$ and $e_2$ are expressions of the calculus, then $(e_1 e_2)$ is an expression too

Note how all the variables appearing in an expression are free variables (because there is no concept of abstraction that can bind them).

The details of a combinatory calculus clearly depend on the predefined combinators and it is clear that not all the combinations of combinators result in a turing complete calculus.

The most important of the combinatory calculus is probably the "SK calculus" (sometimes known as "SKI calculus"), where the default combinators are $S$ and $K$ (plus optionally the identity combinator $I$) defined by the following properties:

$$\begin{aligned} Kxy &= x \\ Sxyz &= xz(yz) \\ Ix &= x \end{aligned}$$

The combinator $I$ is often used to simplify calculus expressions, but it is not strictly necessary, as it can be obtained as $I = SKK$: $(SKK)x = SKKx = Kx(Kx) = x$.

The $S$ and $K$ combinators presented in Section 2 ($S = \lambda f.\lambda g.\lambda x.fx(gx)$ and $K = \lambda x.\lambda y.x$) enjoy the properties described above (and the proof of this is is left to the reader). It is therefore easy to convert an expression of the SK calculus (or SKI calculus) into a $\lambda$-expression. Since it is also possible to convert any $\lambda$-expression into an expression of the SK calculus, the SK calculus has the same expressive power as the $\lambda$-calculus and is therefore Turing-complete.

The conversion of a generic $\lambda$-expression $E$ into an expression of the SK calculus can be performed proceeding by cases. In particular, $E$ can be:

- An identifier $x$, which maps to the expression $x$ of the SK calculus
- An application $E_1 E_2$, which maps to the expression $E_1 E_2$ of the SK calculation
- An abstraction $\lambda x.E'$, which must be converted into an expression $E''$ of the SK calculus proceeding by case:

– If $E'$ is an identifier, it can be $x$, in which case $E = \lambda x.x$ maps to $E'' = I = SKK$, or a symbol $y \neq x$, in which case $E = \lambda x.y$ maps to $E'' = Ky$

– If $E'$ is an application $E'_1 E'_2$, $E = \lambda x.E'_1 E'_2$ must be converted into $E''$ such that $(\lambda x.E'1E'_2)v = E''v$. This implies that

$$(\lambda x.E'_1 E'_2)v = E''v \Rightarrow E'_1[x \to v]E'_2[x \to v] = E''v \Rightarrow$$
$$(\lambda x.E'_1 v)(\lambda x.E'_2 v) = E''v \Rightarrow S(\lambda x.E'_1)(\lambda x.E'_2)v = E''v \Rightarrow$$
$$E'' = S(\lambda x.E'_1)(\lambda x.E'_2)$$

– If $E'$ is an abstraction $\lambda y.E'_1$, $E = \lambda x.\lambda y.E'_1$ must be converted into $E''$ by recursively applying this procedure at $E'_1$.

Applying this reasoning, one can convert any $\lambda$-expression into an expression based only on free variables, the operator $S$ and the operator $K$.

Another important property of the SK calculus is that every $\lambda$-expression that contains no free variables (that is, a combinator) can be converted to an SK calculus expression that contains no variables. In other words, to model combinators it is possible to remove the first clause (a variable name is an expression of the SK calculus) from the definition of the SK calculus.

The process which converts an expression "$E$" into an expression $R_x(E)$ which does not contain the free variable "$x$" but behaves like $\lambda x.E$ (that is that is, $R_x(E)E_1 = E[x \to E_1]$) is known as *bracket abstraction* and can be used to convert any $\lambda$-expression to an expression of the SK calculation eliminating the abstractions $\lambda x.$ one by one. A very simple (although not efficient) bracket abstraction algorithm is based on the following transformations:

1. $R_x(x) = SKK$, where "$x$" is the variable to be eliminated
2. $R_x(y) = Ky$, where $y$ is a predefined combinator ($S$ or $K$) or a variable other than the variable $x$ to be eliminated
3. $R_x(E_1 E_2) = SR_x(E_1)R_x(E_2)$

To make the algorithm slightly more efficient, the second rule can be replaced by $R_x(E) = KE$, where "$E$" is an expression that does not contain as a variable free the variable "$x$" that has to be removed.

