Short Introduction to the Lambda-calculus

Franco Barbanera

with the collaboration of Claudia & Tiziana Genovese

and Massimiliano Salfi


Contents

1. Introduction to functional programming
   1.1 Introduction
   1.2 Functional programming
   1.3 Sketchy introduction to types
   1.4 Higher order functions and currying
   1.5 Recursion

2. The $\lambda$-calculus and computation theory
   2.1 Lambda-terms
   2.2 Free and bound variables
   2.3 Substitution
   2.4 The formalization of Notion of Computation in $\lambda$-calculus: the Theory of $\beta$-reduction
   2.5 Bracketing conventions
   2.6 Curried function
   2.7 Multi-step reduction
   2.8 Normalization and possibility of nontermination
   2.9 Some important results of $\beta$-reduction theory
   2.10 Formalizing the notion of equivalent programs: the theory of $\beta$-conversion

3. Representation of data type and operations in $\lambda$-calculus
   3.1 The Natural numbers
   3.2 The Booleans
   3.3 Arithmetic on Church's numerals
   3.4 Defining recursive functions using fixed points
   3.5 Reduction strategies
   3.6 Head normal forms

4. Types
   4.1 Introduction to types
   4.2 Type assignment 'a la Curry
   4.3 Principal pair
   4.4 The typed lambda calculus 'a la Church.
   4.5 Relation between the systems of Curry and Church.
4.5 Lambda-calculus extensions and their type systems a' la Curry.

1. Introduction to functional programming

1.1 Introduction

What does computing mean?
Roughly, to transform information from an implicit into an explicit form.

But this notion means nothing unless we make it precise, formal, by defining a computational model.
What is a Computational model?
A particular formalization of the notions connected to the concept of "computing".
The rather vague concept of computing can be formalized in several different ways, that is by means of several different computational models:
the Lambda-calculus, the Turing machine, etc.
"To program" means to specify a particular computation process, in a language based on a particular computational model.

Functional programming languages are based on the Lambda-calculus computational model.
Imperative programming languages are based on the Turing-machine computational model.

The fundamental concepts in imperative programming languages (coming from the Turing Machine model) are:

- store
- variable - something it is possible to modify ("memory cell")
- assignment
- iteration

In functional programming these concepts are absent (or, in case they are present, have a different meaning).
Instead, in it we find:

- expression
• recursion - different from the notion of iteration
• evaluation - different from the notion of execution:
  o when we evaluate a mathematical expression we do not modify anything.
• variable - intended in a mathematical sense:
  o unknown entity
  o abstraction from a concrete value

1.2 Functional programming

A program in a functional language consists in a set of function definitions and an expression (usually formed using the defined functions). The Interpreter of the language evaluates this expression by means of a discrete number of computation steps, making its meaning explicit (as pointed out before, during a computation, nothing is actually created, but - a fact extremely clear in functional programming - the representation of the information is "transformed").

A function definition in functional programming is not simply a means to uniquely identify a mathematical function on a domain of objects.

Example: a mathematical definition of two functions

\[ f: R \rightarrow R, \quad g: R \rightarrow R \]
\[ f'' - 6g' = 6 \sin x \]
\[ 6g'' + a^2 f' = 6 \cos x \]
\[ f(0) = 0, \quad f'(0) = 0, \quad g(0) = 1, \quad g'(0) = 1 \]

This set of equations precisely identify two precise \( f \) and \( g \).

In the above example \( f \) and \( g \) are uniquely identified by the set of differential equation, but such a definition is not useful from a computational point of view.

A function definition in a functional programming language, instead, has also to show how to compute (evaluate) the defined function on a given argument.

Example: function definition in Scheme and in Haskell

\[(\text{define Inctwo} \ (\lambda n \ (+ \ n \ 2)))\]
\[\text{Inctwo} \ n = n + 2\]

The evaluation of the application of a (user-defined) function to an argument consists in evaluating the expression consisting in the body of the function in which the formal parameter is replaced by the actual parameter (the argument).

To evaluate an expression means to first look for a sub-expression which is the application of a function to some arguments and then to evaluate such an application. In this way, in a sense, the meaning of the whole expression is made more and more explicit.

Example: evaluating (Inctwo 3)
Inctwo is a user defined function, and it is applied to an argument, so we take the body of the function, replace the formal parameter \( n \) by the actual parameter \( 3 \), obtaining

\[ 3 + 2, \text{ and then we evaluate } 3 + 2 \text{ obtaining } 5 \]

In the example above we have seen two kinds of computational steps from \( \text{Inc} \ 3 \) to \( 5 \). In the first one we have simply substituted in the body of the function the formal parameter by the actual parameter, the argument, and in the second one we have computed a predefined function. We shall see that the essence of the notion of computation in functional programming is indeed embodied by the first type of computational step. The simple operation of replacing the arguments of a function for its formal parameters in its body contains all the computing power of functional programming. Any sort of computation in functional programming could indeed be defined in terms of this one. We shall see this later, when we shall represent this sort of computational step with the beta-reduction rule in the lambda calculus and we shall see that, if one wish, any possible computation can be defined in terms of this beta-rule.

When we evaluate an expression there are sometimes more than one sub-expression one can choose and hence different possible evaluation paths can be followed. An evaluation strategy is a policy enabling to decide on which sub-expression we shall perform the next evaluation step. Some evaluation strategies could lead me in a never-ending path (this fact is unavoidable); but I would like to be sure that all finite path do lead me to the very same value (this will be guaranteed by the Confluence Theorem of the Lambda-calculus).

**Example: In Haskell**

```haskell
inf = inf  // definition of the value inf
alwaysSeven x = 7  // definition of the function alwaysSeven
alwaysSeven inf  // expression to be evaluated
```

Two possible choices in the evaluation of alwaysSeven \( \text{Inf} \):

1. We evaluate \( \text{inf} \) first (the evaluation strategy of Scheme).
2. We evaluate \( \text{alwaysSeven inf} \) first (the evaluation strategy of Haskell, a lazy language).

**case 1. Replace \( \text{inf} \) by \( \text{inf} \) \( \Rightarrow \) we go on endlessly**

**case 2. Replace alwaysSeven \( \text{inf} \) by \( 7 \) \( \Rightarrow \) we terminate in a single step.**

One of the distinguishing features in a programming language is the evaluation strategy it uses.

We know how to define the function factorial in a recursive way. Let us give now an example of mutually recursive functions.

```haskell
even x = if x = 0 then true
         else odd(x - 1)
odd x = if x = 0 then false
       else even(x - 1)
```

With our recursive definitions we do not only specify functions, but we also provide a method to compute them: a recursive definition definition says that the left-hand side part of
the equation is equivalent (has the same "meaning") of the right-hand side. The right-hand side, however, has its meaning expressed in a more explicit way. We can replace the left-hand side part with the right-hand one in any expression without modifying its meaning. We can notice that in order to evaluate "even 3" we can follow different evaluation strategies. It is also possible to perform more reductions in parallel, that is to replace two or more sub-expressions with equivalent sub-expression (reduction step) at the same time. A similar thing is not possible in imperative languages since we need to take into account the problem of side effects: running in parallel the instructions of the subprograms computing two functions \(F_1\) and \(F_2\) can produce strange results, since the two subprograms could use (and modify) the value of global variables which can affect the results produced by the two functions. Of course, because of the presence of global variables, the value of a function can depend on time and on how many times we have computed it before (in general in imperative languages it is difficult to be sure that we are programming actual mathematical functions.) We have not the above mentioned problems in functional programming languages, since there not exists side effects (there are no global modifyable variables). What we define are therefore actual mathematical functions; hence in an expression the value of any sub-expression does not depend on the way we evaluate it or other ones, that is the same expression always denotes the same value (a property called referential transparency).

1.3 Sketchy introduction to types

Some functional programming languages are typed. A type, in general, can be looked at as a partial specification of a program. These partial specifications, according to the particular type sytem of the language considered, can be more or less detailed.

By means of types we can avoid errors like:

\[
true + 3
\]

in fact the interpreter knows that

\[
+: : \text{int} \times \text{int} \rightarrow \text{int}
\]

In (strongly) typed languages this error is detected by the Type Checker, before the program is "executed". In not (strongly) typed languages this error is detected at run-time, by the interpreter itself (the program's execution is stopped whenever an error is detected). In typed languages not only the functions have a type, but all the possible correct expressions have (or can be given) a type.

In some typed functional languages it is possible to define polymorphic functions, that is function which, roughly, have the same behaviour on arguments of different types (a polymorphic type describes a whole set of (uniform) behaviours).

**Example:**

\[
\text{ident}:: \, a \rightarrow a \, \text{(a is a type-variable. It denotes a "generic" type.)}
\]

\[
\text{ident} \, x = x
\]
We shall study very simple types, like int->bool (it simply specifies that a function (program) with this type has an integer as an argument and a boolean as result). Notice, however, that there exist type systems so expressive that they can specify in a very precise way what a function computes. For instance there could be type systems with a type like

\[
\text{Forall } x : \text{Int Exists } y : \text{Int . } y = x + 1
\]

A function with this type have integers as argument, integers as results, and computes the successor function (of course the type does not specify how the successor of a given integer is computed).

### 1.4 Higher order functions and currying

In functional programming languages functions are first-class entities, i.e. can be treated as normal data. Hence they can be given as argument to other functions or be the result of a function.

A simple example is the following Scheme function

\[
(\text{define High-inc (lambda (x) (lambda (y) (+ x y)))})
\]

High-inc takes a number x and returns a function that increments its argument by x. The function Inc two seen before can be seen as the result of applying 2 to High-inc. As a matter of fact we could have defined Inc two in the following alternative way:

\[
(\text{define Inc two (High-inc 2)})
\]

Function that can have functions as argument or result are called higher-order functions.

The notion of higher-order function enables us to introduce the notion of currying, a relevant concept in functional languages. To curry means to transform a function f of arity n, in an higher-order function f_{cur} of arity 1 such that also f_{cur}(a_1) is a higher order function of arity 1, and so are f_{cur}(a_1)(a_2) up to f_{cur}(a_1)(a_2)\ldots(a_{n-1}) and such that f(a_1,\ldots,a_n)=f_{cur}(a_1)\ldots(a_n).

A binary function of type:

\[
A \times B \rightarrow C
\]

is currifyed to a function of type:

\[
A \rightarrow (B \rightarrow C)
\]

where any function arity is 1. (Higher-order functions and currying are treated in Section 1.7 of the reference below.)

We can notice that the function High-inc is indeed the currified version in Scheme of the addition.

The possibility of using currified functions definitely improves the expressive power of
functional languages (that is why in Haskell all functions are already in their curried version, even the used-defined ones.)

Note: Also in imperative languages it is possible to program functions which have functions as arguments, but in general what is given as argument is simply a pointer to some code (which needs to be treated as such).

1.5 Recursion

The computational power of functional languages is provided by the recursion mechanism; hence it is natural that in functional languages it is much easier to work on recursively defined datatypes.

Example:

the datatype List is recursively defined as follows:
  a list is either [] (the empty list) or elem:list (the concatenation of an element with a list).

BinaryTree (unlabelled) is another example:
  an element of BinaryTree is either EmptyTree or a root at the top of two elements of BinaryTree.

Notice that there is a strong relation between the Induction principle as a mathematical proof tool and Recursion as a computation tool.

A property is true (for natural numbers) if

- One proves the base case;
- By assuming the property to hold for n, one proves the property for n+1

This is the principle of numerical induction; it closely corresponds to the mechanism of defining a function on numbers by recursion: A function (on numbers) is defined if

- One defines it for the base case;
- By assuming the function to be defined for n, one defines it for n+1

Of course we can use recursion on complex recursive (inductive) data types. And in fact there exist general versions of the induction principle, as we shall see in the course.

Data are not modified in (pure) functional programming

We are used to work on data structures that are intrinsically tied to the notion of "modification", so we usually thing that an element of a data type can be modified. This is not true in functional programming. For example, a functional program that takes as input a numeric labelled tree and increments its labels, does not modify the input. It simply builds a new tree with the needed characteristics. The same holds when we push or pop elements from a stack: each time a new stack is generated.

This causes a great waste of memory (the reason why in implementations of functional programming languages a relevant role is played by the Garbage Collector). Of course there
can be optimizations and tricks used in implementations that can limit this phenomenon.

---

Text to use: R. Plasmeijer, M. van Eekelen, "Functional Programming and Parallel Graph Rewriting", Addison-Wesley, 1993. Cap. 1 (basic concepts)

2. The λ-calculus and computation theory

2.1 Lambda-terms

The fundamental concepts the Lambda-calculus is based on are:

- **variable**  \( (\text{formalisable by } x, y, z, \ldots) \)
- **abstraction**  \( (\text{formalisable by } \lambda x. M \text{ where } M \text{ is a term and } x \text{ a variable}) \)
- **application**  \( (\text{formalizable by } M N, \text{ where } M \text{ and } N \text{ are terms}) \)

We have seen that these are indeed fundamental notions in functional programming. It seems that there are other important notions: basic elements, basic operators and the possibility of giving names to expressions, but we shall see that the three above notions are the real fundamental ones and that we can found our computational model just on these three concepts.

Using the formalization of these three concepts we form the lambda-terms. These terms will represent both *programs* and *data* in our computational model (whose strengh strongly depends on the fact it does not distinguish between "programs" and "data".)

The formal definition of the terms of the Lambda-calculus (lambda-terms) is given by the following grammar:

\[ \Lambda ::= X \mid (\Lambda \Lambda) \mid \lambda X. \Lambda \]

where \( \Lambda \) stands for the set of lambda-terms and \( X \) is a metavariable that ranges over the (numerable) set of variables \( (x, y, v, w, z, x_2, x_2, \ldots) \)

*Variable* and *application* are well-known concepts, but what is abstractation?

In a sense, it is a notion tied to that of "abstracting from a concrete value" (i.e. disregarding the specific aspects of a particular object, which, in our case, is an input.)

