Introduction to the notation of λ-calculus (Lecture Notes)

Gérard Boudol, INRIA, Sophia-Antipolis

(Translated and typeset by Madhavan Mukund)

1 Functional expressions: Application

The λ-calculus is a *calculus of functions* (the main goal of this course is to justify this assertion), and, moreover, a *notation* for functions.

What is a function? In set theory, it is a *graph*—that is, a (total) functional part of a Cartesian product. The notation

$$f : A \to B$$

or $f \in B^A$ stands for the fact that $f \subseteq A \times B$ and $\forall x \exists! y.\ (x, y) \in f$. From the point of view of programming, or more generally of computation, this definition does not say much. More interesting is the notion of function in the usual sense of mathematics—a *computational procedure* which permits us to calculate an answer when provided with some arguments. For example, we write

$$f : x \mapsto (x^2 + x)^2$$

or, more generally,

$$f : x \mapsto E \text{ or, for instance, } f(x) = E$$

where E is an *expression* consisting of some operations, some constants and some variables (on which the function depends — x in our example).

This notion of function is vague: to make it more precise, we need to specify what we mean by expressions—what are the "basic building blocks" from which we construct expressions and what are the rules for putting them together?

We will introduce the notation of λ-calculus by addressing the following question: how can we write expressions which describe computational procedures in a suitably general manner?

At first sight, there do not appear to be any general rules — what is common to the following expressions?

$$\frac{\sqrt{1-x^2}}{x^3+1} \qquad \int_0^{2\pi}\int_0^{\infty} e^{-r^2} r \; dr \; d\theta \qquad \begin{pmatrix} x-\lambda & 1 \\ 0 & x-\lambda \end{pmatrix}$$

or, for instance,

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

In fact, the set theoretic definition of a function suggests a uniform way of constructing expressions, by applying a function f to its arguments $x_1, x_2, \ldots, x_k$, written as $f(x_1, x_2, \ldots, x_k)$. For example, the expressions $(x^2 + x)^2$ should be written thus, using the functions $\mathsf{plus}(x, y) = x + y$ and $\mathsf{exp}(x, y) = y^x$:

$$\mathsf{exp}(2, \mathsf{plus}(\mathsf{exp}(2, x), x))$$

If we assume that we know the number of arguments required by the basic functions, we can get rid of parentheses and commas and write:

$$\mathsf{exp}\ 2\ \mathsf{plus}\ \mathsf{exp}\ 2\ x\ x$$

The reason why we do not systematically use this notation is that it rapidly becomes unreadable—except by a machine. In addition, in some cases this notation can be ambiguous: if f and g are two functions of one argument, fgx can be read as either $f(g(x))$ if $g : A \to B$ and $f : B \to C$, or as $f(g)(x)$ if $g : A \to B$ and $f : B^A \to D^C$. We do not take into account the domains of definition and the values of functions, because we want to deal with functions which can be understood without referring to their arguments, like the identity function $\iota(x) = x$, or the constant function $k_a(x) = a$ associated with an object a.

An early step towards the notation of λ-calculus was taken by Schönfinkel (1924): *"we denote the value of a function f for the argument x by simple juxtaposition of the signs for the function and the argument, that is fx"*. Note that we immediately have to take some precautions with this notation: if we wish to write correctly the function which we normally denote by $f(g(x))$, we are forced to use several parentheses. The notation for applying a function f to its argument x is formally (fx), but we are allowed to omit superfluous external parentheses. This permits us to denote the preceding expression by $f(gx)$.

Schönfinkel further noted that we can use this notation for functions of more than one argument. Thus, $f(x_1, x_2, \ldots, x_k)$ becomes

$$(\cdots((fx_1)x_2)\cdots x_k).$$

Since this notation is rather heavy, Schönfinkel proposed a convention by which $(\cdots((fx_1)x_2)\cdots x_k)$ is abridged as

$$fx_1x_2\cdots x_k.$$

For instance, we write $\mathsf{plus}\ x\ y$ instead of the more formal $(\mathsf{plus}\ x)y$ for the function $\mathsf{plus}(x, y)$. This means that we consider the function plus of two variables as a function of one variable, whose output is itself a function of one variable. For example, $\mathsf{plus}\ 2$ is the function which adds 2, which becomes clear if we write $(\mathsf{plus}\ 2)n = 2 + n$. This is a general phenomenon: any function

$$f : A \times B \to C$$

can be transformed into a function

$$f^* : A \to C^B$$

where, for each $a \in A$, the function $f^*(a)$ is defined (using normal mathematical notation this time) as $f^*(a)(b) = f(a, b)$.

