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 <u>Introduction</u>

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 <u>imperative</u> programming languages (coming from the Turing Machine model) are:

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

In <u>functional</u> 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:
  - when we evaluate a mathematical expression we do not modify anything.
- variable - intended in a mathematical sense:
  - unknown entity
  - abstraction from a concrete value

1.2 <u>Functional programming</u>

A program in a functional language is 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, in such a way its meaning turns out to be represented in an explicit way. In general in programming we have not to *produce* anything; in functional programming, in particular, what we modify is <u>only</u> the *representation* of the information.

A function can be defined in several ways.

Example:

> *f: R -> R ,   g: R -> R*
> *f '' - 6g' = 6 sen x*
> *6g'' + a² f ' = 6 cos x*
> *f(0) = 0, f '(0) = 0, g(0) = 1, g'(0) = 1*

> *This set of equations precisely identify two precise f and g.*

A function definition in a functional programming language, however, has not simply to uniquely identify a mathematical function on a domain of objects, but it is also necessary that such a definition provides a means to compute the function on a given argument.

Example: function definition in Scheme and in Haskell

> *(define Inctwo (lambda (n) (+ n 2)))*

> *Inctwo n = n + 2*

To evaluate an expression in Scheme (as in Haskell) means to look for a sub-expression which is the application of a function to some arguments and then to evaluate the body of the function in which the formal parameter is replaced by the actual parameter. In this way, in a sense, we make the meaning of such sub-expression more *explicit*.

Example: let us evaluate Inctwo 3

> *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, and then we evaluate 3+2 obtaining 5*

In the example above we have seen two kinds of computational steps from *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 led me in a never ending path (this fact is unavoidable); but I would like to be sure that all finite path lead me to the very same value (this will be guaranteed by the Church-Rosser Theorem of the Lambda-calculus).

Example: In Haskell

   inf = inf   // definition of the value inf
  alwayseven x = 7 // definition of the function alwayseven
  alwayseven inf   // expression to be evaluated

Two possible choices in the evaluation of alwayseven Inf:

  1. *We evaluate inf first (the evaluation strategy of Scheme).*
  2. *We evaluate alwayseven inf first (the evaluation strategy of Haskell, a lazy language).*
case 1. Replace inf by inf   => we go on endlessly
case 2. Replace alwayseven inf by 7 => 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.

   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 then: 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 <u>Sketchy introduction to types</u>

Some functional programing 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

> $+::$ *int x int* $\rightarrow$ *int*

In (strongly) typed languages this error is detected by the interpreter, hence before the program is run. In not (strongly) typed languages this error is detected at run-time (the program stops during its run). In some typed functional languages it is possible to define *polymorphic* functions, that is which, roughly, have the same behaviour on arguments of different types (these functions have a polymorphic type: a type describing a whole set of (uniform) behaviours).

Example:

>   *ident:: a* $\rightarrow$ *a (where a is a type-variable, it denotes a "generic" type.)*
>   *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 istance

>   *Forall x : Int Exists y: Int . y=x+1*

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

1.4 <u>Higher order functions and curryfication</u>

In functional programming languages functions are treated as any other object. Hence they can be given as argument to other functions or be the result of a function. Function that can have functions as argument or result are called *higher-order functions*. This enables us to introduce the notion of *curryfication*, a relevant concept in functional languages. To currify 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 $f_{cur}(a_1)(a_2)..(a_{n-1})$ and such that $f(a_1,..,a_n)=f_{cur}(a_1)..(a_n)$.

A binary function of type:

   A x B → C

is curryfied to a function of type:

   A → (B → C)

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

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:
   an element of List is either [] or elem : 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* a computation tool.

- One starts from the basis case;
- By assuming a property to hold for n, one shows the property for n+1

(this is the principle of numerical induction; it corresponds to the mechanism of defining a function on numbers by recursion; We shall see that induction principles on complex inductive data types correspond to the general mechanism of defining function by recursion)

Disadvantages of 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. We need to be careful, in functional programming a function that, say, take a numerically labeled tree and increment its labels, does not modify the input. It simply builds a new tree with the needed characteristics. The same when we push or pop elements from a stack: each time a new stack is created. This cause 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 implementation 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 <u>Lambda-terms</u>

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

> variable   (*formalisable by   x, y ,z,…)*
> abstraction   *(formalisable by   λx.M   where M is a term and x a variable)*
> application   (*formalizable by M N, where M and N 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 we can found our computational model just on these concepts. Out of the formalization of these three concepts we form the lambda-terms. These terms will represent both *programs* and *data* in our computational model (whose strenght is indeed also 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:

> Λ **::=** X | (ΛΛ) | λX**.**Λ

where Λ 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, x2, x2,...)
Variable and *application* are well-known concepts, but **what is abstraction** ?
In our sense it is a notion tied to that of "abstracting from a concrete value" (disregarding the specific aspects of a particular object, that in our case is an input.) It is needed to formalize the notion of anonymous functions creation (when we have given the example of definition in Haskell of the function fact, we have done indeed two things: we have specified a function and we have given to it a name, fact. To be 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". But how is it possible to define a function like fact without being able to refer to it trought 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 <u>abstract</u> from the actual values of the input by simply saying that the function square associates the generic input x to x*x.
We can use the mathematical notation x ⊢─► x*x to specify a particular relation between domain and codomain; this is a mathematical notation to specify a function without assigning to it a particular name (an anonymous function).

In Lambda-calculus we describe the relation input-output by means of the lambda-abstraction operator: λx. x*x (anonymous function associating x to x*x). Other notions, usually considered as primitive, like basic operators (+, *, -, ...) or natural numbers are not necessary in the definition of the terms of our functional computational model (indeed we shall see that they can be considered, in our model, as <u>derived</u> concepts.)

2.2 <u>Free and bound variables</u>

Definition of free variable (by induction on the structure of the term)
We define the FV (Free Variables), the function which gives the free variables of a term as:

   FV(x) = {x}
   FV(PQ) = FV(P) ∪ FV(Q)
   FV(λx.P) = FV(P) \ {x}

Definition of bound variable (by induction on the structure of the term)
The function BV (Bound Variables) is defined by:

   BV(x) = Φ (the emptyset)
   BV(PQ) = BV(P) ∪ BV(Q)
   BV(λx.P) = {x} ∪ BV(P)

In a term M we abstract with respect to the variable x (λx.M) then x is said to be bound in M.

2.3 <u>Substitution</u>

With the notation **M [L / x]** we denote the term M in which any **free** 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 = λy.P) then:

$$\lambda y.P\ [L/x] \equiv \begin{cases} \lambda y.P & \text{if } x=y \\[2ex] \lambda y.(P\ [L\ /x]) & \text{if } x \neq y \end{cases}$$

Notice that in a lambda abstraction a substitution is performed only in case the variable to substitute is not bound.

This definition, however, is not completely correct. Indeed it does not prevent some harmful ambiguities and errors:

Example:

(λx.z)   and   (λy.z)   they both represent the constant function which returns z for any argument

Let us assume we wish to apply to both 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 (λx.z) we get:

(λx.z) [x / z] => λx.(z[x / z]) => λx.x   representing the identity function
(λy.z) [x / z] => λy.(z[x/ z]) => λy.x   representing the function that returns always x

This is absurd, since both the initial terms are intended to denote the <u>very same thing</u> (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 FV(L) and BV(M) be disjunct sets.