Abstraction is needed to create anonymous functions (i.e. functions without a name). When we define in Haskell the function fact, \( \text{fact } n = \text{if } (n=0) \text{ then } 1 \text{ else } n \times \text{fact}(n-1) \) we are doing indeed two things: we are specifying a function and we are giving the name fact to it.

Being able to specify a function is essential in our computational model, but we can avoid to introduce in our model the notion of "giving a name to a function". You could ask: how is it possible to define a function like fact without being able to refer to it through its name???

We shall see.

*Example of (functional) abstraction:*

*We can specify the square function by saying that it associates 1 to 1*\(1\), 2 to 2*\(2\), 3 to 3*\(3\), ecc. In the specification of the function the actual values of the possible particular inputs are not needed, that is we can abstract from the actual values of the input by simply saying that the function square associates x*x to the generic input x.*
We could use the mathematical notation \( x \mapsto x^*x \) to specify such a particular relation between domain and codomain; with such mathematical notation we specify a function without assigning to it a particular name (i.e. we specify an anonymous function).

In Lambda-calculus we describe the relation input-output by means of the lambda-abstraction operator: the term \( \lambda x. M \) represents the anonymous function that, given an input \( x \), returns the "value" of the body \( M \). Other notions, usually considered as primitive, like basic operators (+, *, -, ...) or natural numbers, are not strictly needed in our computational model (indeed we shall see that they can actually be treated as derived concepts.)

In an abstraction \( \lambda x.P \), the term \( P \) is said to be the scope of the abstraction.

### 2.2 Free and bound variables

A variable \( x \) is said to be bound in \( M \) whenever it is abstracted in some subterm \( \lambda x.P \) of \( M \). A variable is free in \( M \) if it is not bound. These two concepts can be formalized by the following definitions.

**Definition of bound variable**
We define \( BV(M) \), the set of the Bound Variables of \( M \), by induction on the term as follows:

\[
\begin{align*}
BV(x) & = \emptyset \text{ (the emptyset)} \\
BV(PQ) & = BV(P) \cup BV(Q) \\
BV(\lambda x.P) & = \{x\} \cup BV(P)
\end{align*}
\]

**Definition of free variable**
We define \( FV(M) \), the set of the Free Variables of \( M \), by induction on the term as follows:

\[
\begin{align*}
FV(x) & = \{x\} \\
FV(PQ) & = FV(P) \cup FV(Q) \\
FV(\lambda x.P) & = FV(P) \setminus \{x\}
\end{align*}
\]

Free and bound variables represent two different concepts in a term. A bound variable in the body of an abstraction represents the argument of a function, whereas a free variable represent an (unknown) value.

### 2.3 Substitution

The notation \( M [L / x] \) denotes the term \( M \) in which any free occurrence of the variable \( x \) in \( M \) is replaced by the term \( L \).

**Definition of substitution** (by induction on the structure of the term)
1. If M is a variable (M = y) then:

\[
y [L/x] \equiv \begin{cases} 
L & \text{if } x = y \\
y & \text{if } x \neq y 
\end{cases}
\]

2. If M is an application (M = PQ) then:

\[PQ [L/x] = P[L/x] Q[L/x]\]

3. If M is a lambda-abstraction (M = \(\lambda y.P\)) then:

\[
\lambda y.P [L/x] \equiv \begin{cases} 
\lambda y.P & \text{if } x = y \\
\lambda y.(P [L/x]) & \text{if } x \neq y 
\end{cases}
\]

Notice that in a lambda abstraction a substitution is performed only if the variable to be substituted is free (not bound).

The definition of substitution given above, however, is not completely correct. Indeed it does not prevent some harmful ambiguities and errors:

**Example:**

Both terms \((\lambda x.z)\) and \((\lambda y.z)\) obviously represent the constant function which returns z for any argument.

Let us assume we wish to apply to both of these terms the substitution \([x/z]\) (we replace x for the free variable z). If we do not take into account the fact that the variable x is free in the term x and bound in \((\lambda x.z)\) we get:

\[
(\lambda x.z) [x/z] \Rightarrow \lambda x.(z[x/z]) \Rightarrow \lambda x.x \quad \text{representing the identity function}
\]

\[
(\lambda y.z) [x/z] \Rightarrow \lambda y.(z[x/z]) \Rightarrow \lambda y.x \quad \text{representing the function that returns always x}
\]

This is absurd, since both the initial terms are intended to denote the very same thing (a constant function returning z), but by applying the substitution we obtain two terms denoting very different things.

Hence a necessary condition in order the substitution \(M[L/x]\) be correctly performed is that \(\text{FV}(L)\) and \(\text{BV}(M)\) be disjoint sets.

**Example:**

Let us consider the following substitution:
\[(\lambda z. x) [zw/x]\]

The free variable \(z\) in the term to be substituted would become bound after the substitution. This is meaningless. So, what has to be done is to rename the bound variables in the term on which the substitution is performed, in order the condition stated above be satisfied:

\[(\lambda q.x) [zw/x]\]

now, by performing the substitution, we correctly get: \((\lambda q.zw)\)

The example above introduces and justifies the notion of \textit{alpha-convertibility}

The notion of \textit{alpha-convertibility}
A bound variable is used to represent where, in the body of a function, the argument of the function is used. In order to have a meaningful calculus, terms which differ just for the names of their bound variables have to be considered identical. We could introduce such a notion of identity in a very formal way, by defining the relation of \textit{\(a\)-convertibility}. Here is an example of two terms in such a relation.

\[
\lambda z.z =_a \lambda x.x
\]

We do not formally define here such a relation. For us it is sufficient to know that two terms are \(a\)-convertible whenever one can be obtained from the other by simply renaming the bound variables.

Obviously alpha convertibility maintains completely unaltered both the meaning of the term and how much this meaning is explicit. From now on, we shall implicitly work on \(\lambda\)-terms modulo alpha conversion. This means that from now on two alpha convertible terms like \(\lambda z.z\) and \(\lambda x.x\) are for us the \textit{very same} term.

Example:

\[
(\lambda z. ((\lambda t. tz) z)) z
\]

All the variables, except the rightmost \(z\), are bound, hence we can rename them (apply \(a\)-conversion). The rightmost \(z\), instead, is free. We cannot rename it without modifying the meaning of the whole term.

It is clear from the definition that the set of free variables of a term is invariant by \(a\)-conversion, the one of bound variables obviously not.

2.4 The formalization of the notion of computation in \(\lambda\)-calculus: The Theory of \(\beta\)-reduction

In a functional language, to \textit{compute} means to \textit{evaluate} expressions. The process of evaluation makes the meaning of a given expression more and more explicit. Up to now we
have formalized in the lambda-calculus the notions of programs and data (the terms). Let us see now how to formalize the notion of computation. In particular, the notion of "basic computational step".
In the introduction we noticed that, during the evaluation of a term, whenever there is a function applied to an argument, this application can be replaced by the body of the function in which the formal parameter is replaced by the actual argument.
So, if we call **redex** any term of the form \((\lambda x.M)N\), and we call \(M[N/x]\) its **contractum**, to perform a basic computational step on a term \(P\) means that \(P\) contains a subterm which is a redex and that such a redex is replaced by its contractum (\(\beta\)-reduction is applied on the redex.)

**Definition (The notion of \(\beta\)-reduction)**
The relation between redexes and their contracta is called **notion of \(\beta\)-reduction**, and it is formally represented by the following \(\beta\)-rule: \((\lambda x.M) N \rightarrow_{\beta} M [N/x]\)

**Definition (One step \(\beta\)-reduction)**
We say that a term \(P\) \(\beta\)-reduces in **one-step** to a term \(Q\) (notation: \(P \rightarrow_{\beta} Q\)) if \(Q\) can be obtained out of \(P\) by replacing a subterm of \(P\) of the form \((\lambda x.M)N\) by \(M[N/x]\)

**Example:**

If we had the multiplication in our system, we could perform the following computation steps

\((\lambda x. x*x) 2 \rightarrow 2*2 \rightarrow 4\)

Here the first step is a \(\beta\)-reduction; the second computation is the evaluation of the multiplication function. We shall see how this last computation can indeed be realized by means of a sequence of basic computational steps (and how also the number 2 and the function multiplication can be represented by \(\lambda\)-terms).

**Example:**

\((\lambda z. (zy) z) (x (xt)) \rightarrow_{\beta} (x (xt) y) (x (xt))\)

**Example:**

\((\lambda y. zy) (xy) \rightarrow_{\beta} z(xy)\)

here it could be misleading the fact that the variable \(y\) appears once free and once bound. As a matter of fact these two occurrences of \(y\) represent two very different things (in fact the bound occurrence of \(y\) can be renamed without changing the meaning of the term, the free one not). Notice that in this case, even if there is this small ambiguity, the beta reduction can be performed without changing the name of the bound variables. Why?

**A formal system for the one-step beta-reduction relation**

If one wishes to be very precise, it is possible to define the relation of one-step beta reduction by means of a formal system.
First recall roughly what a formal system is.

**Definition**  
A formal system is composed by axioms and inference rules enabling us to deduce valid "Judgements"  

**Example:**  
*In logics we have formal systems enabling to infer the truth of propositions, that is the judgments we derive are of the form "P is true".*

We can introduce a formal system whose only axiom is the $\beta$-rule we have seen before:

$$(\lambda x. M) N \rightarrow_\beta M \ [N/x]$$

and whose inference rules are the following ones:

\[
\begin{array}{ccc}
\text{if} & P \rightarrow_\beta P' & P \rightarrow_\beta P' & P \rightarrow_\beta P' \\
\text{then we can infer} & \lambda x. P \rightarrow_\beta \lambda x. P' & \text{L } P \rightarrow_\beta \text{ L } P' & \text{P L } \rightarrow_\beta \text{ P' L}
\end{array}
\]

The judgements we derive in the formal system defined above have the form $M \rightarrow_\beta M'$ and are interpreted as "the term $M$ reduces to the term $M'$ in one step of computation ($\beta$-reduction)".

**Example:**  
*in this formal system we can have the following derivation:*

\[
\begin{align*}
\lambda w.((zz) ((\lambda x. y)) y) & \rightarrow \lambda w.((zz) y) \\
\end{align*}
\]

This is a (very formal) way to show that the term $\lambda w.((zz) ((\lambda x. y)) y)$ reduces in one step to the term $\lambda w.((zz) y)$ (less formally, and more quickly, we could have simply spotted the redex $((\lambda x. y) y$ inside the term and replaced it by its contractum $y$)

So, our formal system defines a binary relation on lambda-terms, beta reduction, and we have seen that this can be done informally by saying that a term $M$ is in the relation $\rightarrow_\beta$
with a term N (M β-reduces with one step to N) if N can be obtained by replacing a subterm of M of the form \((\lambda x.P)Q\) by \(P[Q/x]\)

### 2.5 Bracketing conventions

For readability sake, it is useful to abbreviate nested abstractions and applications as follows:

\[
(\lambda x_1. (\lambda x_2. \ldots (\lambda x_n. M) \ldots)) \text{ is abbreviated by } (\lambda x_1 x_2 \ldots x_n. M)
\]

\[
...( (M_1 M_2) M_3) \ldots M_n) \text{ is abbreviated by } (M_1 M_2 M_3 \ldots M_n)
\]

We also use the convention of dropping outermost parentheses and those enclosing the body of an abstraction.

**Example:**

\[
(\lambda x. (\lambda y. (y))) \text{ can be written as } \lambda x. (\lambda y. y)
\]

### 2.6 Curried functions

Let us see what's the use of the notion of currying in the lambda-calculus.

As we mentioned before, if we have, say, the addition function, is is always possible to use its curried version \textit{AddCurr} : \(N \rightarrow (N \rightarrow N)\) and any time we need to use it, say \textit{Add}(3,5), we can use instead ((\textit{AddCurr}(3)) 5). This means that it is not strictly needed to be able to define functions of more than one argument, but it is enough to be able to define one argument functions. That is why the λ-calculus, that contains only the strictly essential notions of functional programming, has only one-argument abstraction.

Notice that, by using curried functions, not only we lose no computational power, but, even better, we get more expressive power. For instance, if we wish to use a function that increments an argument by 3, instead of defining it we can simply use \textit{AddCurr}(3).

Not all the functional languages have a built-in currying feature. Scheme has it not. In Scheme any function you define has a fixed arity, 1 or more. If you wish to use a curried version of a function you have to define it explicitly. In Haskell instead, all function are implicitly intended to be curried: when you define \(\text{Add} \ x \ y = x + y\) you are indeed defining \textit{AddCurr}.

### 2.7 Multi-step reduction

Strictly speaking, \(M \rightarrow N\) means that M reduces to N by exactly one reduction step. Frequently, we are interested in whether M can be reduced to N by any number of steps.
we write \( M \rightarrow N \) if \( M \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots \rightarrow M_k \equiv N \) (\( K \geq 0 \))

It means that \( M \) reduces to \( N \) in zero or more reduction steps.

**Example:**

\[(\lambda z.(zy))(\lambda x.x) \rightarrow y\]

### 2.8 Normalization and possibility of non termination

**Definition**

If a term contains no \( \beta \)-redex, it is said to be in **normal form**.

Example:

\[\lambda xy.y \text{ and } xyz \text{ are in normal form.}\]

A term is said to **have a normal form** if it can be reduced to a term in normal form.

Example:

\[(\lambda x.x)y \text{ is not in normal form, but it has the normal form } y\]

**Definition:**

A term \( M \) is **normalizable** (has a normal form, is weakly normalizable) if there exists a term \( Q \) in normal form such that

\[M \rightarrow Q\]

A term \( M \) is **strongly normalizable**, if there exists no infinite reduction path out of \( M \). That is, any possible sequence of \( \beta \)-reductions eventually leads to a normal form.