Note that by writing $(fx)y$ the notion of "number of arguments" of a function disappears—more precisely, *every* function is a function of one argument. Moreover, as we have seen, a function can very well be an argument to another function. For example, in accordance with the "uniform" functional notation that we have adopted, we should denote the transformation $f \mapsto f^*$ by the form $(*f)$. (This transformation is know as "currying" but is in fact due to Schönfinkel and was actually known even earlier, to Frege, for example.)

Recall that a function which takes other functions as arguments, like $*$, is called a *functional.* The λ-calculus notation (fx) has the advantage of making more "symmetric" the role of the function and the argument, and thus removing the distinction, to some extent, between the two.

It is time to define more formally the syntax which we have discussed for functional expressions. Assume that we have an arbitrarily large set $\mathcal{X}$ of *variables*, whose elements are denoted by lower case letters such as f, g, $\ldots$, x, y, z, $\ldots$and a set $\mathcal{F}$ of functional symbols (among which are, for example, plus and exp). The functional expressions (or, more precisely, the *applicative expressions*) which we consider are generated by the following grammar:

$$E ::= x \mid \mathsf{f} \mid (EE)$$

where x is an arbitrary variable and f is an arbitrary functional symbol. In other words, applicative expressions are the sequences of symbols obtained from the following rules:

(i) Every variable x and every functional symbol f is an applicative expression.

(ii) If E_0 and E_1 are applicative expressions, so is (E_0E_1), the *application* of E_0 to E_1.

Recall the conventions adopted so far: we can omit external parentheses in an expression—that is, we can write E_0E_1 for (E_0E_1). Further, $E_0E_1E_2 \cdots E_k$ should be understood as $(\cdots((E_0E_1)E_2)\cdots E_k)$. For instance, $\mathsf{exp}(\mathsf{plus}\ 2)$ and $(\mathsf{exp}\ \mathsf{plus})2$, as also $\mathsf{plus}\ xyz$ and $2(\mathsf{plus}x)$, are expressions.

Question: What is the meaning of these expressions?

This syntax for applicative expressions may appear rather weak (what interesting things can we do with these expressions?). It may even appear suspect (does the expression $2(\mathsf{plus}\ x)$ have any meaning?) In fact, if we are also given the power to *define* functions with these expressions, we find that we already have a very rich game on our hands. Suppose, for instance, that f and g are functions and x is a variable. We define the operation, or more precisely the functional, $\mathbf{B}$ by:

$$\mathbf{B}fgx =_{\mathrm{def}} f(gx)$$

It is clear that $\mathbf{B}fg$ is the *composition* of the functions f and g, normally denoted by $f \circ g$. Now, we can define the powers of a function f—that is

$$f^n = \underbrace{f \circ \cdots \circ f}_{n \text{ times}}$$

More formally, we begin with the identity function I, defined by $\mathsf{I}x =_{\text{def}} x$. We then have:

$$\begin{array}{rcl} f^0 & =_{\text{def}} & \mathsf{I} \\ f^{n+1} & =_{\text{def}} & \mathsf{B}ff^n \end{array}$$

We have, for example $f^0x = x$ and

$$f^{n+1}x = (\mathsf{B}ff^n)x = f(f^nx)$$

EXERCISE 1.1: *Using the preceding definition for f^n, show that*

$$f^nx = \underbrace{f(\cdots(f}_{n \text{ times}}x)\cdots)$$

From this, show that $f^n(f^mx) = f^{n+m}x$ and $(f^n)^mx = f^{nm}x$.