To convert a $\lambda$-expression into an SK calculus expression, one can proceed by removing the $\lambda$ abstractions one by one by applying the three rules above. Note the close relationship between this bracket abstraction algorithm and the conversion methodology shown above.

## 4 Typed Lambda Calculus

As noticed, in the "pure" $\lambda$-calculus there is no concept of data type. It has been shown how it is possible to use expressions of the untyped $\lambda$-calculus to encode the various data types (and the operations on them), but the variables of the $\lambda$-calculus represent generic functions (with unspecified domain and codomain).

Although the lack of data types does not impact the expressivity of the formalism (as mentioned, the $\lambda$-calculus is Turing complete), it can compromise the readability and simplicity of use, making it easier to introduce programming errors (for this reason the $\lambda$-calculus is considered a sort of "Assembly of functional languages"). For example, if you code the function $f(a) = a + 2$ using $\lambda$-calculus you get $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$ (not exactly intuitive expression...), which using the applied $\lambda$-calculation simplifies to $\lambda y.y{+}2$. However, this encoding has lost a fundamental characteristic of the initial function: the fact that the function operated on numbers! In fact, it is possible to apply $\lambda a.a{+}2$ (which, we recall, is equivalent to $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx)))$ to any $\lambda$-expression, even if it doesn't encode a number! If the function is applied to an expression $E$ which encodes a natural number, a $\lambda$-expression which encodes a natural number is generated as a result, otherwise a $\lambda$-expression $E'$ without any clear interpretation can be generated.

To solve this kind of problems, it is possible to associate a type with each $\lambda$-expression (or with each argument). For example, introducing the constraint that $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$ is a function $\mathcal{N} \to \mathcal{N}$ or, better, that in this expression "$a$" is the encoding of a natural number ($a : \mathcal{N}$).

In this section (which does not claim to be exhaustive, but only to introduce some concepts that can then be further explored by the readers) it will be shown how to extend the original formalism to specify domain and codomain for each function. Of course this can be done in various ways which result in different definitions of typed $\lambda$-calculus, the most famous of which are due again to Alonso Church and Haskell Curry. We will then talk about $\lambda$-calculus with types "a-la Church" or "a-la Curry".

Surprisingly enough, associating types to functions actually reduce the expressive power of the formalism, which is no longer Turing complete. This happens because it can be proved that the reduction of any "well-typed" expression (this concept will be introduced intuitively in the next few pages) by a typed $\lambda$-calculus always ends (it is therefore no longer possible to express infinite recursions). For a simple intuition of this fact try to compute the type of the Y operator, which is necessary to encode recursive functions.

First of all, to define a typed $\lambda$-calculus we need to introduce the concept of type; this can be done by introducing a set $\mathcal{P}$ of basic types, or *primitive types*, and a rule for defining new datatypes starting from existing ones (this is analogous to what was done to define the expressions of the $\lambda$-calculus, created starting from a set of base names and 2 rules which allow to create new expressions starting from valid expressions). Since we are defining types for a $\lambda$-calculus, a new type $\gamma$ could be defined starting from two types $\alpha$ and $\beta$: $\gamma = \alpha \to \beta$ ($\gamma$ is the type of functions having $\alpha$ as domain and $\beta$ as codomain). In other words, the set $\mathcal{T}$ of possible data types can be generated through the following inductive definition:

- A primitive type name refers to a type: $\alpha \in \mathcal{P} \Rightarrow \alpha \in \mathcal{T}$
- If $\alpha$ and $\beta$ are types, then $\alpha \to \beta$ is also a type: $\alpha, \beta \in \mathcal{T} \Rightarrow \alpha \to \beta \in \mathcal{T}$

As it is easy to understand from this definition, the number of possible types (the cardinality of the set $\mathcal{T}$ of types) is infinite.