Example:

let us consider the following substitution:

(λz.x) [zw/x]

The free variable z in the term to be substituted would become bound after the substitution. This is meaningless. So, what it 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:

(λq.x) [zw/x];   now, by performing the substitution, we correctly get: (λq.zw)

2.4 The formalization of the notion of computation in λ-calculus: The Theory of β-reduction

In a functional language to compute means to evaluate expressions, making the meaning of the expression itself more and more explicit. In our computational model for functional programming, the Lambda-calculus, we then need to formalize also what a computation is. In particular we need to formalize the notion of "basic computational step".

Formalization of the notion of computational step (the notion of β-reduction):

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

This notion of reduction expresses what we intend by "computational step": Inside the body M of the function λx.M the formal parameter x is replaced by the actual parameter (argument) given to the function.

<u>Note</u>: A term like **(λx.M)N** is called **redex**, while **M[N/x]** is its **contractum**.

What does it mean that a computation step is performed on a term?
It means that the term contains a subterm which is a redex and such a redex is substituted by its contractum (β-reduction is applied on that redex.)
More formally, a term M β-reduces in one-step to a term N if N can be obtained out of M by replacing a subterm of M of the form (λx.P)Q by P[Q/x]

Example:

    (λx. x*x) 2 => 2*2 => 4

    here the first step is a β-reduction; the following 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 λ-terms).

Example:

    (λz. (zy) z) (x (xt)) $\longrightarrow_\beta$ *(x (xt) y) (x (xt))*

Example:

    (λy. zy) (xy) $\longrightarrow_\beta$ *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 even if there is this small ambiguity, the beta reduction can be performed without changing the name of the bound variables. Why?

The notion of 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 the notion of differing just for bound variables names in a very formal way, by defining the relation of **α-convertibility**. Here is an example of two terms in such a relation.

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

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

Obviously alpha convertibility mantains completely unaltered both the meaning of the term and how much this meaning is explicit. From now on, we shall implicitely work on λ-terms modulo alpha conversion. This means that from now on two alpha convertible terms like λz.z and λx.x are for us the **very same** term.

Example:

(λz. ((λt. tz) z)) z

All the variables, except the rightmost z, are bound, hence we can rename them (apply α-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 α-conversion, the one of bound variables obviously not.

A formal system for beta-reduction

If one wishes to be very precise, it is possible to define a formal system expressing in a detailed way the notion of one-step evaluation of a term, that is of one-step beta reduction.

First recall roughly what a formal system is.

Definition

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

Example:

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

We can now introduce a formal system whose only axiom is the β-rule we have seen before:

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

and whose inference rules are the following:

if

$P \longrightarrow_\beta P'$	$P \longrightarrow_\beta P'$	$P \longrightarrow_\beta P'$
$\lambda x.\ P \longrightarrow_\beta \lambda x.\ P'$	$L\ P \longrightarrow_\beta L\ P'$	$P\ L \longrightarrow_\beta P'\ L$

then we can infer

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

Example:

   in this formal system we can have the following derivation:

$$\frac{\dfrac{(\lambda x.x)\ y \longrightarrow y}{(zz)\ ((\lambda x.x)\ y) \longrightarrow (zz)\ y}}{\lambda w.((zz)\ ((\lambda x.x)\ y)) \longrightarrow \lambda w\ ((zz)\ y)}$$

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

Actually we are defining 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 $\longrightarrow_\beta$ 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**

Abbreviating nested abstractions and applications will make curried function easier to write. We shall abbreviate

$$(\lambda x_1.(\lambda x_2.\ ...\ .(\lambda x_n.M)\ ...\ ) \quad \text{by} \quad (\lambda x_1 x_2\ ...\ x_n.M)$$

$$(\ ...\ ((M_1 M_2)M_3)\ ...\ M_n) \quad \text{by} \quad (M_1 M_2 M_3\ ...\ M_n)$$

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

Example:

(λx.(x (λy.(yx))))   *can be written as*   *λx.x(λy.yx)*

2.6 <u>Curried functions</u>

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

As we mentioned before, if we have, say, the addition function, is is always possible to use its currified version AddCurr : **N $\longrightarrow$ (N $\longrightarrow$ N)**   and any time we need to use it, say *Add*(3,5), we can use instead ((*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 not only the use of curried functions does not make us lose computational power, but instead, can give us more expressive power, since, for instance, if we wish to use a function that increments an argument by 3, instead of defining it we can simply use *AddCurr*(3).

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

2.7 <u>Multi-step reduction</u>

Strictly speaking, M $\longrightarrow$ 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.

$$\text{we write}\quad M \longrightarrow\!\!\!\!\!\twoheadrightarrow N \quad \text{if}\quad M \longrightarrow M_1 \longrightarrow M_2 \longrightarrow ... \longrightarrow M_k \equiv N \quad (K \geq 0)$$

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

Example:

((λz.(zy))(λx.x)) —» y

2.8 <u>Normalization and possibility of non termination</u>

Definition
If a term contains no β-redex it is said to be in **normal form**. For example, *λxy.y* and *xyz* are in

normal form.

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

For example:

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

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

$$M \longrightarrow\!\!\!\!\gg Q$$

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

Example:

(λx.y)((λz.z)t)

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

1. (λx.y)((λz.z) t) → y

2. (λx.y)((λz.z) t) → (λx.y) t) → y

To *normalize* a term means to apply reductions until a normal form (if any) is reached. Many λ-terms cannot be reduced to normal form. A term that has not normal form is:

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

This term is usually called **Ω**.

Although different reduction sequences cannot yield different normal forms, they can yield completely differend outcomes: one could terminate while the other could run forever. Typically, if M has a normal form and admits an infinite reduction sequence, it contains a subterm L having no normal form, and L can be erased by some reductions. The reduction

$$(\lambda y.a)\mathbf{\Omega} \longrightarrow a$$

reaches a normal form, by erasing the **Ω**. This corresponds to a *call-by-name* treatment of arguments: the argument is not reduced, but substituted 'as it is' into the body of the abstraction. Attempting to normalize the argument generate a non terminating reduction sequence:

$$(\lambda y.a)\Omega \longrightarrow (\lambda y.a)\Omega \longrightarrow ...$$

Evaluating the argument before substituting it into the body correspons to a *call-by-value* treatment of function application. In this examples, 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 **value** in programming languages. In fact for us, intuitively, a value is something that cannot be further evaluated. But we cannot state a-priori what is, in general, a value, since there can be different possible interpretations of the notion of *value*. In Scheme, for instance, a lambda expression **is** a value, that is it cannot be evaluated anymore.
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 in general the notion of value, but what is a value has to be specified according to the context we are in. When we define a functional programming language we need to specify, among other things, also what we intend by value in our language.

2.9 <u>Some important results of β-reduction theory</u>

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

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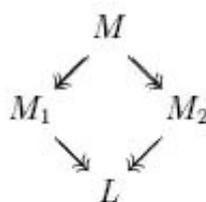
Theorem (Confluence)
If we have terms M, M2 and M3 such that:

$$M \longrightarrow\!\!\!\!\twoheadrightarrow M_1 \quad \text{and} \quad M \longrightarrow\!\!\!\!\twoheadrightarrow M_2$$

then there always exists a term L such that:

$$M_1 \longrightarrow\!\!\!\!\twoheadrightarrow L \quad \text{and} \quad M_2 \longrightarrow\!\!\!\!\twoheadrightarrow L$$

That is, in diagram form:

[Sketck of the Proof of the Theorem of Confluence (in italian)](#)

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

<u>Proof</u>:

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

$$M \longrightarrow\!\!\!\!\gg N \quad \text{and} \quad M \longrightarrow\!\!\!\!\gg N'$$

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

$$N \longrightarrow\!\!\!\!\gg Q \quad \text{and} \quad N' \longrightarrow\!\!\!\!\gg Q$$

But N and N' are in normal form, then they have not any redex. It means:

$$N \longrightarrow\!\!\!\!\gg Q \text{ in 0 steps}$$

then $N \equiv Q$. Analogously you can prove $N' \equiv Q$, then $N \equiv N'$.