All this suggests that we can do some arithmetic, as follows: for each integer n, define a functional k_n which iterates n times the function which it is supplied as an argument: $\mathsf{k}_nfx = f^nx$. The definition of these functionals is as follows:

$$\begin{array}{rcl} \mathsf{k}_0fx & =_{\text{def}} & x \\ \mathsf{k}_{n+1}fx & =_{\text{def}} & \mathsf{B}f(\mathsf{k}_nf)x \end{array}$$

EXERCISE 1.2: *Verify that $\mathsf{k}_1fx = fx$, $\mathsf{k}_2fx = f(fx)$, etc.*

We can now give meaning to the problematic expression which we encountered earlier—$2(\mathsf{plus}\ x)$—if we understand this to be $\mathsf{k}_2(\mathsf{plus}\ x)$. In effect, we have

$$(\mathsf{k}_2(\mathsf{plus}\ x))y = (\mathsf{plus}\ x)((\mathsf{plus}\ x)y)$$

which we can read as $2x + y$. Regarding the expressions k_n as "the integers", we can ask if we know how to define "multiplication"—that is, a function(al) mult such that $\mathsf{mult}\ \mathsf{k}_n\mathsf{k}_m = \mathsf{k}_{nm}$. More precisely, we want

$$\mathsf{mult}\ \mathsf{k}_n\ \mathsf{k}_mfx = \mathsf{k}_{nm}fx$$

Keeping in mind the fact that $\mathsf{k}_nfx = f^nx$ and $(f^n)^mx = f^{nm}x$, whereby $\mathsf{k}_{nm}fx = (\mathsf{k}_m(\mathsf{k}_nf))x$, we should have

$$\mathsf{mult}\ \mathsf{k}_n\ \mathsf{k}_mfx = (\mathsf{k}_n(\mathsf{k}_mf))x$$

Replacing the parameters k_n and k_m by the variables p and q in this equation, we have:

$$\mathsf{mult}\ pqfx =_{\text{def}} q(pf)x$$

Furthermore

$$\begin{array}{rcl} \mathsf{mult}\ \mathsf{k}_n\mathsf{k}_mfx & = & (\mathsf{k}_m(\mathsf{k}_nf))x \\ & = & (\mathsf{k}_nf)^mx \\ & = & (f^n)^mx \\ & = & f^{nm}x = \mathsf{k}_{nm}fx \end{array}$$

Observe that in the reasoning we have followed, we have implicitly used two principles:

(i) When given a definition $X =_{\text{def}} E$, we can replace X by E in all contexts.

(ii) When we have $F = E$, we can replace F' by E' in all contexts, where F' and E' are expressions obtained by replacing in F and E respectively a variable x by an expression X.

Continuing with the idea of doing a little arithmetic, it is also easy to see how to define a functional "successor", such that $\mathsf{succ}\ \mathsf{k}_n = \mathsf{k}_{n+1}$—more precisely, $\mathsf{succ}\ \mathsf{k}_n fx = \mathsf{k}_{n+1} fx = f(\mathsf{k}_n fx)$. It is sufficient to write:

$$\mathsf{succ}\ pfx =_{\text{def}} f(pfx)$$

Similarly, since

$$\begin{aligned} \mathsf{k}_{n+m} fx &= f^{n+m} x \\ &= f^n(f^m x) = \mathsf{k}_n f(\mathsf{k}_m fx) \end{aligned}$$

we can, by replacing the parameters k_n and k_m with variables p and q in this equation, define "addition" by:

$$\mathsf{plus}\ pqfx =_{\text{def}} pf(qfx)$$

Exercise 1.3: *Verify that* $\mathsf{plus}\ \mathsf{k}_1 \mathsf{k}_n fx = \mathsf{succ}\ \mathsf{k}_n fx$.

Exponentiation is even easier: define $\mathsf{exp}\ pq = (pq)$ (by which we observe that application is not a "product" but actually an exponential).

We thus begin to see that the simple notion of applying one expression to another is a much richer concept than we could have ever imagined. Let us now look at another example: suppose we wish to define a "conditional branch" which allows function definitions like $\mathsf{if}\ B\ \mathsf{then}\ X\ \mathsf{else}\ Y$. We will write this as a function cond which takes three arguments, B, X and Y. We want this function to satisfy

$$\mathsf{cond}\ BXY = \begin{cases} X & \text{if } B \text{ is "true"} \\ Y & \text{if } B \text{ is "false"} \end{cases}$$

The idea is the following. We define

$$\mathsf{cond}\ bxy =_{\text{def}} bxy$$

so that it only remains to find representations for "true" and "false" such that "true" applied to X and then Y is X while "false" applied to X and then Y is Y. The solution is obvious: for "true" we write

$$\mathbf{T}xy =_{\text{def}} x$$

while

$$\mathbf{F}xy =_{\text{def}} y$$

It is conventional to use the symbol $\mathbf{K}$ rather than $\mathbf{T}$, because this symbol brings to mind the idea of a constant function, which does not depend on its argument—this becomes clearer if one writes the previous equation for $\mathbf{T}$ in the form $(\mathbf{K}a)x = a$, where $\mathbf{K}a$ is the constant function which always yields the value a.