Given a $\lambda$-expression $E$, its type is computable according to the following rules:

- The type of a free variable $x$ must be known a-priori
- If $E_1$ and $E_2$ are expressions with types $\alpha \to \beta$ and $\alpha$, then the type of $E_1E_2$ is $\beta$: $E_1 : \alpha \to \beta, E_2 : \alpha \Rightarrow E_1E_2 : \beta$
- If $E$ is an expression of type $\beta$, $\lambda x.E$ has type $\alpha \to \beta$: $E : \beta \Rightarrow \lambda x.E : \alpha \to \beta$

the third rule can be better clarified by modifying the syntax of the abstraction to specify the type of the argument. In these cases $\lambda x : \alpha.E$ (explicit typing) is used instead of $\lambda x.E$ (implicit typing). Making an analogy with higher level languages, we can consider the C language (in which a variable declaration requires to specify the variable's type), the C++ language (in which the compiler can be asked to infer the tyoe of a variable, by usign the ' '`auto`" keyword), and standard ML or Haskell (where variables can be declared without specifying their type, because the compiler is able to infer it himself).

Also note that to associate a type with an expression $E$ it is necessary to make assumptions about the types of the free variables contained in $E$ (see the first rule above). These assumptions (obviously only necessary for non-closed expressions) are contained in some sort of "type environment", or "type context". In summary, the type of a closed expression can be somehow "computed" (better: inferred) without needing additional information, while the type of an open expression depends on the environment (or context) of the types.

Informally speaking, an expression $E$ is correctly typed if it is possible to associate $E$ with a type $\alpha \in \mathcal{T}$ that is consistent with the rules presented above. For example, the expression $\lambda x : \texttt{int}.x$ is correctly typed (and has type $\texttt{int} \to \texttt{int}$). The expression $I = \lambda x.x$ (or $I = \lambda x : \alpha.x$) is also correctly typed and has type $\alpha \to \alpha$. The combinator $\omega = \lambda x.xx$ is instead not correctly typed: assuming that $x$ has type $\alpha$ ($x : \alpha$), we have that $\omega : \alpha \to \beta$, where $\beta$ is the type of the expression "$xx$". But for "$xx$" to be a valid expression, $x$ must be a function, with domain $\alpha$ (the type of the argument). Thus, $x : \alpha \to \beta$, but also $x : \alpha$, from which we derive $\alpha = \alpha \to \beta$, which is not a valid expression in the type system we have defined (that is, using the type generation rules given above it is not possible to "construct" a type $\alpha \in \mathcal{T}$ that has this property).

Once the concept of types and correctly typed expressions has been introduced, two different approaches can be followed:

- The first approach consists in *defining the semantics of expressions regardless of their type* (in practice, we define $\beta$ reduction rules that do not depend on the types of the expressions). The concept of type is then only used a posteriori to "reject" incorrectly typed expressions as invalid
- The second approach is to specify semantics only to correctly typed expressions. In other words, if an expression is not correctly typed (that is, its type cannot be generated with the rules presented above), it doesn't even make sense to try to reduce it.

According to the first approach, which results in the so-called “$\lambda$-calculus with types a-la Curry”, the introduction of types is used to “eliminate” from the calculus the expressions which ‘ ‘do not behave as desired” (for example, expressions whose reduction does not finish). But the reduction of such expressions is still defined. In essence, types simply add extra constraints on expressions, which characterize “valid expressions”.

In the second approach, which results in the so-called “$\lambda$-calculus with types a-la Church”, the type of an expression is considered fundamental for its semantics (it is not possible define the semantics of an incorrectly typed expression). Hence, the reduction rules for $\lambda$-expressions explicitly refers to the expressions’ types.

In the literature, implicit typing is sometimes used in the a-la Curry calculus and explicit typing in the a-la Church calculus, but this is not strictly necessary.

Regardless of whether implicit or explicit typing is used, it is possible to reduce a typed $\lambda$-calculus expression using the traditional $\beta$ reduction rule of the untyped $\lambda$-calculus, after removing type annotations (“: $\alpha$” and similar) from bound variables. Following the “a-la Church” approach, this can only be done provided that the correct typing of the expression has been verified.

As an alternative, once a type is associated to each expression, the $\beta$ reduction rule must be updated to take it into account: if in the typeless $\lambda$-calculus

$$(\lambda x.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

in typed lambda calculus, reduction is possible only if “$x$” and “$E_1$” have the same type:

$$E : \alpha \Rightarrow (\lambda x : \alpha.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

.