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

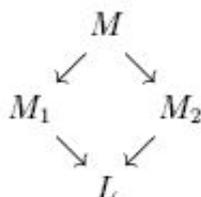if aRb and aRc, then there exist d such that bRd and cRd

Using the above definition we can reword the theorem of confluence in simpler way:
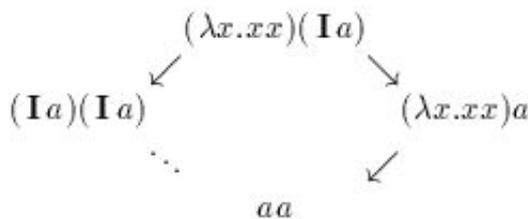
$$DP(\longrightarrow\!\!\!\!\gg)$$

Note that:

$$\neg\, DP(\longrightarrow)$$

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

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

$$
\begin{array}{ccc}
 & (\lambda x.xx)(\mathbf{I}\, a) & \\
 \swarrow & & \searrow \\
(\mathbf{I}\, a)(\mathbf{I}\, a) & & (\lambda x.xx)a \\
 \ddots & & \swarrow \\
 & aa &
\end{array}
$$

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

2.10 <u>Formalizing the notion of equivalent programs: the theory of β-conversion</u>

In the actual programming activity we usually talk about "computational equivalence" of pieces of code or of whole programs, for instance when trying to do some optimization. Being able to formally deal with such a notion would be useful for many things, like safely modifying and optimizing programs or for proving properties of programs. Let us try to formalize the notion of *equivalence* of programs into the lambda calculus. In order to do that we can ask ourself what does it means for two terms to be *computationally equivalent*. Well, informally we could say that two terms are so when they "contain 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]* $\longleftarrow$ *(λx.P)Q*

That is, if and only if there exist terms $N_1,..,N_k$ (k$\geq$ 0) such that

$$M \longrightarrow\!\!\!\!\!\twoheadrightarrow N_1 \twoheadleftarrow\!\!\!\!\!\longleftarrow M_1 \longrightarrow\!\!\!\!\!\twoheadrightarrow N_2 \twoheadleftarrow\!\!\!\!\!\longleftarrow M_2 \,...\, M_{k-1}\longrightarrow\!\!\!\!\!\twoheadrightarrow N_k \twoheadleftarrow\!\!\!\!\!\longleftarrow M_k = N$$

Let us define now a formal system allowing to derive all, and only, valid judgements of beta-convertibility of the form "M is β-convertible to N".

A formal system for β-conversion
<u>Axiom</u>:

$$\frac{\rule{3cm}{0.4pt}}{(\mathbf{\lambda x.M)\, N) =_\beta M\ [N/x]}}$$

Inference rules:

$$\text{if}\quad \frac{P =_\beta P'}{\lambda x.\, P =_\beta \lambda x.\, P'} \qquad \frac{P =_\beta P'}{Q\,P =_\beta Q\,P'} \qquad \frac{P =_\beta P'}{P\,Q =_\beta P'\,Q}$$

(then we can infer)

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

$$M =_\beta M \qquad \frac{M =_\beta N}{N =_\beta M} \qquad \frac{L =_\beta M \quad M =_\beta N}{L =_\beta N}$$

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 corresponds to the formalization of the notion of equivalence between (functional) programs, since this relation equates terms containing the "same information". But if we look at a term as a function (this is reasonable, since any term can be applied to any other term), what does it mean for two terms to be β-equivalent? Does it mean that they simply represent the same function? That is they "return" the same output when given the same input?

If this were the case the relation of β-equivalence would not be useful from the point of view of the computer scientist, 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 when they compute the same function. In fact, let us take the two terms $(\lambda x.yx)$ and y. They can be considered to compute the same function, since if apply the same input, say M, we get the same output

1. *(λx.yx)M —» yM*

2. *yM —» yM*

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

$$(\lambda x.yx) \neq_\beta y$$

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 λnfx**.**(f(nfx)) and λnfx**.**(nf(fx)) 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 (=$_{βη}$)
The formal system for =$_{βη}$ is like the one for =$_β$, but it contains an extra axiom:

$$\textbf{λx.Mx =}_{\textbf{η}} \textbf{ M}\quad\textbf{if x is not in FV(M)}$$

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:

$$\frac{\textbf{For all L: ML = NL}}{\textbf{M = N}}$$

Obviously, from a Computer Science point of view we are more interested in =$_β$ than in =$_{βη}$

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

$$\textit{(λx.yx) =}_{βη}\textit{ y}$$

Remark: you could now wonder why the two non β-convertible terms considered before, λnfx**.**(f(nfx)) and λnfx**.**(nf(fx)) 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 β-conversion and let us have a look at a few relevant results of the theory of β-conversion.

Theorem (Church-Rosser)
If two terms are beta-convertible (M =$_β$ N), then there exists Q such that

$$M \longrightarrow\!\!\!\!\gg Q\quad\text{and}\quad N \longrightarrow\!\!\!\!\gg Q$$

The Church-Rosser theorem enables to prove that the theory of β-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 ¬ P. In general, a theory is consistent if not all judgements are provable. Thanks to the Church-Rosser theorem, we can show that the λ-calculus is consistent.

Corollary (Consistency of $=_\beta$)

There exist at least two terms that are not β-convertible

Proof:

Using the Church-Rosser theorem, we can prove:

$$(\lambda x.yx) \neq_\beta y$$

The two terms *(λx.yx)* and *y* are both in normal form. So, by the definition of β-conversion, they can be β-convertible only if

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

But that is not the case, so they are not β-convertible.

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

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 => 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

$$M \longrightarrow\!\!\!\!\gg M_1 \quad \text{and} \quad M \longrightarrow\!\!\!\!\gg M_2$$

By definition of β-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

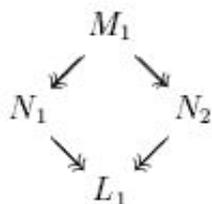$$M_1 \longrightarrow\!\!\!\!\gg L \quad \text{and} \quad M_2 \longrightarrow\!\!\!\!\gg L$$

CONF => C-R

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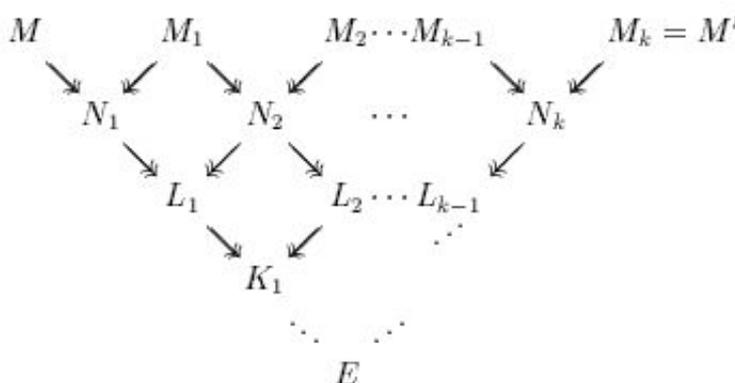
$$M =_\beta M'$$

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

$$M \longrightarrow\!\!\!\!\twoheadrightarrow N_1 \ll\!\!-\!\!\! M_1 \longrightarrow\!\!\!\!\twoheadrightarrow N_2 \ll\!\!-\!\!\! M_2 \ ... \ M_{k\text{-}1} \longrightarrow\!\!\!\!\twoheadrightarrow N_k \ll\!\!-\!\!\! M_k = M'$$

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

$$\begin{array}{ccc} & M_1 & \\ \swarrow & & \searrow \\ N_1 & & N_2 \\ \searrow & & \swarrow \\ & L_1 & \end{array}$$

Iterating the process we get:

$$\begin{array}{ccccccc} M & & M_1 & & M_2 \cdots M_{k-1} & & M_k = M' \\ & N_1 & & N_2 & \cdots & N_k & \\ & & L_1 & & L_2 \cdots L_{k-1} & & \\ & & & K_1 & & & \\ & & & & E & & \end{array}$$

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 *a*R*b* implies *b* belongs to S

Theorem (Undecidability)

Let A be a set of lambda-terms with is not trival (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 (it is not possible in general, given a term, to decide that it belongs to A or not).