Thus, the simple notion of expression constructed using variables and application is in fact very rich—one can, in fact, do away with functional symbols $\mathsf{f} \in \mathcal{F}$ and still have an interesting calculus. Often, we have also used the facility of *defining* the value of an expression in terms of the result of applying it to other expressions, like $\mathsf{B}\ fgx =_{\mathrm{def}} f(gx)$. Actually, this facility should not be used indiscriminately, as it can lead to paradoxes. For example, nothing prevents us from defining a functional φ which, applied to any function f, gives the following result:

$$\varphi f =_{\mathrm{def}} \begin{cases} \mathsf{k}_0 & \text{if } ff \neq \mathsf{k}_0 \\ \mathsf{k}_1 & \text{otherwise} \end{cases}$$

(This example is due to A. Meyer.) We thus have $\varphi f \neq ff$, and, in particular, $\varphi\varphi \neq \varphi\varphi$. We thus find ourselves forced to either abandon (or restrict) the ability to apply any expression to any other or to abandon (or restrict) the facility to define expressions in terms of functions of other expressions. We shall see that the real problem comes from the ability to write inequations such as $ff \neq \mathsf{k}_0$. In what follows, we shall restrict ourselves to the following method of defining functions: we can define a function f by one (and only one) equation

$$\mathsf{f}\ x_1 \cdots x_n =_{\mathrm{def}} E$$

where E is an applicative expression which does not contain any variables other than $x_1, \ldots, x_n$. This definition mechanism is just a means of abbreviation—in all contexts, we are allowed to replace $\mathsf{f}\ X_1 \cdots X_n$ by E', where E' is obtained from E by replacing each variable x_i by the corresponding argument X_i. The resulting calculus is essentially the *combinatory logic* of Curry.

2 Functional expressions: Abstraction

The second step towards the notation of λ-calculus consists of simplifying the definition mechanism which we have just described. To introduce this simplification, let us return to the notion of a function as a computational procedure. When we write $f : x \mapsto E$ or $f(x) = E$, it is understood that the expression E is a "function of x" such as, for example, $E = (x^2 + x)^2$. At the same time, the *formal* expressions which we have been writing till now were made up of variables and applications (and also eventually of predefined functional symbols) and we have assumed when writing $\mathsf{f}\ x_1 \cdots x_n =_{\mathrm{def}} E$ that the expression E is a "function of $x_1, \ldots, x_n$".

What Church proposed with the λ-calculus was to make explicit the fact that an expression E is a function of x. More precisely, the λ-calculus provides a notation for designating *the function of x represented by the expression E*—that is, precisely the function f in $f : x \mapsto E$. This notation is $\lambda x E$, which we are supposed to read as "the function of x represented by the expression E". This gives us a new way of forming expressions: if x is a variable and E an expression, then $\lambda x E$ is an expression, which we call the *abstraction* of x in E. For clarity, this is often written as $\lambda x.E$, though the dot is not necessary. This expression is also written sometimes as $(\lambda x M)$—for example, consider the expression $(\lambda x M)N$—because writing $\lambda x M N$ leads to some confusion. In a certain sense, which will be eventually be made formal, if $f = \lambda x E$, the application of f to x is just E—-that is, $(\lambda x E)x = E$.

This notation will allow us to greatly abdridge the definitions we have seen earlier. The principle is the following: the expression $Xx =_{\text{def}} E$ can be transformed into $X =_{\text{def}} \lambda x E$, using the idea that $fx = E$ if $f = \lambda x E$. For instance, the identity function, which was defined as $\mathsf{I}x =_{\text{def}} x$, can now be defined directly as:

$$\mathsf{I} =_{\text{def}} \lambda x x$$

This is the "function of x whose value is x". At the same time, the definition $\mathsf{B}fgx =_{\text{def}} f(gx)$ can be transformed into $\mathsf{B}fg =_{\text{def}} \lambda x.f(gx)$. But nothing forces us to stop here: we can continue in this vein and obtain $\mathsf{B}f =_{\text{def}} \lambda g \lambda x.f(gx)$. Eventually, we obtain:

$$\mathsf{B} =_{\text{def}} \lambda f \lambda g \lambda x.f(gx)$$

Here, again, we can read B as the "function of f, g and x whose value is $f(gx)$". In the same way, for the "integers":

$$\begin{aligned} \mathsf{k}_0 &=_{\text{def}} \lambda f \lambda x.x \\ \mathsf{k}_{n+1} &=_{\text{def}} \lambda f \lambda x.\mathsf{B}f(\mathsf{k}_n f)x \end{aligned}$$