**Example:**

\[(\lambda x.y)((\lambda z.z)t)\]

is a term not in normal form, normalizable and strongly normalizable too. In fact, the only two possible reduction sequences are:

1. \((\lambda x.y)((\lambda z.z)t) \rightarrow y\)

2. \((\lambda x.y)((\lambda z.z)t) \rightarrow (\lambda x.y)t \rightarrow y\)

To **normalize** a term means to apply reductions until a normal form (if any) is reached.
Many \( \lambda \)-terms cannot be reduced to normal form. A term that has no normal form is

\[(\lambda x.xx)(\lambda x.xx)\]

This term is usually called \( \Omega \).

Although different reduction sequences cannot lead to different normal forms, they can produce completely different outcomes: a reduction sequence could lead to a normal form while another could never terminate. Typically, if \( M \) has a normal form and admit also an infinite reduction sequence, it contains a subterm having no normal form but that can be erased by some reduction. For instance, the reduction

\[(\lambda y.a)\Omega \rightarrow a\]

reaches a normal form, by erasing the \( \Omega \). This corresponds to a \textit{call-by-name} treatment of arguments: the argument is not reduced, but substituted \textit{as it is} into the body of the abstraction.
The attempt to normalize the argument, instead, generates an endless reduction sequence:

\[(\lambda y.a)\Omega \rightarrow (\lambda y.a)\Omega \rightarrow \ldots\]

Evaluating the argument before its substitution into the body of the function corresponds to a \textit{call-by-value} treatment of function application. In the two examples above, a call-by-value strategy never reaches the normal form.

One is naturally led to think that the notion of normal form can be looked at as the abstract formalization in the lambda-calculus of the notion of \textit{value} in programming languages.

This, however, is a too naive approach. Even if it is intuitively true that something that cannot be further evaluated can be considered as a value, the vice-versa does not hold \textit{a-priori}. As a matter of fact there can be many different interpretations of the notion of \textit{value}. In Scheme, for instance, a lambda expression \textit{is} a value: the interpreter does not evaluate any expression of the form \( \lambda y.M \).

Why does the interpreter of Scheme choose to consider a lambda abstraction as a value independently from what there is in its body? Well, otherwise it would not be possible to define in Scheme, say, a function that takes an argument and then never stops.

So we cannot formalize \textit{in general} the notion of value. What a value is has to be specified according to the context we are in. So, when we define a functional programming language we need to specify, among other things, also what we mean by \textit{value} in our language.

### 2.9 Some important results of \( \beta \)-reduction theory

The \textit{normal order} reduction strategy consists in reducing, at each step, the leftmost outermost \( \beta \)-redex. \textit{Leftmost} means, for instance, to reduce \( L \) (if it is reducible) before \( N \) in a term \( LN \). \textit{Outermost} means, for instance, to reduce \( (\lambda x.M)N \) before reducing \( M \) or \( N \) (if reducible). The Normal Order reduction is obviously a call-by-name evaluation strategy.

**Theorem (Standardization)**

If a term has a normal form, this is always reached by applying the normal order reduction strategy.
Theorem (Confluence)
If we have terms M, M2 and M3 such that:

\[ M \rightarrow M_1 \quad \text{and} \quad M \rightarrow M_2 \]

then there always exists a term L such that:

\[ M_1 \rightarrow L \quad \text{and} \quad M_2 \rightarrow L \]

That is, in diagram form:

![Diagram](image)

Corollary (Uniqueness of normal form)
If a term has a normal form, it is unique.

Proof:
Let us suppose to have a generic term M, and let us suppose it has two normal forms: N and N'. It means that

\[ M \rightarrow N \quad \text{and} \quad M \rightarrow N' \]

Let us now apply the Confluence theorem. Then there exists a term Q such that:

\[ N \rightarrow Q \quad \text{and} \quad N' \rightarrow Q \]

But N and N' are in normal form and hence they contain no redex. So, it necessarily means that

\[ N \rightarrow Q \quad \text{in 0 steps,} \]

and hence that \( N \equiv Q \).
Analogously we can show that \( N' \equiv Q \). It is then obvious that \( N \equiv N' \).

Definition
For a binary relation \( R \), the **diamond property** holds whenever:

if \( aRb \) and \( aRc \), then there exists \( d \) such that \( bRd \) and \( cRd \)

Using the definition above, we can reword the theorem of confluence in simpler way:
Note that:

\[ \Gamma \text{ DP}(\rightarrow) \]

The one-step reduction relation does not satisfy the diamond property, that is, it is not true that the following diagram always commutes

\[ M \]
\[ \leftarrow M_1 \]
\[ M_2 \]
\[ \\]
\[ L \]

In fact, consider the term \((\lambda x.x)(I \, a)\), where \(I \equiv \lambda x.x\).
In one step this term reduces to \((\lambda x.xx)a\) or \((I \, a)(I \, a)\). These terms can both be reduced to \(aa\), but there is no way to complete the diamond with a single-step reduction:

The problem, of course, is that \((\lambda x.xx)\) replicates its argument, which must then be reduced twice.

Sketch of the Proof of the Theorem of Confluence

2.10 Formalizing the notion of equivalent programs: the theory of \(\beta\)-conversion

In the actual programming activity we usually talk about "computational equivalence" of pieces of code or of whole programs (for instance when we try to do some optimizations.) Being able to formally deal with such a notion would be useful for many things, like safely modifying and optimizing programs or proving properties of programs.
In the present section, we formalize the notion of equivalence of programs into the lambda
calculus. In order to do that we can start by asking ourselves what does it means for two terms to be *computationally equivalent.* Informally, we could say that two terms are so when they "represent the same information". In such a case the following definition should be the right one.

**Definition** (The relation of *β*-conversion)

M is *β*- convertible to N, in symbols

\[ M =_{\beta} N \]

if and only if M can be transformed into N (or N into M) by performing zero or more β-reductions and β-expansions.

(A *β*-expansion being the inverse of a reduction: \[ P[Q/x] \rightarrow (\lambda x.P)Q \])

That is, two terms M and N are β-convertible if and only if there exist terms \( N_1, \ldots, N_k \) (\( k \geq 0 \)) such that

\[ M \rightarrow N_1 \leftarrow M_1 \rightarrow N_2 \leftarrow M_2 \ldots \ldots \rightarrow M_{k-1} \rightarrow N_k \leftarrow M_k \equiv N \]

Let us define now a formal system allowing to derive all, and only, the valid judgements concerning the beta-convertibility of terms, that is of the form \( M =_{\beta} N \) ("M is β-convertible to N").

**A formal system for β-conversion**

**Axiom:**

\[ (\lambda x.M) N =_{\beta} M[N/x] \]

**Inference rules:**

\[
\begin{align*}
\text{if} & \\
\text{then we can infer} & \\
\lambda x. P =_{\beta} \lambda x. P' & \quad P =_{\beta} P' & \quad P =_{\beta} P' \\
Q P =_{\beta} Q P' & \quad P Q =_{\beta} P' Q
\end{align*}
\]

β-conversion is reflexive, symmetric and transitive too, hence we must add also the following axiom and rules:

\[
\begin{align*}
M =_{\beta} M & \quad N =_{\beta} M & \quad L =_{\beta} M & \quad M =_{\beta} N \\
M =_{\beta} M & \quad N =_{\beta} M & \quad L =_{\beta} N
\end{align*}
\]

We have thus formally defined the relation of beta-conversion as the smallest equivalence relation containing the beta-rule and compatible with the operations of formation of terms.
(hence a congruence).
We have said before that such a relation aims at being a precise formalization of the notion of equivalence between (functional) programs, since, by its definition, it equates terms containing the very same "information".
Are we sure, however, that such "information" we are equating is actually a "computational information"? In order to answer to such a question, let us notice that we can look at any term as a function (this is reasonable, since any term can be applied to any other term). From such a point of view, what does it mean for two terms to represent the same "computational information"? Does it just mean that they represent the same function? i.e. that they "return" the same output when given the same input?
If this were the case, the notion represented by the relation of β-equivalence would be useless for us computer scientists, since we are not only concerned about what output is produced by a program, given a certain input, but we are also (and mainly) interested in "how" the result is computed. In other words, we are more interested in algorithms than in functions (a computer scientist would not consider a program implementing the bubblesort equivalent to a program implementing the mergesort, although they both compute the same function, the sorting one.)
Fortunately the β-conversion relation does not equate two terms just in case they compute the same function, as can be roughly shown by the following counterexample. Let us consider the two terms (\(\lambda x. y x\)) and \(y\). They actually compute the same function from terms to terms since, if we apply the same input, say \(M\), we get the very same output

1. \((\lambda x. y x)M \rightarrow y M\)
2. \(y M \rightarrow y M\)

Nonetheless \((\lambda x. y x)\) and \(y\) are not considered equivalent by the β-conversion relation, since it is easy to show, using the Church-Rosser theorem below, that

\[(\lambda x. y x) \not\approx_\beta y\]

So, informally, we can not only say that two β-convertible terms "contain the same information", but also that this information is actually a computational, algorithmic information.
It is not wrong then to state that the actual notion formalized by β-conversion is:

two terms are β-convertible if and only if they compute the same function using the same algorithm.

For a more convincing example, when later on we shall represent numerical mathematical function in the lambda-calculus, you will be able to check that both \(\lambda n. f. (f(n f x))\) and \(\lambda n. f. (n f (f x))\) compute the successor function on natural numbers. Nonetheless, being two distinct normal forms, they cannot be β-convertible (by the Church-Rosser theorem). You can notice in fact that they represent different "algorithms".

And in case we would like to look at lambda-terms from an extensional point of view? (that is from a mathematical point of view, looking at them as functions). Well, in case we were interested in equating terms just when they produce the same output on the same input, we
should consider a different relation: the beta-eta-conversion.

The theory of beta-eta-conversion (\(=_{\beta\eta}\))
The formal system for \(=_{\beta\eta}\) is like the one for \(=_{\beta}\), but it contains an extra axiom:

\[\lambda x. M x =_{\eta} M \text{ if } x \text{ is not in } \text{FV}(M)\]

Instead of adding this axiom, we could, equivalently, add the following inference rule, more intuitive, but with the drawback of having a universally quantified premise:

\[
\begin{align*}
\text{For all } L: & \quad ML = NL \\
\hline
\text{M} = N
\end{align*}
\]

Obviously, from a Computer Science point of view we are more interested in \(=_{\beta}\) than in \(=_{\beta\eta}\)

Example: In the theory of beta-eta-conversion it is possible to prove that

\[(\lambda x. y x) =_{\beta\eta} y\]

Remark: you could now wonder why the two non \(\beta\)-convertible terms considered before, \(\lambda n f x. (f(nf)x)\) and \(\lambda n f x. (n f (f x))\) cannot be made equal even in the beta-eta-conversion theory. Well, the lambda-calculus is a very abstract theory and a term has be looked at as a function on all possible terms, while the two terms above compute the same function only on a subset of the lambda-calculus terms, that is on those lambda-terms which represent natural number, the so called Church numerals we shall discuss shortly.

Let us go back to our \(\beta\)-conversion and let us have a look at a few relevant results of the theory of \(\beta\)-conversion.

Theorem (Church-Rosser)
If two terms are beta-convertible (\(M =_{\beta} N\)), then there exists \(Q\) such that

\[M \rightarrow \rightarrow Q \quad \text{and} \quad N \rightarrow \rightarrow Q\]

The Church-Rosser theorem enables to prove that the theory of \(\beta\)-conversion is consistent. In a logic with negation, we say that a theory is consistent, if one cannot prove a proposition \(P\) and also its negation \(\neg P\). In general, a theory is consistent if not all judgements are provable (i.e. there exists some judgments that are not provable). Thanks to the Church-Rosser theorem, we can show that the \(\lambda\)-calculus is consistent.

Corollary (Consistency of \(=_{\beta}\))
There exist at least two terms that are not \(\beta\)-convertible

Proof:
Let us consider the two terms \((\lambda x. y x)\) and \(y\) (any pair of distinct normal forms will work the same).
By contradiction, let us assume that

$$(\lambda x.y) \equiv \beta y$$

Then, by the Church-Rosser theorem, there exists a term $Q$ such that

$$(\lambda x.y) \rightarrow Q \quad \text{and} \quad y \rightarrow Q$$

But since the two terms are both in normal form, the above reductions must necessarily be 0-step reductions. This implies that

$$(\lambda x.y) \equiv y$$

This is impossible, hence

$$(\lambda x.y) \not\equiv \beta y$$

The Church-Rosser theorem and the Confluence theorem are equivalent and in fact are often used and referred to interchangeably:

*The Church-Rosser theorem (C-R) and the Theorem of confluence (CONF) are logically equivalent, that is it is possible to prove one using the other*

**Proof:**

**C-R $\Rightarrow$ CONF**

Let us assume the Church-Rosser theorem and prove Confluence. In order to prove confluence let us take $M$, $M_1$ and $M_2$ such that

$$M \rightarrow M_1 \quad \text{and} \quad M \rightarrow M_2$$

By definition of $\beta$-conversion we have then that $M_1 = \beta M_2$. We can now use the Church-Rosser theorem that guarantees that, whenever $M_1 = \beta M_2$, there exists $L$ such that

$$M_1 \rightarrow L \quad \text{and} \quad M_2 \rightarrow L$$

**CONF $\Rightarrow$ C-R**

Let us assume the Confluence theorem and prove the Church-Rosser one. We then consider two terms $M$ and $M'$ such that

$$M = \beta M'$$

By definition of beta-conversion, it means that I can transform $M$ into $M'$, by performing zero or more $\beta$-reductions / expansions:

$$M \rightarrow N_1 \leftarrow M_1 \rightarrow N_2 \leftarrow \cdots \rightarrow N_k \leftarrow \cdots \rightarrow \leftarrow M_k = M'$$
We can now use the Confluence theorem on the terms \( M_2, N_1 \) and \( N_2 \), inferring that there exists a term \( L_1 \) such that

![Diagram](image)

By iterating the process we get:

![Diagram](image)

We give now an important theorem of undecidability for the theory of beta conversion.