3. Representation of data types and operations in λ-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 so primitive concepts. As a matter of fact we can represent them using the essential notions already contained 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 it 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 *λfx. 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** be zero and **what** be the successor. Do you?).

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

> $\underline{0} \equiv \lambda fx.\, x$ (it represents the natural number 0)
> $\underline{1} \equiv \lambda fx.\, f\, x$ (it represents the natural number 1)
> $\underline{2} \equiv \lambda fx.\, f\,(f\, x)$ (it represents the natural number 2)
> $\underline{3} \equiv \lambda fx.\, f\,(f\,(f\, x))$ (it represents the natural number 3)
> ....
> $\underline{n} \equiv \lambda fx.\, f^{n}x$ (it represents the natural number n)

where we define $f^{n}x$ as follows: $f^{0}x = x$; $f^{k+1}x = f(f^{k}x)$

$\underline{n}$ denotes the lambda-representation of the number n.

The above encoding of the natural numbers is the original one developed by Church and the terms just defined are called **Church's numerals**. Church's encoding is just one among the many one 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 emptytree 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 means for a function on natural numbers to be *definable* in the lambda-calculus?

Definition (λ-definability)

A function f: **N** $\longrightarrow$ **N** is said to be **λ-definible** (representable in the λ-calculus) iff there exists a λ-term F such that:

$$F \; \underline{n} =_\beta \underline{f(n)}$$

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 <u>fact</u>) 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*

$$\underline{fact} \; \underline{3} =_\beta \underline{6}$$

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 curryfied 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   AddCurr : **N** $\longrightarrow$ **(N** $\longrightarrow$ **N)**

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

3.2 <u>The Booleans</u>

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

> **if true** M N = M
> **if false** M N = N

The following encoding is usually adopted:

> **true** ≡ λxy.x
> **false** ≡ λxy.y
> **if** ≡ λpxy.pxy

Check that **if** actually works.

We have **true** $\neq$ **false** by the Church-Rosser theorem, since **true** and **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:

and $\equiv \lambda pq.$ **if** $p\ q$ **false**
or $\equiv \lambda pq.$ **if** p **true** q
not $\equiv \lambda p.$ **if** p **false true**

3.3 <u>Arithmetic on Church's numerals</u>

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

add $\equiv \lambda mnfx.mf\,(nfx)$
mult $\equiv \lambda mnfx.m\,(nf)\,x$
expt $\equiv \lambda mnfx.nmfx$

Addition is not hard to check. In fact:

$$\mathbf{add}\ \underline{m}\ \underline{n}\ \equiv\ (\lambda mnfx.mf\,(nfx))\ \underline{m}\ \underline{n} \longrightarrow_\beta (\lambda nfx.\ \underline{m}\ f\,(nfx))\ \underline{n} \longrightarrow_\beta$$

$$\longrightarrow_\beta \lambda fx.\ \underline{m}\ f\,(\underline{n}\ f\ x) \equiv \lambda fx.(\lambda gx.g^m\ x)\ f\,((\lambda hx.h^n\ x)\ fx) \longrightarrow_\beta$$

$$\longrightarrow_\beta \lambda fx.(\lambda gx.g^m\ x)\ f\,((\lambda x.\ f^n\ x)\ x) \longrightarrow_\beta \lambda fx.(\lambda gx.g^m\ x)\ f\,(f^n\ x) \longrightarrow_\beta$$

$$\longrightarrow_\beta \lambda fx.(\lambda x.f^m\ x)(f^n\ x) \longrightarrow_\beta \lambda fx.f^m\,(f^n\ x) \equiv \lambda fx.f^{m+n}\ x \equiv$$

$$\equiv\ \underline{m+n}$$

Multiplication is slightly more difficult:

$$\mathbf{mult}\ \underline{m}\ \underline{n}\ \equiv\ (\lambda mnfx.m(nf)\,x)\ \underline{m}\ \underline{n} \longrightarrow_\beta (\lambda nfx.\ \underline{m}\ (nf)\,x)\ \underline{n} \longrightarrow_\beta$$

$$\longrightarrow_\beta \lambda fx.\ \underline{m}\ (\underline{n}\ f)\ x \equiv \lambda fx.(\lambda gx.g^m\ x)\ (\underline{n}\ f)\ x \longrightarrow_\beta \longrightarrow_\beta$$

$$\longrightarrow_\beta \lambda fx.(\underline{n}\ f)^m\ x \longrightarrow \lambda fx.((\lambda hx.h^n\ x)\ f)^m\ x \longrightarrow_\beta \lambda fx.(f^n)^m\ x \equiv \lambda fx.f^{m\times n}\ x \equiv$$

$$\equiv\ \underline{m\times n}$$

These reductions hold for all Church's numerals $\underline{m}$ and $\underline{n}$,.

Here are other simple definitions:

succ $\equiv \lambda nfx.\ f\,(nfx)$
iszero $\equiv \lambda n.n\,(\lambda x.\ \mathbf{false})\ \mathbf{true}$

The following reductions, as you can check by yourself, hold for any Church's numeral $\underline{n}$:

succ $\underline{n} \longrightarrow\!\!\!\gg \underline{n+1}$
iszero $\underline{0} \longrightarrow\!\!\!\gg$ **true**

iszero (<u>n+1</u>) —» **false**

Of course it is also possible to define the predecessor function (**pred**) in the lambda calculus, but we do not do that here.

Now we can easily give the representation of the following mathematical function un natural numbers

$$
f(n,m) = \begin{cases} 3+(m*5) & \text{if } n=2 \text{ or } n=0 \\ \\ (n+(3*m)) & \text{otherwise} \end{cases}
$$

by means of the following lambda-term:

λnm. **if** (**or** (**iszero** (**pred** (**pred** n))) (**iszero** n)) (**add** <u>3</u> (**mult** m <u>5</u>)) (**add** n (**mult** <u>3</u> m))

3.4 <u>Defining recursive functions using fixed points</u>

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

(define fact (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))))

or in Haskell

fact 0 = 1

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 ⟶ 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 : Λ ⟶ Λ, we can define a **fixed point** of M, if any, to be any term Q such that:

$$ M\,Q =_\beta Q $$

To see how we can fruitfully exploit the concept of fixed point, let us consider an example. Les us take the definition of factorial using a generic sintax

*factorial x = 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 function which satisfies the following equation where *f* can be looked at as an "unknown" function

*f x = if x=0 then 1 else x * f(x-1)*

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

*f =λx. if x=0 then 1 else x * f(x-1)*

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

*λf.λx. if x=0 then 1 else x * f(x-1)*

In fact it can be checked that

*(λf.λx. 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

*λf.λx. if x=0 then 1 else x * f(x-1)*

can be easily lambda-represented:

$$\lambda\ f.\lambda x.\textbf{if } (\textbf{iszero } x)\ \underline{1}\ (\textbf{mult } (f(\textbf{sub } x\ 1))\ x)$$

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

$$F\ x =_{\beta} \textbf{if } (\textbf{iszero } x)\ \underline{1}\ (\textbf{mult } x\ (F(\textbf{sub } x\ 1)))$$

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 **all** lambda-terms indeed 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 λ-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:

$$\mathbf{Y} \equiv \lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx))$$

It is easy to prove that **Y** is a fixed point combinator. Let F be an arbitrary λ-term. We have that

$$\mathbf{Y}\ F \longrightarrow (\lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx)))\ F \longrightarrow (\lambda x.\ F(xx))\ (\lambda x.\ F(xx)) \longrightarrow$$

$$\longrightarrow F\ ((\lambda x.\ F(xx))\ (\lambda x.\ F(xx))) \longrightarrow \qquad\qquad (3)$$

$$=\ F\ (\mathbf{Y}\ F)$$

That is

$$F\ (\mathbf{Y}\ F) =_\beta (\mathbf{Y}\ F)$$

Then the λ-term that represents the factorial function is:

$$\mathbf{Y}(\lambda f.\lambda x.\mathbf{if}\ (\mathbf{iszero}\ x)\ \underline{1}\ (\mathbf{mult}\ x\ (f(\mathbf{sub}\ x\ 1))))$$

Let us consider another example of recursively defined function.

$$g\ x\ =\ if\ (x{=}2)\ then\ 2\ else\ g\ x \qquad\qquad (*)$$

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

$$factorial\ x\ =\ if\ (x{=}2)\ then\ 2\ else\ factorial\ x$$

And also the identity function satisfies the equation, sknce

$$identity\ x\ =\ if\ (x{=}2)\ then\ 2\ else\ identity\ x$$

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

$$gee\ x\ =\ if\ (x{=}2)\ then\ 2\ else\ gee\ x$$

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

$$\lambda g.\lambda x.\ \textit{if } x=2 \textit{ then } 2 \textit{ else } g\ x$$

do we have to consider as **the** function identified by the definition *(*)*.
We all know that it is *gee* the intended function defined by *(*)*. Why?
Because implicitely, when we give a definition like *(*)*, we are considering a computational process that identifies the **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 (*), we cannot consider any fixed point, but the <u>least</u> fixed point (the one that corresponds to the least defined function that respects the equation.)

The question now is: does the **Y** combinator *picks* the right fixed point? If a lambda term has more than one fixed point, which is the one *picked* up **Y**?
Well, **Y** is really a great guy. It not only returns a fixed point of a term, but the least fixed point, where informally for least we mean the one that contains only the information strictly needed in order to be a fixed point. Of course we could be more precise in what we mean by a term "having less information" than another one and in what we mean by least fixed-point, but we would go beyond the scope of out introductory notes.

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

$$\lambda f.\ \lambda x.\ \textbf{if iszero}\ (\textbf{pred}\ (\textbf{pred}\ x))\ \underline{2}\ (f\ x)$$

and hence the lambda term lambda-representing the function *gee* is

$$\textbf{Y}(\lambda f.\ \lambda x.\ \textbf{if iszero}\ (\textbf{pred}\ (\textbf{pred}\ x))\ \underline{2}\ (f\ x))$$

Let us look at some other easy example
Example:

   Let us consider the λ-term:

$$\lambda x.x$$

   Any other term M is a fixed point of $\lambda x.x$ (the function identity) because:

$$\lambda x.x =_\beta M$$

   The least fixed point, between all the ones fixed points of $\lambda x.x$, is $\mathbf{\Omega}$ (intuitively you can see that this is the term that has no information at all in it). IWe can check that

$$Y\ (\lambda x.x) = \mathbf{\Omega}$$

Example:

   Let us consider, now, a function indefinite in all the domain, that is the one define by the equation:

$$h\ x = h\ x$$

   The λ-term that represents that, 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 the function indefinite in every value.

If we consider the (3), there we have two β-reductions (the first two), then a β-expansion. It means that it's impossible to have:

$$\mathbf{Y} \, F \longrightarrow\!\!\!\! » \, F \, (\mathbf{Y} \, F)$$

There is another fixed point combinator: Alan Turing's Θ:

$\quad$ A ≡ λxy.y (xxy)
$\quad$ Θ ≡ AA

Its peculiarity is that we have that

$$\Theta \, F \longrightarrow\!\!\!\! » \, F \, (\Theta \, F)$$

Then I can get the relation above, using only β-reductions.

The Why of Y.

Let us see intuitively noy **why** the combinator **Y** (and the fixed point combinators in general) **works**.

Let us consider once again the factorial function

*factorial x = if (x=0) then 1 else x * factorial(x-1)*

You see, in a sense this definition is equivalent to

*factorial x = if (x=0) then 1 else x * (if (x-1=0) then 1 else (x-1) * factorial(x-1-1))*

and also to

*factorial x = if (x=0) then 1 else x * (if (x-1=0) then 1 else (x-1) * (if (x-1-1=0) then 1 else (x-1-1) * 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...-1 gets equal to 0. That is exactly what the **Y** combinator does, in fact

$\mathbf{Y} \, F =_\beta F(\mathbf{Y} \, F) =_\beta F(F(\mathbf{Y} \, F)) =_\beta F(F(F(\mathbf{Y} \, F))) =_\beta F(F(F(F(\mathbf{Y} \, F)))) =_\beta$ ... and so on

If you take as F the lambda term *λf.λx.if (iszero x) 1 (mult x (f (sub x 1)))* , by applying a beta-reduction you get

    Y F =$_β$ *λx.if (iszero x) 1 (mult x ((Y F)(sub x 1)))*

but you can also get

Y F =$_β$ *λx.if (iszero x) 1 (mult x ((if (iszero (sub x 1)) 1 (mult (sub x 1) ((Y F)(sub (sub x 1) 1)))*

and so and so on

That is *(Y F)* is beta-convertible to all the possible "unwindings" of the factorial definition. Then if you apply *2* to *(Y F)* you can get, by a few beta-reduction

(Y F)2 =$_β$ *if (iszero 2) 1 (mult 2 ((if (iszero (sub 2 1)) 1 (mult (sub 2 1) (if (iszero (sub (sub x 1) 1)) 1 (mult (sub (sub x 1) 1) ((Y F)(sub (sub (sub x 1) 1) 1)))))))*

you can now easily see that the right term is beta convertible to *2* that is the factorial of *2*.

Then, why **Y** works? Because, in a sense, it is an "unwinder", an "unroller" of definitions.

Using Y (and hence recursion) to define infinite objects.

In lambda calculus it is easy to represent also the data type List. In fact it is possible to represent the empty list and the operators of list (we do not give here their precise representation).