**Definition** (Closure by a relation)

A set \( S \) is said to be closed with respect to a binary relation \( R \) iff \( a \) belongs to \( S \) and \( aRb \) implies that \( b \) belongs to \( S \)

**Theorem** (Undecidability)

Let \( A \) be a set of lambda-terms which is not trivial (that is, it is not empty and it does not contain all the lambda-terms.) Then if \( A \) is closed by beta-conversion then it is not recursive (that is, it is not possible in general, given a term, to decide in an effective way whether it belongs to \( A \) or not).

### 3. Representation of data types and operations in \( \lambda \)-calculus

#### 3.1 The Natural numbers
In the syntax of the λ-calculus we have not considered primitive data types (the natural numbers, the booleans and so on), or primitive operations (addition, multiplication and so on), because they indeed do not represent really primitive concepts. As a matter of fact, we can represent them by using the essential notions already formalized in the λ-calculus, namely variable, application and abstraction.

The question now is: how can we represent in the λ-calculus a concept like that of natural number? We stressed that the λ-calculus is a theory of functions as algorithms. This means that if we wish to represent natural numbers in the λ-calculus, we need to look at them as algorithms. How can that be possible???? A number is a datum, not an algorithm! Well... but we could look at a number as an algorithm. For instance, we could identify a natural number with the procedure we use to "build" it. Let us consider the number 2. How can we "build" it? Well, give me the successor, give me the zero and I can get the number 2 by iterating twice the successor function on the zero. How can I represent this procedure? By the lambda-term \( \lambda fx. f(fx) \) !

Notice that the algorithm to build the number 2 is independent from what the zero and the constructor actually are. We can abstract from what they actually are (fortunately, since nobody really knows what zero and what the successor are. Do you?).

So, in general, we can represent natural numbers as lambda-terms as follows

\[
\begin{align*}
0 & \equiv \lambda fx. x \quad \text{(it represents the natural number 0)} \\
1 & \equiv \lambda fx. fx \quad \text{(it represents the natural number 1)} \\
2 & \equiv \lambda fx. f(fx) \quad \text{(it represents the natural number 2)} \\
2 & \equiv \lambda fx. f(fx) \quad \text{(it represents the natural number 3)} \\
\cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \\
n & \equiv \lambda fx. f^nx \quad \text{(it represents the natural number n)}
\end{align*}
\]

where we define \( f^nx \) as follows: \( f^0x = x \); \( f^{k+1}x = f(f^kx) \)

\( n \) denotes the lambda-representation of the number \( n \).

The above encoding of the natural numbers is the original one developed by Church. The terms just defined are called Church's numerals. Church's encoding is just one among the many ones that it is possible to devise.

The technique to think of data as algorithms can be extended to all the (inductively defined) data.

**Example:**

I can represent an unlabelled binary tree by the function that takes two inputs (the empty tree and the constructor that adds a root on top of two binary trees) and that returns as output the expression that represents how to build the desired tree out of the empty tree and the constructor.
In order to be sure that the representation of natural numbers we have given is a reasonable one, we have to show that it is also possible to represent in the lambda-calculus the operations on numbers. But what does it mean for a function on natural numbers to be *definable* in the lambda-calculus?

**Definition** (*λ*-definability)

A function \( f : \mathbb{N} \rightarrow \mathbb{N} \) is said to be *\( λ \)-definible* (representable in the \( λ \)-calculus) iff there exists a \( λ \)-term \( F \) such that, for all \( n \):

\[
F \ n =_β f(n)
\]

The term \( F \) is said to *represent* the function \( f \).

**Example:**

To represent the function factorial, we need to find a \( λ \)-term (let us call it \( \text{fact} \)) such that, whenever applied to the Church's numeral that represents the natural number \( n \), we get a term that is \( β \)-convertible to the Church's numeral representing the factorial of \( n \), that is

\[
\text{fact} \ 3 =_β 6
\]

Why have we not defined how to represent in the lambda-calculus the numerical functions with more that one argument? Well, since it is enough to be able to represent their curried versions. So the definition given is enough.

We shall see shortly how to define binary numerical functions like addition. What we shall do is to look at addition as \( \text{AddCurr} : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \)

Before seeing how to lambda-represent some numerical functions, let us see how we can represent the Booleans.

### 3.2 The Booleans

An encoding of the booleans corresponds to defining the terms \( \text{true} \), \( \text{false} \) and \( \text{if} \), satisfying (for all \( M \) and \( N \)):

\[
\text{if} \ \text{true} \ M \ N = M
\]
\[
\text{if} \ \text{false} \ M \ N = N
\]

The following encoding is usually adopted:

\[
\text{true} ≡ \lambda xy.x
\]
\[
\text{false} ≡ \lambda xy.y
\]
\[
\text{if} ≡ \lambda pxy}pxy
\]

As exercise, check that \( \text{if} \) actually works.

We have \( \text{true} ≠ \text{false} \) by the Church-Rosser theorem, since \( \text{true} \) and \( \text{false} \) are distinct
normal forms. All the usual operations on truth values can be defined using the conditional operator. Here are negation, conjunction and disjunction:

\[
\text{and} \equiv \lambda p q. \text{if } p \text{ q false} \\
\text{or} \equiv \lambda p q. \text{if } p \text{ true q} \\
\text{not} \equiv \lambda p. \text{if } p \text{ false true}
\]

3.3 Arithmetic on Church's numerals

Using Church's encoding of natural numbers, addition, multiplication and exponentiation can be defined very easily as follows:

\[
\text{add} \equiv \lambda m n f x. m f (n f x) \\
\text{mult} \equiv \lambda m n f x. (m f) x \\
\text{expt} \equiv \lambda m n f x. m (n f x)
\]

Addition is not hard to check. In fact:

\[
\text{add } m \ n \equiv (\lambda m n f x. m f (n f x)) m \ n \rightarrow_{\beta} (\lambda n f x. m f (n f x)) n \rightarrow_{\beta} \\
\rightarrow_{\beta} \lambda f x. m f (n f x) \equiv \lambda f x. (\lambda g x. g^m x) f ((\lambda h x. h^n x) f x) \rightarrow_{\beta} \\
\rightarrow_{\beta} \lambda f x. (\lambda g x. g^m x) f ((\lambda x. f^n x) x) \rightarrow_{\beta} \lambda f x. (\lambda g x. g^m x) f (f^n x) \rightarrow_{\beta} \\
\rightarrow_{\beta} \lambda f x. (\lambda x. f^n x)(f^n x) \rightarrow_{\beta} \lambda f x. f^{m+n} x \equiv \\
\equiv m+n
\]

Multiplication is slightly more difficult:

\[
\text{mult } m \ n \equiv (\lambda m n f x. m \ n f x) m \ n \rightarrow_{\beta} (\lambda n f x. m \ n f x) n \rightarrow_{\beta} \\
\rightarrow_{\beta} \lambda f x. m (n f x) \equiv \lambda f x. (\lambda g x. g^m x) (n f x) \rightarrow_{\beta} \\
\rightarrow_{\beta} \lambda f x. (n f)^m x \rightarrow \lambda f x. ((\lambda h x. h^n x) f)^m x \rightarrow_{\beta} \lambda f x. (f^n)^m x \equiv \lambda f x. f^{m \times n} x \equiv \\
\equiv m \times n
\]

These reductions hold for all Church's numerals \( m \) and \( n \).

Here are other simple definitions:

\[
\text{succ} \equiv \lambda n f x. f (n f x) \\
\text{iszero} \equiv \lambda n. n \ (\lambda x. \text{false}) \text{ true}
\]

The following reductions, as you can check by yourself, hold for any Church's numeral \( n \):

\[
\text{succ } n \rightarrow n + 1 \\
\text{iszero } 0 \rightarrow \text{ true} \\
\text{iszero } (n + 1) \rightarrow \text{ false}
\]
Of course it is also possible to define the predecessor function (pred) in the lambda calculus, but we do not do that here.

**Example:**
We can easily give the representation of the following mathematical function on natural numbers

\[
\begin{align*}
f(n,m) &= \begin{cases} 
3+(m*5) & \text{if } n=0 \\
(n+(3*m)) & \text{otherwise}
\end{cases}
\end{align*}
\]

by means of the following lambda-term:

\[
\lambda n m. \text{if} \ iszero \ n \ (\text{add} \ 3 \ (\text{mult} \ m \ 5)) \ (\text{add} \ n \ (\text{mult} \ 3 \ m))
\]

### 3.4 Defining recursive functions using fixed points

In the syntax of the \(\lambda\)-calculus there is no primitive operation that enables us to give names to lambda terms. However, that of giving names to expressions seems to be an essential feature of programming languages in order to define functions by recursion, like in Scheme

\[
\text{(define fact (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))))}
\]

or in Haskell

\[
\text{fact 0 = 1}
\]

\[
\text{fact n = n * fact (n-1)}
\]

Then, in order to define recursive functions, it would seem *essential* to be able to give names to expressions, so that we can recursively refer to them. Surprisingly enough, however, that's not true at all! In fact, we can define recursive functions in the lambda-calculus by using the few ingredients we have already at hand, exploiting the concept of *fixed point*.

**Definition**

Given a function \( f : A \to A \), a fixed point of \( f \) is an element \( a \) of \( A \) such that \( f(a) = a \).

In the lambda-calculus, since we can look at any term \( M \) as a function from lambda-terms to lambda-terms, i.e. \( M : \Lambda \to \Lambda \), we can define a *fixed point* of \( M \), if any, to be any term \( Q \) such that:

\[
M Q \equiv Q
\]
To see how we can fruitfully exploit the concept of fixed point, let us consider an example. Let us take the definition of factorial using a generic syntax

\[
factorial \ x = \text{if} \ (x=0) \ then \ 1 \ else \ x \ * \ factorial(x-1)
\]

The above equation can be considered a definition because it identifies a precise function. In fact the function factorial is the only mathematical function which satisfies the following equation, where \( f \) can be looked at as an "unknown" function

\[
f x = \text{if} \ x=0 \ then \ 1 \ else \ x \ * \ f(x-1)
\]

The above equation can be rewritten as follows, by using the lambda notation

\[
f = \lambda x. \text{if} \ x=0 \ then \ 1 \ else \ x \ * \ f(x-1)
\]

The function we are defining is hence a fixpoint, the fixpoint of the functional

\[
\lambda f. \lambda x. \text{if} \ x=0 \ then \ 1 \ else \ x \ * \ f(x-1)
\]

In fact it can be checked that

\[
(\lambda f. \lambda x. \text{if} \ x=0 \ then \ 1 \ else \ x \ * \ f(x-1)) \ factorial = factorial
\]

Now, we want to lambda-represent the factorial function. It is easy to see that the functional

\[
\lambda f. \lambda x. \text{if} \ x=0 \ then \ 1 \ else \ x \ * \ f(x-1)
\]

can be easily lambda-represented:

\[
\lambda f. \lambda x. \text{if} \ (iszero \ x) \ then \ (mult \ (f(sub \ x \ -1))) \ else \ x
\]

Then, a lambda term \( F \) that lambda-represents the factorial function must be a fixed point of the above term. Such an \( F \) would in fact satisfy the following equation

\[
F \ x =_\beta \text{if} \ (iszero \ x) \ then \ (mult \ (f(sub \ x \ -1))) \ else \ x
\]

and hence would be really the lambda-representation of the factorial.

But is it possible to find fixed points of lambda-terms? And for which lambda-terms? Well, the following theorem settles the question once and for all, by showing us that indeed all lambda-terms have a fixed point. Moreover, in the proof of the theorem, it will be shown how easy it is possible to find a fixed point for a lambda term.

**Theorem**
Any \( \lambda \)-term has at least one fixed point.

**Proof.**
We prove the theorem by showing that there exist lambda terms (called fixed point
operators) which, given an arbitrary lambda term, produce a fixed point of its. One of those operators (combinators) is the combinator \( Y \) discovered by Haskell B. Curry. It is defined as follows:

\[
Y \equiv \lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))
\]

It is easy to prove that \( Y \) is a fixed point combinator. Let \( F \) be an arbitrary \( \lambda \)-term. We have that

\[
YF \equiv (\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))) F \rightarrow (\lambda x. F(xx)) (\lambda x. F(xx)) \rightarrow F ((\lambda x. F(xx)) (\lambda x. F(xx))) \leftarrow \tag{3}
\]

\[
\leftarrow F((\lambda f.(\lambda x. f(xx)) (\lambda x. f(xx)))F) \equiv F (YF)
\]

That is

\[
F (YF) =_\beta (YF)
\]

Then the \( \lambda \)-term that represents the factorial function is:

\[
Y(\lambda f.\lambda x.\text{if (iszero x)} \downarrow (\text{mult x} (f(\text{sub x} 1))))
\]

Let us consider another example of recursively defined function.

\[
g x = \text{if (x=2)} \text{ then 2 else } g x \tag{\*}
\]

What is the function defined by this equation? Well, it is the function that satisfies the equation, where \( g \) is considered an "unknown" function.

Ok, but we have a problem here, since it is possible to show that there are many functions that satisfy this equation. In fact, it is easy to check that

\[
\text{factorial} \ x = \text{if (x=2)} \text{ then 2 else } \text{factorial} \ x
\]

And also the identity function satisfies the equation, since

\[
\text{identity} \ x = \text{if (x=2)} \text{ then 2 else } \text{identity} \ x
\]

And many more. For instance, let us consider the function \( \text{gee} \) that returns 2 when \( x=2 \) and that is undefined otherwise. Well, also \( \text{gee} \) satisfies the equation:

\[
g x = \text{if (x=2)} \text{ then 2 else } \text{gee} \ x
\]

So, which one among all the possible solutions of (\*), that is which one among all the
possible fixed point of the functorial

\[ \lambda g. \lambda x. \text{if } x=2 \text{ then } 2 \text{ else } g \ x \]

We all know that it is \emph{gee} the intended function defined by \((\ast)\). Why?