Let us consider now the following λ-term:

$$\textbf{Y} (λl. \textbf{cons} \ x \ l) \qquad\qquad (*)$$

It represent a fixed point. We used **Y** before to represent recursively defined function in lambda-calculus, but it is not the only way to use **Y**. Here we use it to represent the fixed-point of the function λl. **cons** x l, whose fixed point, you can check is an infinite list, the infinite list of x's. This means that the term **Y** (λl. **cons** x l) indeed represent an infinite object!
Of course we cannot completely evaluate this term (i.e. get all the information it contains), since the value of this term would be an *infinitely* long term in normal form, but we can evaluate it in order to get **all** possible finite approximation of the infinite list of x's.
You see, after a few beta reductions and beta conversions you can obtain:

    (**cons** x (**cons** x (**cons** x (**Y** F))))   where F represents the term *(*)*

and this term is a finite approximation of the infinite list of x's. Of course the term (*) is convertible to all the possible finite approximations of the infinite list of x's. This means that it is possible to *use* and *work with* terms representing infinite object. In fact let us consider

$$\textbf{car} (\textbf{Y} (λl. \textbf{cons} \ x \ l)) =_β \ x \ (**)$$

where **<u>car</u>** is the λ-representation of the function that, taken a list, returns its first element. We can see that

it is possible manipulate the term *(*)*, even if it represent an infinite object. This possibility show that in functional programming it is possible to deal also with infinite objects. This possibility, however is not present in all functional languages, since to evaluate an expression in a functional language we precise reduction strategies, and not all the reduction strategies enable to handle infinite objects. In Scheme it is not possible since, in a term like (**) above Sheme would keep on evaluating the argument of the **car**, going on to the infinite. 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 Let us define such notion in the lambda-calculus.

Definition

A reduction strategy is a "policy" that, given a term, permits to choose a β-redex among all the possible ones present in the term. Abstractly, we can consider a strategy like a function

$$S: \Lambda \longrightarrow \Lambda$$

(where Λ is the set of all the λ-terms), such that:

$$\text{if } S(M) = N \text{ then } M \longrightarrow_\beta N$$

(M reduces to N in one step). Let us see, now, same classes of strategy.

Call-by-value strategies

These are all the strategies that never reduce a redex when the argument is not a value (that is, when we can still reduce the argument). 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:

$$((\lambda x.x)((\underline{\lambda y.y)\ z})) \ ((\lambda x.x)((\underline{\lambda y.y)\ z}))$$

   A call-by-value strategy never reduce the term $(\lambda x.x)((\lambda y.y)\ z)$ because the argument it is not a value. In a call-by-value strategy, then, I 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 I have:

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

Scheme's reduction strategy considers the biggest redex (all the term), reduces it and then stops because it get something that is a value for Scheme (for Scheme the term $\lambda z.((\lambda x.x)\ x)$ is a value and hence it is left as it is by the interpreter.).

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 strategy call-by-name 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:*

$$\underline{(I\ (II))}\ (\ I\ (II))$$

    the leftmost-outermost strategy first reduces the underlined redex.

This strategy is very important because if a term has a normal form, it finds that. The problem is that the number of reduction steps is sometimes greater then 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 ahead to the infinite.

Example:

    Let us consider the functional F that has the factorial function as fixed point. Then, we can say that:

$$(\Theta\ F)\ \underline{3} \longrightarrow\!\!\!\gg \underline{6}$$

    Let us suppose to use a call-by-value strategy. I get:

$$(\Theta\ F)\ \underline{3} \longrightarrow (F\ (\Theta\ F))\ \underline{3} \longrightarrow (F\ (F\ (\Theta\ F)))\ \underline{3} \longrightarrow ...$$

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 I could use in Scheme and any aty I have a reduction strategy like that used by Scheme. Let us consider the following λ-term:

$$\mathbf{H} = \lambda g.((\lambda f.ff)\ (\lambda f.(g\lambda x.ffx)))$$

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

$$(\mathbf{H}\ F) \longrightarrow\!\!\!\gg F\ (\lambda x.\ AAx)$$

where A is (λf.F λx.ffx) and (**H**F) —» AA. From the point of view of the computations, we can write it in this form:

$$(\textbf{H}\ F) \longrightarrow\!\!\!\!\gg F\ (\lambda x.\ \textbf{H}\ Fx)$$

Functionally, F (λx. **H** Fx) has the same behaviour of **H**F, but scheme does not reduce it because that's a value for scheme. Then if, for example, F is the functional whose fixed-point represents the function factorial, in order to compute fact 5 we have to evaluate the term ((**H**F)5).That is we have to evaluate

$$(F\ (\lambda x.\ \textbf{H}\ Fx))\ \underline{5}$$

and this evaluation is possible withouth getting an an infinite reduction sequence.

3.6 Head Normal Form

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

$$\boldsymbol{\Omega}\ \text{has not normal form and it represents the indefinite value}$$

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.
Let us consider the following λ-term:

$$\textbf{Y}\ (\lambda l.\ \textbf{cons}\ x\ l) \qquad\qquad (*)$$

which represents the infinite list of x's. We cannot completely evaluate this term (i.e. get all the information it contains), since the value of this term would be an *infinitely* long term in normal form.
The term above does not have a normal form **but** nonetheless it has a "meaning" (the infinite list of x´s). In fact after some β-reductions I can get a *finite* approximation of its "meaning". You can see that after a few beta reductions we can obtain:

$$(\textbf{cons}\ x\ (\textbf{cons}\ x\ (\ \textbf{cons}\ x\ (\textbf{Y}\ F))))\quad \text{where F represents the term } (*)$$

and this term is a finite approximation of the infinite list of x's.

It is possible to *use* and *work with* terms that have <u>no</u> normal form but nonetheless posses a "meaning. In fact, let us consider:

$$\textbf{hd}\ (\textbf{Y}\ (\lambda l.\ \textbf{cons}\ x\ l)) \longrightarrow\!\!\!\!\gg x$$

where **<u>hd</u>** is the λ-representation of the function that, taken a list, returns its first element. We can see that
it is possible manipulate the term *(*)*, even if it represent an infinite object.

> *In this case, I have to be careful about the strategy I intend to use. If I use a call-by-value strategy, where I consider as values the terms in normal form, I get an infinite chain of β-reductions where in every step I reduce the subterm*

$$\mathbf{Y} \ (\lambda l. \ \mathbf{cons} \ x \ l)$$

> *In a language like Haskell, instead, I 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 it is possible to formalize in lambda-calculus the notion we have informally discussed above? that is, the notion of a term that contains some information even if it could also have no normal form.
We do that by introducing the concept of **head normal form**, representing 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 looks like this:

$$\lambda x_1...x_k.yM_1...M_n \quad (n, k \geq 0)$$

where y can be any variable, also one of the x_i's.
If this term had normal form, that it will be of the form:

$$\lambda x_1...x_k.yN_1...N_n$$

where $N_1,...,N_n$ would be the normal form of $M_1,...,M_n$. Instead, if the term had no normal form, I'm sure that in any case the portion of the term
$$\lambda x_1...x_k.y$$

will stay always the same in any possible reduction sequence.
(Let us note that a term in normal form is also in head normal form).

Example:

> *Let us consider the term:*

$$\lambda xyz.z(\lambda x.x)(\lambda zt.t(zz))(\lambda t.t(\lambda \ (\lambda xx)))$$

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

Let us consider, now, **Ω**. 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 contained also in terms having no normal form. So *when* can I say that *a term contain no information*? 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 Church-Rosser theorem I can prove they are not β-convertible. Now If I add to the β-conversion theory the axiom:

$$\Omega =_\beta \Omega x$$

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. Indeed I could consistently add to the theory of the beta-conversion the axiom stating that all the terms with no head normal form are β-convertible (pratically we could consistently "collapse" all the possible representations of the indeterminate into one element).