Because implicitly, when we give a definition like \((\ast)\), we are considering a computational process that identifies the \emph{least} among all the possible solutions of the equation (least in the sense of "the least defined"). This means that, in order to correctly lambda-define the function identified by an equation like \((\ast)\), we cannot consider any fixed point, but the \emph{least} fixed point (the one that corresponds to the least defined function that satisfies the equation.)

The question now is: does the \(Y\) combinator \emph{picks} the right fixed point? If a lambda term has more than one fixed point, which is the one \emph{picked up} by \(Y\)?

Well, fortunately \(Y\) behaves really well. It not only returns a fixed point of a term, but the least fixed point, where, informally, by "least" we mean the one containing only the information strictly needed in order to satisfy the equation. Of course we could formalize in a precise mathematical sense what is the meaning, for a term, to have "less information" than another one, and hence what is the precise meaning of being a \emph{least} fixed-point. But such a formalization would go beyond the scope of these introductory course notes.

It is enough for us to know that the least fixed point of the functional

\[ \lambda f. \lambda x. \text{if } \text{iszero} \ (\text{pred} \ (\text{pred} \ x)) \ 2 \ (f \ x) \]

and hence the lambda term lambda-representing the function \emph{gee} is

\[ Y(\lambda f. \lambda x. \text{if } \text{iszero} \ (\text{pred} \ (\text{pred} \ x)) \ 2 \ (f \ x)) \]

Let us look at some other examples

\textbf{Example:}

Let us consider the \(\lambda\)-term

\[ \lambda x. x \]

It is immediate to check that any term \(M\) is a fixed point of \(\lambda x. x\) (the identity function) because

\[ (\lambda x. x)M =^\beta M \]

The least fixed point of \(\lambda x. x\) is \(\Omega\), since

\[ Y \ (\lambda x. x) = \Omega \]

It is reasonable that \(\Omega\) be the least among all the lambda-terms (any lambda-term is a fixed point of the identity) since this term is intuitively the one "containing" the least information (actually no information at all).

\textbf{Example:}

Let us consider the function that diverges on any element of its domain. It can be defined by the following recursive equation:

\[ h \ x = h \ x \]
The $\lambda$-term that represents the function $h$ is the solution of the equation

$$h = \lambda x. h x$$

that is, the least fixed point of

$$\lambda f. \lambda x. f x$$

The least fixed point of this term is hence the lambda-representation of the function that diverges everywhere.

Let us go back now to the Figure (3), where $Y$ F is proved to be beta convertible to $F(Y$ F) by means of two $\beta$-reductions followed by one $\beta$-expansion. Notice that it is impossible to have

$$Y F \rightarrow F (Y F)$$

If we wish to have such a behaviour we have to look for a different fixed point combinator, for instance Alan Turing's $\Theta$, which is defined in the following way:

$$A \equiv \lambda xy. y (xx)$$
$$\Theta \equiv AA$$

Its peculiarity is that we have that

$$\Theta F \rightarrow F (\Theta F)$$

**The Why of $Y$.**

We now show, in an intuitive way, why the combinator $Y$ (and the fixed point combinators in general) works.

Let us consider once again the equation that defines the factorial function (and that at the same time provides a way to compute it.)

$$\text{factorial } x = \text{if } (x=0) \text{ then } 1 \text{ else } x * \text{factorial}(x-1)$$

From this equation we can obviously get

$$\text{factorial } x = \text{if } (x=0) \text{ then } 1 \text{ else } x * ( \text{if } (x-1=0) \text{ then } 1 \text{ else } (x-1) * \text{factorial}(x-1-1) )$$

and also

$$\text{factorial } x = \text{if } (x=0) \text{ then } 1 \text{ else } x * ( \text{if } (x-1=0) \text{ then } 1 \text{ else } (x-1) * ( \text{if } (x-1-1=0) \text{ then } 1 \text{ else } (x-1-1) * \text{factorial}(x-1-1-1) ) )$$
and so on....

To compute the factorial, in a sense, we unwind its definition as many times as it is needed by the argument, that is until \( x-1-1-1\ldots -1 \) gets equal to 0. Such an unwinding is exactly what the \( \text{Y} \) combinator does. In fact

\[
\text{Y} \ F =_\beta F(\text{Y} \ F) =_\beta F(F(\text{Y} \ F)) =_\beta F(F(F(\text{Y} \ F))) =_\beta \ldots \text{ and so on}
\]

If you take as \( F \) the lambda-term \( \lambda f. \lambda x. \text{if} (\text{iszero} \ x) \downarrow (\text{mult} \ x \ (f \ (\text{sub} \ x \ 1))) \), by applying a beta-reduction you get

\[
\text{Y} \ F =_\beta \lambda x. \text{if} (\text{iszero} \ x) \downarrow (\text{mult} \ x \ ((\text{Y} \ F)(\text{sub} \ x \ 1)))
\]

but you can also get

\[
\text{Y} \ F =_\beta \lambda x. \text{if} (\text{iszero} \ x) \downarrow (\text{mult} \ x \ ((\text{if} (\text{iszero} \ (\text{sub} \ x \ 1))) \downarrow (\text{mult} \ (\text{sub} \ x \ 1) \ ((\text{Y} \ F)(\text{sub} \ (\text{sub} \ x \ 1) \ 1)))))
\]

and so on and so on...

That is \( (\text{Y} \ F) \) is beta-convertible to all the possible "unwindings" of the factorial definition. Then if you apply \( \downarrow \) to \( (\text{Y} \ F) \) you can get, by a few beta-reduction

\[
(\text{Y} \ F) \downarrow =_\beta \text{if} (\text{iszero} \ \downarrow) \downarrow (\text{mult} \ \downarrow ((\text{if} (\text{iszero} \ (\text{sub} \ \downarrow \ 1))) \downarrow (\text{mult} \ (\text{sub} \ \downarrow \ 1) \ (\text{if} (\text{iszero} \ (\text{sub} \ (\text{sub} \ x \ 1) \ 1)))) \downarrow (\text{mult} \ (\text{sub} \ (\text{sub} \ x \ 1) \ 1) \ ((\text{Y} \ F)(\text{sub} \ (\text{sub} \ (\text{sub} \ x \ 1) \ 1) \ 1))))
\]

We can easily check that the term on the right is beta convertible to \( \downarrow \), that is to the factorial of \( \downarrow \).

So, why does \( \text{Y} \) work? Because, in a sense, it is an "unwinder", an "unroller" of definitions.

**Using \( \text{Y} \) (and hence recursion) to define infinite objects.**

We have seen before how to represent a few simple data types in the lambda-calculus and we have said that it is easy to devise a representation for all the inductively defined data types. In particular, it is possible to represent the data type \( \text{List} \) with its basic element \text{emptylist} and the list constructor \text{cons} that we shall use below without giving their explicit definition.

Let us consider the following \( \lambda \)-term:

\[
\text{Y} \ (\lambda l. \text{cons} \ x \ l) \quad (\ast)
\]

It is definitely a fixed point. It is the fixed point of the term \( \lambda l. \text{cons} \ x \ l \). If we give a list \( ls \) as input to this term, it returns a new list formed by \( x \) followed by \( ls \). It is not difficult to check that the least fixed point of \( \lambda l. \text{cons} \ x \ l \) is the infinite list of \( x \)'s. This means that the term \( \text{Y} \ (\lambda l. \text{cons} \ x \ l) \) represents an infinite object!

So, in the lambda-calculus \( \text{Y} \) can be used not only to represent recursively defined
functions, but also recursively defined object. In fact we could define the infinite list of x's by means of the following equation:

\[ \text{Infinite} = \text{cons} \times \text{Infinite} \]

Of course we cannot completely evaluate \( Y (\lambda. \text{cons} \times I) \) (i.e. get all the information it contains), since the value of this term would be an \textit{ininitely} long term (and in fact it does not have a normal form). But nonetheless we can partially evaluate it in order to get \textit{finite} approximations of the infinite list of x's.
You see, after a few beta reductions and beta conversions we can obtain

\[ YF =_\beta (\text{cons} \times (\text{cons} \times (\text{cons} \times (YF)))) \quad \text{where } F \text{ is the term (*)} \]

and this term is a finite approximation of the infinite list of x's. Of course the term (*) is convertible to \textit{all} the possible finite approximations of the infinite list of x's. This means that it is possible to \textit{use} and \textit{work with} terms representing infinite object. In fact, it is easy to check that

\[ \text{car} (Y (\lambda. \text{cons} \times I)) =_\beta x \quad (***) \]

where \text{car} is the \( \lambda \)-representation of the function that, taken a list, returns its first element. So it is possible to use the term (**), even if it represents an infinite object (it is reasonable to be able to pick the first element from an infinite list.)

What we have seen here shows that in functional programming it is possible to deal also with infinite objects. Such a possibility, however, is not present in all functional languages, since when we evaluate an expression in a functional language we follow a precise reduction strategy, and not all the reduction strategies enable us to safely handle infinite objects.
In Scheme it is not possible since, in a term like (***) above, Scheme would keep on endlessly evaluating the argument of \text{car}. In Haskell, instead, the evaluation of a term like (***) terminates.
In order to better understand this fact, see the section below on evaluation strategies.

3.5 Reduction strategies

One of the aspects that characterizes a functional programming language is the reduction strategy it uses. Informally a reduction strategy is a "policy" which chooses a \( \beta \)-redex , if any, among all the possible ones present in the term. Abstractly, we can look at a strategy as a function.

\textbf{Definition}
A reduction strategy is a function

\[ S: \Lambda \rightarrow \Lambda \]

(where \( \Lambda \) is the set of all the \( \lambda \)-terms), such that:
if $S(M) = N$ then $M \rightarrow^\beta N$

Let us see, now, some classes of strategies.

**Call-by-value strategies**

These are all the strategies that never reduce a redex when the argument is not a value. Of course we need to precisely formalize what we intend for "value". In the following example we consider a value to be a lambda term in normal form.

*Example:*

Let us suppose to have:

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

A call-by-value strategy never reduce the term $(\lambda x. x)((\lambda y. y) \ z)$ because the argument is not a value. In a call-by-value strategy, then, we can reduce only one of the redex underlined.

Scheme uses a specific call-by-value strategy, and a particular notion of value. Informally, up to now we have considered as "values" the terms in normal form. For Scheme, instead, also a function is considered to be a value (an expression like $\lambda x. M$, in Scheme, is a value). Then, if we have

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

The reduction strategy of Scheme first reduces the biggest redex (the whole term), since the argument is considered to be a value (a function is a value for Scheme). After the reduction the interpreter stops, since the term $(\lambda z. (\lambda x. x) \ x)$ is a value for Scheme and hence cannot be further evaluated.

**Call-by-name strategies**

These are all the strategies that reduce a redex without checking whether its argument is a value or not. One of its subclasses is the set of lazy strategies. Generally, in programming languages a strategy is lazy if it is call-by-name and reduces a redex only if it's strictly necessary to get the final value (this strategy is called call-by-need too). One possible call-by-name strategy is the leftmost-outermost one: it takes, in the set of the possible redex, the leftmost and outermost one.

*Example:*

Let us denote by $I$ the identity function and let us consider the term:

$$(I \ (I \ I)) \ (I \ (I \ I))$$

the leftmost-outermost strategy first reduces the underlined redex.
We have seen before that the leftmost-outermost strategy is very important since, by the Standardization Theorem, if a term has a normal form, we get it by following such a strategy. The problem is that the number of reduction steps is sometimes greater than the strictly necessary ones.

If a language uses a call-by-value strategy, we have to be careful because if an expression has not normal form, the search could go on endlessly.

**A fixed point combinator for Scheme-like strategies**

Let us consider the functional $F$ that has the factorial function as fixed point. Then we have that

$$(\Theta F) \xrightarrow{\gamma} 6$$

Let us suppose to use a call-by-value strategy like that of Scheme. We get

$$(\Theta F) \xrightarrow{\gamma} (F (\Theta F)) \xrightarrow{\gamma} (F (F (\Theta F))) \xrightarrow{\gamma} \ldots$$

If we would like to use the above fixed point operator to define recursive function in Scheme (we do not do that when we program in Scheme, of course), we have this problem. We could overcome this problem, however, by defining a new fixed point combinator, $H$, that, if we wished, we could safely define and use in Scheme as well (do it as an exercise when you study Scheme).

Let us consider the following $\lambda$-term:

$$H \equiv \lambda g.((\lambda f. f f) (\lambda f. (\lambda x. f f x)))$$

It's easy to prove that this term is a fixed point operator. In fact we have that:

$$(H F) \rightarrow F (\lambda x. A A x)$$

where $A$ is $(\lambda f. F (\lambda x. f f x))$ and $(H F) \rightarrow A A$. From the point of view of the computations, we can write it in this form:

$$(H F) \rightarrow F (\lambda x. H F x)$$

Functionally, $F (\lambda x. H F x)$ has the same behaviour of $H F$, but Scheme does not reduce it because that is a value for Scheme. Then if, for example, $F$ is the functional whose fixed-point represents the function factorial, if we wish to compute the factorial of 5 we can evaluate the term $((H F) 5)$. That is we have to evaluate

$$(F (\lambda x. H F x)) 5$$

and this evaluation is possible without getting into any infinite reduction sequence. If you compute the factorial of 5 in Scheme in this way, you will have to wait a bit of time in order to get the result...
3.6 Head Normal Forms

In the previous sections, we have implicitly assumed that in \( \lambda \)-calculus, if a term has not a normal form, then it represents no information at all. For example:

\[ \Omega \text{ has not normal form and it represents the indefinite value} \]

But this is not always true, that is it is possible to have terms that have no normal form but nonetheless represent ("contain") some information.

We have already seen an example of this fact: the \( \lambda \)-term

\[ Y(\lambda . \text{cons } x \ l) \quad (*) \]

which it has no normal form, but it represents a precise object that contains an infinite amount of information. We cannot get such information all together (it would be an infinitely long term), but we have seen before that we can get all possible finite approximations of the information it contains.

So, it is definitely possible to use and work with terms that have no normal form but nonetheless possess a "meaning".