4. Types

4.1 <u>Introduction to types</u>

The functional programming languages can be typed or not typed. A typed one is, for example, Haskell, while one not typed is Scheme. Let us introduce the concept of type in λ-calculus. We can look at types as predicates on programs or, better, as **partial specifications** about the behaviour of a program. By means of types a number of programming errors can be prevented.

There are two different approach to the use of types.

- **the approach a' la Curry** (the one used in Haskell)
- **the approach a' la Church** (the one used in Pascal)
-

In the Curry approach, I usually first write a program and then I check whether it satisfies a specification.
Instead, in the Church approach the types are included in the syntax of the language and, in a sense, represent some constraints I have to follow during the writing of my code (they prevent me to write wrong terms).
In the approach a' la Curry, checking if a term M satisfies a specification represented by a type t, it is usually referred to as "assigning the type t to the term M".

4.2 <u>Type assignment 'a la Curry</u>

In this approach we write a program and then we check if the program satisfies the specification he have in mind for it. It means that we have two distinct sets: programs and 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:

<u>Programs</u> : the set of λ-terms, i.e.   /   <u>Partial specifications</u>: the set of types, i.e.

$$\Lambda \qquad\qquad\qquad\qquad T ::= \varphi \mid T \longrightarrow T$$

where **φ** is a meta-variable which value is defined over a set of type variables (φ, φ', φ", a, b, ... etc.). In this formalization we have considered the simplest form of type (in fact this types are called "simple types"). Elements of T in general will be denoted by σ, τ , ρ, etc.

There are now a whole bunch of questions we would like to answer:

1) How can I *formally* show that, given a program (a term) M and a specification (a type) σ, M satisfies the specification σ, that is that M has type σ?

2) Is it decidable, given a M and a σ, that M is of type σ?

3) Do types play any role during a computation?

4) Is it possible to "compare" types, that is to formalize the notion that a type is "more general" than another one?

5) What is the intrinsic meaning of having type variables in our system? has the use of type variable some implication for our system?

6) Is it possible that among all the possible types I can give to a term M, there exists the most general one? If so, can it be algorithmically found?

Let us start with **question 1)**.
So, what we need to do is to device a formal system whose judgments are of the form

$$M : \sigma$$

whose meaning is: the program M respects the specification σ (the term M has type σ).

Thinking about it better, however, the judgments we need should be a bit richer that simply M **:** σ.
In fact, let us consider the term λx.yx. It is possible to infer that this terms has type, for instance:

$$\varphi \rightarrow \textbf{something}$$

But how can we determine what **something** is if we do not have any information about the free variable y?
This means that the (typing) judgments we are looking for need to be of the shape:

$$B \vdash M : \sigma$$

where B provide informations about the free variables of M. The meaning of B ⊢ M **:** σ is then "using the informations given by the basis B, the term M has type σ" (M has type σ in the basis B).

The basis B contains informations about the types of the free variables of M.
Formally B will be a set of pairs of the form :   variable **:** type

Example:

    One example of basis is:

$$B = \{x : \tau , z : \rho\}$$

Let us introduce now the formal system.

Definition (Curry type assignment system)

Axiom

$$\frac{\quad\quad\quad\quad}{B \vdash x : \sigma}$$
        if x **:** σ is in B

(a variable x has type σ if the statement x **:** σ is in B).

Inference rules

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

-

$$\frac{B \vdash M : \sigma \to \tau \quad\quad\quad B \vdash N : \sigma}{B \vdash MN : \tau}$$

-

  This rule says that if, using the assumptions in B, I can prove that M has type σ → τ and N has type σ, then I can prove that in the basis B the application MN has type τ. This rule is usually called (→E): "arrow elimination". Of course this rule can be "read" also bottom-up, that is: if we wish to prove that MN has type τ in B, then we need to prove that M has type σ → τ in B for some σ and that N has type σ in B.

-

$$\frac{B, x : \sigma \vdash M : \tau}{B \vdash \lambda x.M : \sigma \to \tau}$$

-

  This rule says that if, using the assumptions in B and the further assumption "x has type σ", I can prove that M has type τ, then I can prove that in B λx**.**M has type σ → τ. This rule is called (→I) : "arrow introduction". Also this rule can be "read" bottom-up. Do it as exercise.

We remark that 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.

Example:

In Haskell, before the definition of a function, I can declare what type the function is intended to have:

$$fact :: Int \rightarrow Int$$

Once I have defined the function fact, the Haskell interpreter checks if the program satisfies the specification I had given, that is it tries to derive, in a formal system like the one seen before, a typing judgement.
<u>Note</u>: *In Haskell we have not the concept of basis, because all correct programs have to be closed expressions (without free variables). That is, better, all the free variables in a program must have been associated previously to some value.*

Example:

Let Ø be the empty set and let us consider the judgment

$$\emptyset \vdash \lambda xy.x : \sigma \rightarrow (\tau \rightarrow \sigma)$$

whose informal meaning is: starting from the empty basis, the term λxy.x satisfies the given specification. We want now to prove this judgment to be valid using the formal system seen before. Let's start from the axiom:

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

Using (→I) two times, we can prove:

$$\frac{\dfrac{\{x : \sigma, y : \tau\} \vdash x : \sigma}{\{x : \sigma\} \vdash \lambda y.x : \tau \rightarrow \sigma}}{\emptyset \vdash \lambda x.\lambda y.x : \sigma \rightarrow (\tau \rightarrow \sigma)}$$

Let us notice that in the last two steps, we discharge the statements from the basis because when a variable is not free anymore I can't leave it there (in the basis we have only assumptions for free variables because the information about a bound variable is already contained in the type of the term).

Example:

We want to prove that:

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

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

$$\cfrac{\cfrac{\overline{} \qquad \overline{}}{\cfrac{B \vdash f : \sigma \to \sigma \qquad B \vdash x : \sigma}{B \vdash (fx) : \sigma} \qquad \cfrac{\overline{}}{B \vdash f : \sigma \to \sigma}}{B \vdash f(fx) : \sigma} \qquad \cfrac{\overline{}}{B \vdash f : \sigma \to \sigma}}{B \vdash f(f(fx)) : \sigma}$$

I can still use the rules of the formal system and derive, from the one above, other judgments:

$$\cfrac{\cfrac{\{f : \sigma \to \sigma, x : \sigma\} \vdash f(f(fx)) : \sigma}{\{f : \sigma \to \sigma\} \vdash \lambda x.f(f(fx)) : \sigma \to \sigma}}{\varnothing \vdash \lambda f.\lambda x.f(f(fx)) : (\sigma \to \sigma) \to (\sigma \to \sigma)}$$

This last specification says that I can iterate f three times, only if f has the same type as both domain and codomain. The one above is the specification of $\underline{3}$ (Church's numeral).

Note:
There are programs that does 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 is and whatever type τ is. That is, there is no basis and type I can use to prove that the term *xx* has a type. What is the implication of this simple observation? Well, typed languages restrict the set of programs you can write, but the programs you can write are guaranteed not to contain certain errors and to satisfy their partial specifications. You loose something of course, that is flexibility, the possibility of writing maybe tricky an cunning programs. So, in typed languages: less flexibility = more safeness.