Note:

As discussed in the section about infinite objects, we have to be careful about the strategy we intend to use when manipulating expressions without normal form. If we tried to evaluate an expression like \( \text{car}(Y(\lambda . \text{cons } x \ l)) \) and we used a call-by-value strategy with terms without normal form as values, we would get stuck in a loop, since we would try to get a value out of \( Y(\lambda . \text{cons } x \ l) \). In a language like Haskell, instead, we would obtain \( x \) (When you begin to have a look at Haskell, read also the notes [HASK] on laziness and infinite objects, available in the reading material).

How is it possible to formalize in the lambda-calculus the notion we have informally discussed above? That is, the notion of a term containing some information even if it has no normal form.

We do that by means of the definition of head normal form, the formalization of the notion of finite approximation of the meaning of a term.

Definition
A term is in head normal form (hnf) if, and only if, it has a form as follows:

\[ \lambda x_1...x_k. y M_1...M_n \quad (n,k \geq 0) \]

where \( y \) can be any variable, also one of the \( x_i \)'s.

If a term in head normal form \( \lambda x_1...x_k. y M_1...M_n \) had a normal form, the normal form would have the following shape:

\[ \lambda x_1...x_k. y N_1...N_n \]

where \( N_1,..,N_n \) would be the normal forms of \( M_1,..,M_n \).
If the term, instead, had no normal form, it is easy to check that in any case the portion of
the term
\[ \lambda x_1...x_k y \]
will stay always the same after any possible sequence of reduction steps.
(Nota that a term in normal form is also in head normal form).

Example:

Let us consider the term:
\[ \lambda xyz.z(\lambda x.x)(\lambda z.t(zz))((\lambda t.t(\lambda (\lambda x)) \lambda t.t(\lambda (\lambda x)) )) \]

it is in normal form, but it is in head normal form too.

Let us consider, now, \( \Omega \). It is not in head normal form and it has no head normal form.

Now, going back to the notion of term containing no information (term representing the
indeterminate), we have just seen that some information can be obtained out of terms having
no normal form. So when can we say that a term really contains no information at all? Well,
now we can answer: when a term has no head normal form!

Are we sure this to be the right answer? Well, the following fact confirms that it is really the
right answer:
Let us take two terms not having head normal form, for example:

\[ \Omega \quad \text{and} \quad \Omega x \]

Using the Church-Rosser theorem we can prove they are not \( \beta \)-convertible. Now If we add
to the \( \beta \)-conversion theory the axiom:

\[ \Omega = \beta \quad \Omega x \]

the theory of beta-conversion does remain consistent. It is no surprise, since this axiom is
stating that a term that contains no information is equivalent to a term that contains no
information, no contradiction in that!

Then, we can rightly represent the indeterminate with any term that has not head normal
form.

In fact we could consistently add to the theory of the beta-conversion the axiom stating that
all the terms with no head normal form are \( \beta \)-convertible (this intuitively means that we
could consistently make "collapse" all the possible representations of the indeterminate into
one single element).

4. Types

4.1 Introduction to types

The functional programming languages can be typed or untyped. A typed one is, for
example, Haskell, while an untyped language is Scheme. By introducing the notion of type in the \(\lambda\)-calculus we shall be able to study it in an abstract and simplified way.

We can look at types as predicates on programs or, better, as \textbf{partial specifications} about the behaviour of programs. In the programming practice the use of types is helpful to prevent a number of programming errors.

There are two different approaches to the use of types.

- \textbf{the approach a' la Curry} (the one used in Haskell)
- \textbf{the approach a' la Church} (the one used in Pascal)

In the Curry approach, we first write a program and then check whether it satisfies a given specification. Instead, in the Church approach, types are included in the syntax of the language and, in a sense, represent some constraints we have to follow during the writing of our programs in order to avoid making particular errors.

In the approach a' la Curry, checking if a term \(M\) satisfies the specification represented by the type \(t\), it is usually referred to as "assigning the type \(t\) to the term \(M\)".

\[ 4.2 \textbf{Type assignment 'a la Curry} \]

In this approach we first write a program and then we check whether it satisfies the specification we had in mind. This means that we have two distinct sets: \textit{programs} and \textit{partial specifications}.

If we wish to talk in a general and abstract way about types (a' la Curry) in functional programming languages we can proceed with the following formalization:

\[
\begin{align*}
\text{Programs} &: \text{the set of } \lambda\text{-terms}, \text{i.e.} \\
\Lambda & \\
\text{Partial specifications} &: \text{the set of types}, \text{i.e.} \\
T & := \varphi \mid T \rightarrow T
\end{align*}
\]

where \(\varphi\) is a meta-variable which ranges over a set of type variables \((\varphi, \varphi', \varphi'', a, b, \ldots \text{ etc.})\).

In this formalization we have considered the simplest form of type (in fact these types are called \textit{simple types}). Elements of \(T\) will be denoted by \(\sigma, \tau, \rho\), etc.

There arise now a whole bunch of questions we should answer:

1) How can we \textit{formally} show that, given a program (a term) \(M\) and a specification (a type) \(\sigma\), \(M\) satisfies the specification \(\sigma\)? That is, how can we prove that \(M\) has type \(\sigma\)?

2) Is it decidable, given a \(M\) and a \(\sigma\), that \(M\) is of type \(\sigma\)? i.e., given a program (term) \(M\) and a specification (type) \(\sigma\), is there any algorithm enabling us to check whether \(M\) satisfies \(\sigma\)?

3) Do types play any role during a computation?

4) What is the intrinsic meaning of having type variables in our system? has the use of type variables some implication for our system?
5) Is it possible to "compare" types? i.e., is it possible to formalize the notion that a type represents a "more general" specification than another one?

6) Is there any relation among all the possible types we can assign to a term M? there exists the most general type among them? If so, can it be algorithmically found?

Let us start with **question 1**. What we need to do is to devise a formal system whose judgments (typing judgments) are of the form

\[ M : \sigma \]

and whose meaning is: the program \( M \) respects the specification \( \sigma \) (formally, the term \( M \) has type \( \sigma \)).

If we think about it, however, we can notice that our typing judgments should provide some more information than simply \( M : \sigma \). In fact, let us consider the term \( \lambda x.yx \). Which sort of specification is satisfied by such a program? In case the input is of type \( \varphi \), a type for \( \lambda x.yx \) is definitely of the form

\[ \varphi \rightarrow \text{something} \]

But now, how can we found out what **something** could be, if we do not have any information about the free variable \( y \)? Our program (term) takes an input \( x \) and returns the result of the application of \( y \) to \( x \). If we do not make any assumption about the type of the free variable \( y \), we cannot say anything about the type of \( yx \). This means that the (typing) judgments we are looking for should indeed have the following shape:

\[ B \vdash M : \sigma \]

where \( B \) (called a **basis**), provides informations about the types of the free variables of \( M \). The meaning of the typing judgment \( B \vdash M : \sigma \) is then: using the informations given by the basis \( B \), the term \( M \) satisfies the specification \( \sigma \) (i.e. \( M \) has type \( \sigma \) under the assumptions provided by the basis \( B \); or, shortly, \( M \) has type \( \sigma \) in \( B \)).

Formally \( B \) will be a set of pairs of the form: \( \text{variable} : \text{type} \)

An example of basis is, for instance, \( B = \{ x : \tau, z : \rho \} \)

Let us introduce now the formal system enabling us to infer types for lambda-terms.

**Definition (Curry type assignment system)**

**Axiom**

\[ B \vdash x : \sigma \quad \text{if } x : \sigma \text{ is in } B \]
(a variable \( x \) has type \( \sigma \) if the statement \( x : \sigma \) is in \( B \).

**Inference rules**

These rules allow to derive all the possible valid typing judgements.

- \[
    \frac{B \vdash M : \sigma \rightarrow \tau \quad B \vdash N : \sigma}{B \vdash MN : \tau}
\]

This rule says that if, by using the assumptions in \( B \), we can prove that \( M \) has type \( \sigma \rightarrow \tau \) and \( N \) has type \( \sigma \), then we are assured that, in the basis \( B \), the application \( MN \) has type \( \tau \). This rule is usually called \((\rightarrow E)\): "arrow elimination". This rule can be "read" also bottom-up, as follows: if we wish to prove that \( MN \) has type \( \tau \) in \( B \), then we need to prove that \( M \) has type \( \sigma \rightarrow \tau \) in \( B \) for some \( \sigma \) and that \( N \) has type \( \sigma \) in \( B \).

- \[
    \frac{B, x : \sigma \vdash M : \tau}{B \vdash \lambda x. M : \sigma \rightarrow \tau}
\]

This rule says that if, by using the assumptions in \( B \) and the further assumption "\( x \) has type \( \sigma \)"", we can prove that \( M \) has type \( \tau \), then we we are assured that, in \( B \), \( \lambda x. M \) has type \( \sigma \rightarrow \tau \). This rule is called \((\rightarrow I)\): "arrow introduction". Also this rule can be "read" bottom-up. Do it as exercise.

A term \( M \) is called **typeable** if there exist a basis \( B \) and a type \( \sigma \) such that the judgment \( B \vdash M : \sigma \) can be derived in the assignment system above.

The above formal system is the core of the type systems of languages like Haskell, ML and others. This is the main motivation for which we are studying it.

**Remark:**

*In Haskell, before writing the definition of a function, we can declare what type our function is intended to have, for instance:*

\[
    \text{fact} :: \text{Int} \rightarrow \text{Int}
\]

*By doing that, after defining the function fact the Haskell interpreter can check whether our program fact satisfies the given specification. In particular, the interpreter tries to derive, in a formal system like the one defined before, the Haskell typing judgement \( \text{fact} :: \text{Int} \rightarrow \text{Int} \).*

*Note: In Haskell we have not the concept of basis, because in correct programs an
identifier either is a bound variable or it has been previously associated to some value.

Example:

Let $\emptyset$ be the empty set and let us consider the judgment

$$\emptyset \vdash \lambda y.x : \sigma \to (\tau \to \sigma)$$

whose informal meaning is: by considering no assumption at all (the empty basis), the term $\lambda y.x$ satisfies the specification $\sigma \to (\tau \to \sigma)$. If we wish to formally prove that the above judgment is valid, we need to use the Curry type assignment system. Let us start from the axiom:

$$\{x : \sigma, y : \tau\} \vdash x : \sigma$$

By using $(\to I)$ two times, we get

$$\{x : \sigma, y : \tau\} \vdash x : \sigma$$

$$\{x : \sigma\} \vdash \lambda y.x : \tau \to \sigma$$

$$\emptyset \vdash \lambda x.\lambda y.x : \sigma \to (\tau \to \sigma)$$

Let us notice that in the last two steps we have discharged, from the basis we have started with, the assumptions for the bound variables of the term. In fact, a basis has to provide only informations about free variables, since the informations about bound variables are already contained in the type of the term.

Example:

We want to prove that:

$$B = \{f : \sigma \to \sigma, x : \sigma\} \vdash f(f(x)) : \sigma$$

Starting from $B$, we use the axiom and the rules of our formal system:

$$\frac{}{B \vdash f : \sigma \to \sigma}$$

$$\frac{}{B \vdash x : \sigma}$$

$$\frac{}{B \vdash (fx) : \sigma} \quad B \vdash f : \sigma \to \sigma$$

$$\frac{}{B \vdash f(x) : \sigma}$$

$$\frac{}{B \vdash f : \sigma \to \sigma}$$

$$\frac{}{B \vdash f(fx) : \sigma}$$
We can go on and derive, from the derivation above, other judgments:

\[
\begin{align*}
\{f : \sigma \to \sigma, x : \sigma\} & \vdash f (fx) : \sigma \\
\{f : \sigma \to \sigma\} & \vdash \lambda x. f (fx) : \sigma \to \sigma \\
\emptyset & \vdash \lambda f. \lambda x. f (fx) : (\sigma \to \sigma) \to (\sigma \to \sigma)
\end{align*}
\]

Remark:
There are programs that do not satisfy any specification, whatever be the basis \(B\). For example, let us consider:

\[
B \vdash xx : \tau
\]

It is impossible to derive this judgment in our system, whatever basis \(B\) and type \(\tau\) we choose. What is the implication of this simple observation? Well, typed languages restrict the set of programs we can write, but on the other hand the programs we can write are safer, i.e., they are guaranteed not to contain certain errors and to satisfy their partial specifications. We loose something of course, that is flexibility, the possibility of writing "tricky" and "sly" programs.
So, in typed languages: less flexibility = more safeness.

There is a quite relevant property of typeable terms that concerns strong normalizability.

**Theorem**
Any typeable term is strongly normalizable.

So, since self-application cannot be typed, it follows that the combinator \(Y\) cannot be typed. This implies that we cannot type any term representing a recursive algorithm. This is also why, whereas in Scheme, if we wished, we could define recursive programs using fixed-point combinators, in Haskell it is not possible.
Of course this is not a real drawback for Haskell, since it is not realistic to write actual recursive programs by means of fixed-point combinators. In Haskell recursion is obtained by the use of recursive equations (which corresponds to the possibility of giving names to expressions.)

Let us now discuss another relevant property of our type assignment system: **Subject reduction**

Saying that \(M\) has type \(\sigma\) \((M : \sigma)\) amounts to say that \(M\) represents a value of type \(\sigma\). The types do not give information about the form of the terms (about their syntax), but, instead, they provide informations about their intrinsic meaning, their semantics.
Hence, if a term has a certain type, it is important to be sure that during its evaluation the term "maintains" the same type.
For example, if a term \(M\) represents a program that returns a number, types would be useless
if we could reduce \(M\) to a term that represents a program which returns a list.
Fortunately we can prove the following
**Theorem** (Subject reduction)
If in a basis $B$ we can prove that $M$ has type $\sigma$, that is

$$B \vdash M : \sigma$$

and if $M \rightarrow N$, then we can also prove that

$$B \vdash N : \sigma$$

Thanks to this theorem we are sure that, if we check that a term $M$ has a given type at the beginning of a computation, then we can forget about types during the computation, since all the terms that we obtain by reducing $M$ can be given the very same type. This also implies that types are irrelevant, do not play any role, in the computational process.