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

Saying that M : σ (M has type σ) amounts to say that M represents a *value* of type σ. The types do not give information about the form of the terms (about their syntax), but, instead, about their intrinsic meaning, that is about their semantics, the information they contain.
Then, if a term has a certain type, it is important to be sure that that during its evaluation the term "maintain" the same type.
For example, if a term M represents a program that returns a number, types would be useless and would have no sense 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 I can prove that M has type σ, that is

$$B \mathrel{|\!\!\!-\!\!\!-} M : \sigma$$

and if M —» N, then I ca also prove that

$$B \mathrel{|\!\!\!-\!\!\!-} N : \sigma$$

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 we are sure that all the terms that we obtain reducing M in a computation 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. Then, if:

$$B \mathrel{|\!\!\!-\!\!\!-} N : \sigma \quad \text{and} \quad M \text{ —» } N$$

is not always true that:

$$B \mathrel{|\!\!\!-\!\!\!-} M : \sigma$$

By means of the Subcject 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 of types:

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

$(\varphi \mapsto \rho) \, \varphi = \rho$
$(\varphi \mapsto \rho) \, \varphi' = \varphi'$, if $\varphi' \neq \varphi$
$(\varphi \mapsto \rho) \, \sigma \rightarrow \tau = ((\varphi \mapsto \rho) \, \sigma) \rightarrow ((\varphi \mapsto \rho) \, \tau)$

The above definition formalize 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 : Sτ | x : τ 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, σ> = <SB, Sσ>

Example: if we apply the type substitution (φ ↦ σ) to the type φ → (τ→ φ), we obtain the type   σ → (τ→ σ)

- If for σ,τ there is a substitution S such that Sσ = τ, then τ is a (substitution) *instance* of σ.

We answer to **question 2)** above by giving the following definition.

Definition (more general type)
Given two types τ and σ, τ is <u>more general</u> than σ if the latter can be obtained from the former by means of a type substitution, i.e. if there exists S such that τ = S(σ).

The following lemma show that if we can give a type to a term, we can give infinitely many and indeed it will provide an answer also to **question 5)**.

Lemma (substitution lemma)
If starting from a basis B we can prove that M has type σ, that is

$$B \vdash M : σ$$

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

$$S(B) \vdash M : S(σ)$$

Example:
   It's easy to prove that:

$$\{x : φ → ψ\} \vdash λy .xy : φ → ψ$$
   The lemma above says that if I apply any substitution S, I can prove that:

$$S(\{x : φ → ψ\}) \vdash λy .xy : S(φ → ψ)$$
   For example, if I consider the following substitution:

$$(φ ↦ (φ' → φ)) \circ (ψ ↦ ξ)$$
   then it is possible to prove that:

$$x : (φ' → φ) → ξ \vdash (λy .xy) : (φ' → φ) → ξ$$

So if a term satisfies a specification, then it satisfies infinitely many ones because we can have

infinite possible substitutions.

Example:

    If we have derived

$$\varnothing \vdash \lambda x.x : \varphi \to \varphi$$

by the lemma of substitution I am also sure that also the following is derivable

$$\varnothing \vdash \lambda x.x : (\psi \to \varphi) \to (\psi \to \varphi)$$

    But there are infinite substitutions that I can apply, so if a term satisfies a specification, it satisfies infinite specifications.

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 polimorphism. (In Haskell type variables have names like a, b, c etc.)

The polymorphism of our system can be said to be *implicit*, since the fact that a term has more types it is not <u>explicitly</u> expressed in the syntax but it is expressed by results like the substitution lemma.
One example of a syntax for explicit polymorphism could be the following:

$$\lambda x.x : \text{for all } \varphi.\varphi \to \varphi$$

In this case the polymorphic behaviour of the term is explicitely described by the sintax.

We have just said that if I have a term M, it has infinite specifications. It is possible to prove that exists a relation among all the types that a given term can have (i.e. all the possible partial specifications of a program are connected). We are then trying to give an answer to question 6). The answer will also provide an answer to question 2) that we had left unanswered.

4.3 <u>Principal pairs</u>

Theorem
Given a term M, if M is typeable then it has a **Principal Pair** $\langle P, \pi \rangle$, where P is a basis and π a type such that:

1) I can derive the following typing for M

$$P \vdash M : \pi$$

2) the principal pair represents the most generic typing possible for M. That is, I can get any other basis and type from that. It means that if:

$$B \vdash M : \sigma$$

then there exists S such that $B = S(P)$ (as a matter of fact B could be a *superset* of S(P)) and $\sigma = S(\pi)$. This partially answer our **question 6)**

Example:

    Let us consider the term:

$$\lambda x.x$$

    It has principal pair:

$$<\emptyset, \varphi \to \varphi>$$

    This means that any other typing for the identity is an istance of this.

The proof of the principal pair algorithm can be given by providing an algorithm that, given a term M, returns its principal pair, pp(M), and fails in case M be not typable.
These completes the answer of our **question 6)**.

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, φ : $pp(x) = < \{x : \varphi\},\varphi>$

b) If $pp(M) = <P, \pi>$ then:

    i) If x is in FV(M), then there is a σ such that $x : \sigma$ is in P, and $pp(\lambda x.M) = <P \setminus x, \sigma \to \pi>$
    ii) otherwise (x is not a free variable), $pp(\lambda x.M) = <P\varphi \to \pi>$ where φ 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 $<P_i, \pi_i>$ are disjoint in pairs), φ is a type-variable that does not occur in any of the pairs $<P_i, \pi_i>$, and:

$$S_1 = \textit{unify } \pi_1 \ (\pi_2 \to \varphi)$$
$$S_2 = \textit{UnifyBases}(S_1 \ P_1) \ (S_1 \ P_2)$$

then $pp(M_1M_2) = S_2oS_1<P_1 \cup P_2, \varphi>$. (*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 typable the algorithm fails, while 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, can have all its possible types,

since any possible typing can be got out of the principal pair.

Extensions of the above algorithms are used in languages like Haskell and ML. In fact if, after having defined a function, say:

$$Id \ x = x$$

we ask to the Haskell interpreter the following "question":

$$:type \ Id$$

what the interpreter does is to run an algorithm similar to the one above, producing the principal type
of the term associated to the identifier Id (the principal type and not the principal pair, since our Haskell expression need to be closed). For our example the interpreter returns:

$$a \rightarrow a$$

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

$$Id : (Int \rightarrow b) \rightarrow (Int \rightarrow b)$$

Then we write the "code" for Id, that is:

$$Id \ x = x$$

What the interpreter does is: finds the principal type of the expression associated to Id and then checks if the specification we gave, $(Int \rightarrow b) \rightarrow (Int \rightarrow b)$, 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 for a enabling the unify algorithm to terminate successfully.

Definition (**Unify and UnifyBases**)

Let **B** the collection of all bases, **S** be the set of all substitutions, and Id_S be the substitution that replaces all type-variables by themselves.

i) *Robinson's unification algorithm* . Unification of Curry types is defined by:

    unify :: T x T → **S**
    unify φ τ = (φ ↦ τ), if φ does not occur in τ
    unify σ φ = unify φ σ
    unify (σ → τ) (ρ → μ) = S_2oS_1,

    where S_1 = *unify* σ ρ and S_2 = *unify* $(S_1 \ τ) \ (S_1 \ μ)$

ii) By defining the operation *UnifyBases*, the operation *unify* can be generalized to bases:

UnifyBases :: **B** x **B** $\longrightarrow$ **S**
UnifyBases B_0, x**:**σ B_1, x**:**τ = S_2oS_1
UnifyBases B_0, x**:**σ B_1 = *UnifyBases* B_0 B_1, if x does not occur in B_1.
UnifyBases Ø B_1 = Id_s.

where S_1 = *unify* σ τ and S_2 = *UnifyBases* (S_1 B_0) (S_1 B_1)

Notice that *unify* can fail only in the second alternative, when ϕ occurs in μ.