Let us note that the subject reduction property holds only for reductions and not for expansions. That is, if we have that

$$B \vdash N : \sigma \quad \text{and} \quad M \rightarrow N$$

it is not always true that

$$B \vdash M : \sigma$$

By means of the Subject reduction theorem we have also answered question 3).

Let us now try and answer question 4). Question 2) has still to be answered, but it will be done using the answer for 6)

**Definition** (Type substitution)
Let $T$ be the set of simple types:

a) the type substitution $(\phi \mapsto \rho) : T \rightarrow T$, where $\phi$ is a type-variable and $\rho$ is a type, is inductively defined by:

$$
\begin{align*}
(\phi \mapsto \rho) \phi & = \rho \\
(\phi \mapsto \rho) \phi' & = \phi', \text{ if } \phi' \neq \phi \\
(\phi \mapsto \rho) \sigma \rightarrow \tau & = ((\phi \mapsto \rho) \sigma) \rightarrow ((\phi \mapsto \rho) \tau)
\end{align*}
$$

The above definition formalizes the notion of "substituting a type variable by a type inside a type"

b) We can compose type substitution, that is, if $S_1$, $S_2$ are type substitutions, then so is $S_1 o S_2$, where:

$$S_1 o S_2 \sigma = S_1 (S_2 \sigma)$$

c) A type substitution can also be applied to a basis: $SB = \{x : S\tau \mid x : \tau \text{ is in } B\}$
d) A type substitution can also be applied to pairs of basis and type (it will be useful later on)

$$S <B, \sigma> = <SB, S\sigma>$$

**Note:** The notion of substitution for types is much simpler than the one for terms, since in simple types we do not have bound type-variables.

**Example:** if we apply the type substitution $$(\varphi \mapsto \sigma)$$ to the type $$\varphi \rightarrow (\tau \rightarrow \varphi)$$, and if $$\tau$$ does not contain $$\varphi$$, we obtain the type $$\sigma \rightarrow (\tau \rightarrow \sigma)$$

If for $$\sigma, \tau$$ there is a substitution $$S$$ such that $$S\sigma = \tau$$, we say that $$\tau$$ is a (substitution) **instance** of $$\sigma$$.

The following lemma shows that if a term has a type, it has indeed infinitely many types. The Lemma will provide, at the same time, an answer to **question 4**.

**Lemma (substitution lemma)**

If using a basis $$B$$ we can prove that a term $$M$$ has a type $$\sigma$$, that is

$$B \vdash M : \sigma$$

then, for all possible substitutions $$S$$, we can also prove that:

$$S(B) \vdash M : S(\sigma)$$

So the Substitution Lemma states that if a term satisfies a specification, then it satisfies infinitely many ones (because we can have infinitely many possible substitutions.)

**Example:**

*It is easy to prove that*

$$\{x : \varphi \rightarrow \psi\} \vdash \lambda y .xy : \varphi \rightarrow \psi$$

*The Substitution Lemma states that if we apply any substitution $$S$$, it is also true that*

$$S(\{x : \varphi \rightarrow \psi\}) \vdash \lambda y .xy : S(\varphi \rightarrow \psi)$$

*For example, by considering the substitution*

$$(\varphi \mapsto (\varphi' \rightarrow \varphi)) \circ (\psi \mapsto \xi)$$

*it is true that*

$$x : (\varphi' \rightarrow \varphi) \rightarrow \xi \vdash (\lambda y .xy) : (\varphi' \rightarrow \varphi) \rightarrow \xi$$
Example:
If we have derived
\[ \emptyset \vdash \lambda x. x : \varphi \rightarrow \varphi \]

by the Substitution Lemma we are sure that also the following judgment is derivable
\[ \emptyset \vdash \lambda x. (\psi \rightarrow \varphi) \rightarrow (\psi \rightarrow \varphi) \]

What we have just seen means that in our formal system we have polymorphic functions (a function is called *polymorphic* if it can be correctly applied to objects of different types). So, by using type variables, we have implicitly introduced a notion of polymorphism in our type assignment system.

Since this system is the core of the type system of Haskell, this implies that Haskell is a language with polymorphism. (In Haskell type variables have names like `a`, `b`, `c` etc.)

To state that the lambda term \( \lambda x. x \) has type \( a \rightarrow a \) is tantamount to say that \( \lambda x. x \) is a polymorphic function, that is it can be uniformly applied to any element of any type \( T \) and it returns an element of type \( T \). The type \( a \rightarrow a \) is (implicitly) a polymorphic type since, being "\( a \)" a type-variable, it "represents" all the types with shape \( T \rightarrow T \).

The polymorphism of our system can be said to be *implicit*, since the fact that a term has many types it is not *explicitly* expressed in the syntax by means of particular operators on types, but it is expressed by results like the Substitution Lemma.

A possible syntax for explicit polymorphism could be the following one:

\[ \lambda x. x : \text{for all } \varphi, \varphi \rightarrow \varphi \]

In this case the polymorphic behaviour of the term is explicitly described by the syntax.

We can now provide, using the notion of type substitution, an answer to **question 5** by giving the following definition.

**Definition** (more general type)
Let \( \tau \) and \( \sigma \) be two simple types.
\( \tau \) is said to be **more general** than \( \sigma \) if there exists a substitution \( S \) such that \( \sigma = S(\tau) \).
That is if the latter can be obtained from the former by means of a type substitution.

We know that if a term \( M \) has a type, it has infinitely many ones. Now, it is also possible to prove that there exists a precise relation among all the types that a given term has (i.e. all the possible partial specifications of a program are, so to speak, connected to each other).
That would be an answer for **question 6**. Such an answer will also provide an answer for **question 2** that up to now we have left unanswered.

### 4.3 Principal pairs

http://www.dmi.unict.it/~barba/LinguaggiIII.html/READING_MATERIAL/LAMBDA CALCULUS/LAMBDA CALCULUS.1.HTM
**Theorem**

Given a term M, if M is typeable then it has a **Principal Pair** \(<P, \pi>\), where P is a basis and \(\pi\) a type such that:

1) we can derive the following typing for M

\[ P \vdash M : \pi \]

2) Moreover, if we can derive

\[ B \vdash M : \sigma \]
then there exists a substitution S such that \(B = S(P)\) (as a matter of fact B could be a superset of \(S(P)\)) and \(\sigma = S(\pi)\).

In other words, the principal pair represents the most generic typing possible for M. That is, any other basis and type that we can use to type M can be obtained from the principal pair.

This partially answer our **question 6**)

**Example:**

*Let us consider the term:*

\[ \lambda x. x \]

*It has principal pair:*

\[ <\emptyset, \varphi \rightarrow \varphi> \]

*This means that any other typing for the identity is a substitution instance of this principal pair.*

The proof of the principal pair algorithm can be given by providing an algorithm that, given a term M, returns its principal pair, \(\text{pp}(M)\), and fails in case M be not typeable.

These completes the answer of our **question 6**).

Before giving the formal description of the Principal Pair Algorithm, let us first informally describe how it works.

The algorithm tries to type the given term in the most general way (that is to find the most general basis and the most general type for the term.) It tries to do that by using the rules of the assignment system, in a bottom-up way.

- If we consider a **variable** \(x\), well, the only way to type it is by using an axiom, and the most general axiom for \(x\) is definitely the one that has only \(x\) in the basis and a type-variable as type for it.

- If we consider an **abstraction** \(\lambda x. M\) we know that the only way to type it is by using rule \((\rightarrow 1)\). It is then clear that if we wish to type \(\lambda x. M\) in the most general way by means of this rule, we first need to find the most general typing for \(M\) (by using recursively the algorithm itself. If we fail the whole algorithm fails). Once we have the most general typing for \(M\), if \(x\) is in the basis (with a certain type, say \(\sigma\)) then the most general type for \(\lambda x. M\) will be \(\sigma \rightarrow \pi\) (where \(\pi\) is the most general type found for \(M\)). Otherwise, if \(x\) is not in the
basis (that is, x is not a free variable of M) the most general typing for \( \lambda x. M \) is \( \phi \rightarrow \pi \), where \( \pi \) is the most general type found for M and \( \phi \) is a type variable not yet used (if x is not a free variable of M it means that the input of \( \lambda x. M \) could be anything and hence a type-variable \( \phi \) is the most general type for such a generic input.

- If we consider an application \( MN \) we know that the only way to type it is by using rule \( (\rightarrow E) \). It is then clear that if we wish to type MN in the most general way by means of this rule, we first need to find the most general typing for M and the most general typing for N (by using recursively the algorithm itself. If we fail the whole algorithm fails). Once we have the most general basis and type for M, say \( P_1 \) and \( \pi_1 \), and the most general basis and type for N, say \( P_2 \) and \( \pi_2 \), these bases and types, however, cannot be used to type MN with the rule \( (\rightarrow E) \) unless \( \pi_1 \) is an arrow type having \( \pi_2 \) as domain. If this is not the case, we can profit from the Substitution Lemma and find the simplest substitution that, when applied to \( \pi_1 \) and to \( (\pi_2 \rightarrow \phi) \), makes them equal (\( \phi \) is a new type variable). Once we have such a substitution, say \( S_1 \) (we find it by means of another algorithm called unify. If it fails the whole algorithm fails), we apply it to \( P_1 \), \( \pi_1 \), \( P_2 \) and \( \pi_2 \). Unfortunately, we cannot use rule \( (\rightarrow E) \) yet, since the bases used in the two premises of the rule must be the same, and in general it is not true that \( S_1 P_1 \) is the same as \( S_1 P_2 \). This means that we need to find another substitution \( S_2 \) (the simplest one), by means of which we manage to make \( S_1 P_1 \) and \( S_1 P_2 \) equal (in their common variables). If we manage to find such a substitution (otherwise the whole algorithm fails) we are done, since now we can use rule \( (\rightarrow E) \) to type MN with \( S_2 S_1 \phi \) from the basis \( S_2 S_1 (P_1 \cup P_2) \) (we consider the union because the free variables of M could be different from the free variables of N).

The formalization of the process we have informally presented before is therefore the following recursive algorithm.

**Definition (The principal pair algorithm)**

Given a term M we can compute its principal pair pp(M), if any, as follows:

a) For all x, \( \phi \) : pp(x) = \(<\{x : \phi\},\phi>\)

b) If pp(M) = \(<P, \pi>\) then:
   
   i) in case x is in FV(M) (and hence there is a \( \sigma \) such that x : \( \sigma \) is in P) we define
   
   \[ pp(\lambda x. M) = <P x, \sigma \rightarrow \pi> \]

   ii) otherwise, in case x is not a free variable, we define
   
   \[ pp(\lambda x. M) = <P, \phi \rightarrow \pi>, \] where \( \phi \) does not occur in \(<P, \pi>\).

   c) If pp(M_1) = \(<P_1, \pi_1>\) and pp(M_2) = \(<P_2, \pi_2>\) (we choose, if necessary, trivial variants such that the \(<P_1, \pi_1>\) and \(<P_2, \pi_2>\) contain distinct type variables), \( \phi \) is a type-variable that does not occur in any of the pairs \(<P_i, \pi_i>\), and:

   \[ S_1 = unify \pi_1 (\pi_2 \rightarrow \phi) \]

   \[ S_2 = UnifyBases(S_1 P_1) (S_1 P_2) \]

   then we define \( pp(M_1 M_2) = S_2 o S_1 <P_1 \cup P_2, \phi> \). (\( unify \) and \( UnifyBases \) are defined below)
Of course, we should also prove that this algorithm is correct and complete, but this would go beyond our scopes.

The principal pair algorithm provides also an answer to question 2. In fact, if a term is not typeable the algorithm fails, whereas it returns its principal pair if it is typeable. So, if a term is typeable we can not only effectively find it out, but, in a sense, we can have all its possible types, since any possible typing can be got out of the principal pair.

Extensions of the above algorithm are used in languages like Haskell and ML. Let us roughly see how it is used: let us define a function, say

\[ \text{Id } x = x \]

And let us write now the following thing (that amounts to asking to the Haskell interpreter the following question: "what is the principal type of Id?")

:type Id

What the interpreter does is to run an algorithm similar to the one we have defined above, producing the principal type of the term associated to the identifier Id (the principal type and not the principal pair, since our Haskell expressions must be closed).

In case of Id the interpreter returns

\[ a \rightarrow a \]

Another possibility in Haskell is instead to "declare" the type of an expression before we write it, for instance:

\[ \text{Id : (Int } \rightarrow b \text{) } \rightarrow (\text{Int } \rightarrow b) \]

After the type declaration we can write the "code" for Id, that is

\[ \text{Id } x = x \]

Now, what the interpreter does is the following: it finds the principal type of the expression associated to Id and then checks whether the specification we gave, \((\text{Int } \rightarrow b) \rightarrow (\text{Int } \rightarrow b)\), can be obtained out of the principal type of Id by means of a type substitution.

You can notice that the pp algorithm uses the unify and UnifyBases sub-algorithms (described below). The unify algorithms takes two types and checks whether there exists a substitution that makes them equal. That is why sometimes the Haskell interpreter gives you a warning saying that it does not manage to unify some type variable, say a. This means that the unify algorithm implemented in the interpreter does not manage to find a substitution enabling the unify algorithm to successfully terminate.

**Definition (Unify and UnifyBases)**

Let \( B \) the collection of all bases, \( S \) be the set of all substitutions, and \( \text{Id}_a \) be the substitution that replaces all type-variables by themselves.

i) *Robinson's unification algorithm*. Unification of Curry types is defined by:
\[
\text{unify} :: T \times T \to S \\
\text{unify} \varphi \tau = (\varphi \leftrightarrow \tau), \text{ if } \varphi \text{ does not occur in } \tau \\
\text{unify} \sigma \varphi = \text{unify} \varphi \sigma \\
\text{unify} (\sigma \to \tau) (\rho \to \mu) = S_2 o S_1, \\
\]

where \( S_1 = \text{unify} \sigma \rho \) and \( S_2 = \text{unify} (S_1 \tau) (S_1 \mu) \)

ii) By defining the operation \textit{UnifyBases}, the operation \textit{unify} can be generalized to bases:

\[
\text{UnifyBases} :: B \times B \rightarrow S \\
\text{UnifyBases} B_0, x: \sigma \quad B_1, x: \tau = S_2 o S_1 \\
\text{UnifyBases} B_0, x: \sigma \quad B_1 = \text{UnifyBases} B_0 B_1, \text{ if } x \text{ does not occur in } B_1. \\
\text{UnifyBases} \emptyset B_1 = \text{Id}_S. \\
\]

where \( S_1 = \text{unify} \sigma \tau \) and \( S_2 = \text{UnifyBases} (S_1 B_0) (S_1 B_1) \)

Notice that \textit{unify} can fail only in the second alternative, when \( \varphi \) occurs in \( \mu \).

### 4.4 The typed lambda calculus a' la Church.

We have seen that in the Curry approach to types there is a neat separation between terms and types. The underlying "programming philosophy" being that we first write untyped programs and after that we use the type inference system either to check (possibly under certain assumptions - the basis-) if the program is typeable with a given type or to find its principal type (by means of the the \textit{pp} algorithm).

In the Church approach, instead, the types get involved during the construction of the term (program) itself. In a sense, types are part of the program and are used as "constraints" preventing the programmer to make certain mistakes. A fundamental property of the Church approach is that each term has just one single type. This is achieved by imposing every variable to have (to be declared to have) just one type. The constraint of declaring the type of each variable is adopted in several programming languages, like C, C++, Java, Pascal, lambda Prolog,

Since in the typed lambda calculus a' la Church each term has just one type, the set of terms is partitioned in subsets, one for each type.

The set of types is defined as follows.

\[
T :: \text{INT} | \text{BOOL} | T \to T \\
\]

Of course we might choose to have even more base types, not just INT and BOOL. Notice that, unlike in the Curry system, we have no type variable. In the Curry system, if a term \( M \) has a type containing type variables, say \( a \to a \), then \( M \) has infinitely many types
(by the Substitution Lemma), whereas in the Church system a term has just one type. This is also the motivation why, if we wish to introduce the notion of polymorphism in a Church system, it has necessarily to be explicit polymorphism.

Let us introduce now the rules of our system. Unlike in the Curry system, they are not type inference rules, but rather term formation rules, in which types sort of "guide" the correct formation of terms.

For any possible type we assume there exists an infinite set of variables having that particular type. We assume also that there exist no two variables with the same name but with different types.

The type of a variable will be denoted by means of superscripts, like in $x^t$ (the variable with name $x$ and type $t$.)
We shall denote by $\Lambda_t$ the set of well-formed typed lambda terms having type $t$.

**Definition** (Well-formed typed lambda terms)
The (term formation) rules of the typed lambda calculus a' la Church defining the sets $\Lambda_t$, are the following ones:

*For each typed variable $x^t$, $x^t$ belongs to $\Lambda_t$;*

*If $M$ belongs to $\Lambda_{t1} \rightarrow_t \Lambda_{t2}$ and $N$ to $\Lambda_{t1}$, then $(MN)$ belongs to $\Lambda_{t2}$;*

*If $M$ belongs to $\Lambda_{t1}$, then $\lambda x^{t1}.M$ belongs to $\Lambda_{t1} \rightarrow_t \Lambda_{t2}$;*

Notice that in the last clause, in case $x$ is a free variable of $M$, it has necessarily the type $t_1$ of the abstracted variable, since variables with the same name do have the same type.

**Property** (Uniqueness of type)
Let $M$ belongs to $\Lambda_{t1}$ and to $\Lambda_{t2}$, then $t_1=t_2$.

It is possible to develop also for the typed lambda calculus a' la Church a theory of beta-reduction and a theory of beta-conversion, starting, respectively, from the following axioms

$$(\lambda x^{t1}.M)N \rightarrow_\beta M[N/x^{t1}]$$

$$(\lambda x^{t1}.M)N =_\beta M[N/x^{t1}]$$

and proceeding in a similar way as in the untyped lambda-calculus.

In the above axioms, $(\lambda x^{t1}.M)N$ has obviously to be a well-formed term. It can be proved that $M[N/x^{t1}]$ is well formed if $(\lambda x^{t1}.M)N$ is so and that they have both the same type (belong to the same $\Lambda_t$).

In the Church typed lambda-calculus, the following questions naturally arise:
- given a term, is it well-formed? that is, has it been formed correctly using the rules of the system?
- given a well-formed term, what is its type?

The answers to these questions are provided by the **Type Checking algorithm** (TC). Such algorithm, given a term, checks whether it has been correctly formed and, in case, returns its type.

**Definition** (Type Checking algorithm)

\[
TC(x^t) = \text{t if } x^t \text{ belongs to } \Lambda_t; \text{ fails otherwise}
\]

\[
TC(MN) = \text{t2 if } TC(M) = t1 \rightarrow t2 \text{ and } TC(N) = t1; \text{ fails otherwise}
\]

\[
TC(\lambda x^{t1}.M) = t1 \rightarrow t2, \text{ is } TC(M) = t2 \text{ and } x^{t1} \text{ belongs to } \Lambda_{t1}; \text{ fails otherwise.}
\]

We say that a term M is a **type checks** if the TC algorithm does not fail on M.

It is easy to see that \(TC(M) = \text{t if } M \text{ belongs to } \Lambda_t\)

As can be expected, it is possible to prove the "Church version" of the subject reduction property.

**Theorem** (Type preservation under reduction)

If M belongs to \(\Lambda_t\) and \(M \rightarrow N\), then N belongs to \(\Lambda_t\)

Similarly to the Curry system, the following relevant theorem holds:

**Theorem** (Strong Normalization)

Let M be a well-formed typed term in the Church system, then M is strongly normalizable.

### 4.5 Relation between the systems of Curry and Church

As can be expected, there exists a strong relation between the two type systems described in the previous sections.

Given a simply-typed lambda term M, let \(|M|\) denote the untyped lambda term obtained from M by removing the type decorations. Then we have that if M type checks, then \(|M|\) is Curry-typeable. Viceversa, if a lambda term N is Curry-typeable, then we can find a suitable type for the bound variables of N, so to obtain a term in the Church system. More precisely, if we add base types to the types of the Curry system, the following holds:

**Proposition**

(i) If M is a well-formed term in the Church system and \(TC(M) = A\), then \(|M|\) can be given the type A in the Curry system (from a suitable basis). Note that the viceversa does not hold.
Also note that A is not necessarily the principal type of |M|.

(ii) If N can be given a type A in Curry, then there exists a well-formed Church typed term M such that $TC(M) = A$ and $|M| = N$.

4.6 **Lambda-calculus extensions and their type systems a' la Curry.**

We have said that the type system a' la Curry we have introduced in the previous sections is essentially the one used by the language Haskell. However, if we wish to investigate in an abstract way functional languages like Haskell, we cannot use just the pure lambda-calculus terms to represent programs.

In fact, we know that lambda-terms typeable in the type system a' la Curry are strongly normalizable, whereas not all typeable Haskell terms are so.

We cannot type fixed point operators like $Y$ and this prevents using the notion of recursion in the simply typed lambda-calculus a' la Curry.

In Haskell, instead, we have recursion (since we have the possibility of giving names to terms).

Hence, if we wish to study the properties of languages like Haskell in a still general and abstract framework, a possibility is to extend the lambda-calculus in order to make is a bit closer to a real language, for what concerns both terms and types.

Let us call $LC^+$ our extended calculus.

We assume $LC^+$ to contain constants. For instance constants for numbers (once we know that we can represent numbers in the pure lambda-calculus, we can use constants instead, for sake of readability and computational efficiency): $0, 1, 2$, ecc., constants for booleans: $true, false$, constants for numeric functions like $add, mul$, ecc. and also a constant for the conditional expression: $if$. If we wish we can introduce a constant for the pair constructor: $\langle$, and many more, if we wish. (For simplicity's sake we shall denote by $[M,N]$ the term $[M, N]$. If we have the pair constructor we also need the two selectors on pairs: $first$ and $second$.

Notice the difference between having a constant for a certain value, say $true$, and having that value represented by a term, $\lambda xy. x$ (we have called it $true$ just in order to better handle it in our expressions.)

As discussed before, since we wish to have a typed calculus and we wish to deal with recursion, we need to introduce the notion of recursion *explicitly*, by means of a recursor operator constant: $fix$.

The set of terms of $LC^+$ can be defined almost as the set of terms of the pure lambda-calculus. The terms are build using abstraction and application starting from variables and constants.

Adding constants to our calculus, however, it is not enough.

If, from now on, you ask your friends to call you "Carlo Rubbia", is this enough to make you a great scientist? Of course not. It is not the name "Carlo Rubbia" that makes a person a great scientist. It is how the person "behaves". A cat is a cat not beacuse you call it "cat", but because it catches mice.

So, it is not enough to extend the lambda-calculus with a constant $add$ in order to have the
operation of addition in $LC^+$. We need to introduce in $LC^+$ something that makes $add$
behave like the addition operation.
So, we add new axioms in the reduction theory of $LC^+$, besides the one of beta-reductions. Axioms that specify the computational "behaviour" of constants, axioms like

\[
\begin{align*}
\text{add } 1 \ 2 & \rightarrow_{\text{add}} 3 \\
\text{add } 4 \ 5 & \rightarrow_{\text{add}} 8 \\
\end{align*}
\]

etc. etc.
And all those that are necessary for the constants we have, like

\[
\begin{align*}
\text{if } \text{true} \ M \ N & \rightarrow_{\text{if}} M \\
\text{if } \text{false} \ M \ N & \rightarrow_{\text{if}} N \\
\text{first } [M,N] & \rightarrow [\_] M \\
\end{align*}
\]

etc. etc.
Of course we have not to forget the axiom describing the computational behavior of $fix$:

\[
\text{fix } F \rightarrow_{\text{fix}} F(\text{fix } F)
\]

We can now take into account the type system a' la Curry for $LC^+$.
Since we have constants for numbers, booleans, etc., we need also to have base types like $\text{INT}$, $\text{BOOL}$, etc.
Of course in our system we have arrow types, but since we have introduced pairs, we need to have also product types. That is, if $T_1$ and $T_2$, also $T_1xT_2$ is a type.
Now, in order to have a reasonable type system, we cannot allow constants to have arbitrary types. When we use our formal system to infer a type for a term it would be unreasonable to assume that $3$ has type $\varphi \rightarrow \text{BOOL}$. So, what we need to do is to prevent constants to appear in a basis $B$ and to extend our type inference system with new axioms, like

\[
B \vdash 3 : \text{INT}
\]

and

\[
B \vdash \text{add} : \text{INT} \rightarrow \text{INT} \rightarrow \text{INT}
\]

for any possible basis $B$.
And so on.
Notice that in the axiom for $\text{if}$ we need to specify that it can be polymorphically typed:

\[
B \vdash \text{if} : \text{BOOL} \rightarrow S\varphi \rightarrow S\varphi \rightarrow S\varphi
\]

for any possible basis $B$ and substitution $S$.
For what concerns the $\text{fix}$ operator, its typing axiom is obviously

\[
B \vdash \text{fix} : (S\varphi \rightarrow S\varphi) \rightarrow S\varphi
\]

for any possible basis $B$ and substitution $S$.
To deal with product types we need to add an inference rule:

\[
B \vdash M : \sigma \quad B \vdash N : \tau
\]

\[
B \vdash [M,N] : \sigma \times \tau
\]
Of course the property of strong normalization for typeable term does not hold anymore in \( LC^+ \).

We can now use \( LC^+ \) and its typing system to describe particular notions of programming languages like Haskell and to study their properties. For instance we can formally define the lazy reduction strategy of Haskell by means of the following formal systems for typeable terms of \( LC^+ \).

**The Lazy reduction strategy**

**Axioms**

\[
M \rightarrow^{\text{lazy}} N
\]

If \( M \rightarrow N \) is a reduction axioms, that is the one of beta-reduction or one for the operator constants, for instance \( if \text{ true} \) \( M \rightarrow \text{ if} M \).

**Rules**

\[
M \rightarrow^{\text{lazy}} M'
\]

\[
\text{add} \ M \ N \rightarrow^{\text{lazy}} \text{add} \ M' \ N
\]

\[
M \rightarrow^{\text{lazy}} M'
\]

\[
\text{add} \ n \ M \rightarrow^{\text{lazy}} \text{add} \ n \ M'
\]

if \( n \) is a numeric constant

\[
M \rightarrow^{\text{lazy}} M'
\]

\[
\text{Eq} \ M \ N \rightarrow^{\text{lazy}} \text{Eq} \ M' \ N
\]

\[
M \rightarrow^{\text{lazy}} M'
\]

\[
\text{Eq} \ n \ M \rightarrow^{\text{lazy}} \text{Eq} \ n \ M'
\]

if \( n \) is a numeric constant

\[
M \rightarrow^{\text{lazy}} M'
\]

\[
\text{if} \ M \ N \ P \rightarrow^{\text{lazy}} \text{if} \ M' \ N \ P
\]
\[
\text{first} \ M \rightarrow_{\text{lazy}} \text{first} \ M' \\
\text{M} \rightarrow_{\text{lazy}} \text{M}'
\]
\[
\text{second} \ M \rightarrow_{\text{lazy}} \text{second} \ M'
\]
\[
\text{M} \rightarrow_{\text{lazy}} \text{M}' \\
\text{MN} \rightarrow_{\text{lazy}} \text{M}'\text{N}
\]

Notice that the lazy reduction strategy does not reduce neither inside an abstraction nor inside a pair.

Of course it is possible to define extended version of the lambda-calculus also with type systems a' la Church (the system PCF is a well known example of typed extended lambda-calculus a' la Church).