Let us introduce an abbreviation to economize on symbols: a sequence of abstractions such as $\lambda x_1 \lambda x_2 \cdots \lambda x_k E$ will be denoted by $\lambda x_1 x_2 \ldots x_k.E$, and thus a defining equation of the form $\mathsf{f}x_1 \cdots x_n =_{\text{def}} E$ can be transformed into $\mathsf{f} =_{\text{def}} \lambda x_1 \cdots x_n.E$. Thus, we can redefine the functions we have defined earlier on the "integers" as follows:

$$\begin{aligned} \mathsf{mult} &=_{\text{def}} \lambda pqfx.q(pf)x \\ \mathsf{succ} &=_{\text{def}} \lambda pfx.f(pfx) \\ \mathsf{plus} &=_{\text{def}} \lambda pqfx.pf(qfx) \\ \mathsf{exp} &=_{\text{def}} \lambda pq.(pq) \end{aligned}$$

and, in the same vein,

$$\begin{aligned} \mathsf{cond} &=_{\text{def}} \lambda bxy.bxy \\ \mathsf{K} &=_{\text{def}} \lambda xy.x (= \mathsf{T}) \\ \mathsf{F} &=_{\text{def}} \lambda xy.y \end{aligned}$$

The terms K and F are *selectors*, which select one or the other of their two arguments.

We can now officially introduce the syntax of the *pure λ-calculus*—pure in the sense that we dispense with predefined function symbols. The terms of the λ-calculus are given the following grammar:

$$M ::= x \mid \lambda x M \mid (MM)$$

Normally, λ-terms are denote by capital letters such as M, N, P, R, ...

The question which we have to address now is how to compute with our new expressions $\lambda x M$. For instance, how do we show that $\mathsf{I}X = X$ or that $\mathsf{B}FGX = F(GX)$? To approach this question, let us look one last time at the example $f(x) = (x^2 + x)^2$ and see how to calculate $f(2)$, for instance:

$$\begin{aligned} f(2) &= (2^2 + 2)^2 \\ &= (4 + 2)^2 \\ &= 6^2 = 36 \end{aligned}$$

If we examine the first step in the computation, we observer that computing $f(2)$ consists, in the first place, of replacing the formal parameter x by 2 in the expression of the function, here $(x^2 + x)^2$. This is what we will do in the λ-calculus as well: when we apply a function $f = \lambda x E$ to an argument X, we begin by substituting X for x in the expression E of the function. And, in truth, the λ-calculus "stops right there"—except for iterating this process. In other words, the only mechanism of calculation that we use will be that of replacing the application of a function to its argument—that is, a term of the form $(\lambda x M)N$—by the result of substituting the argument N for the parameter x in the expression M of the function. Thus, for instance:

$$\begin{array}{rcccl} \mathsf{I}\,X & =_{\text{def}} & (\lambda x.x)X & = & X \\ \mathsf{B}\,FGX & =_{\text{def}} & (\lambda fgx.f(gx))FGX & = & (\lambda gx.F(gx))GX \\ & & & = & (\lambda x.F(Gx))X \\ & & & = & F(GX) \end{array}$$

In the same way

$$\mathsf{k}_0 fx =_{\text{def}} (\lambda fx.x)fx = (\lambda x.x)x = x$$

Notice that we do not permit the replacement of a function symbol by its definition as a computation step. If we wish to compute an expression M which contains a functional f defined as $\mathsf{f} =_{\text{def}} N$, where N is a λ-term, we treat f as a variable and compute instead the value of $(\lambda\mathsf{f}.M)N$.

Let us denote by $M[N/x]$ the result of substituting N for x in M. The *"fundamental law"* of the λ-calculus is that we can replace, in all contexts, a term of the form $(\lambda x M)N$, which is called a *radical*, by $M[N/x]$. This is expressed by the equation

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

Note that a special case of this equation is $(\lambda x M)x = M$, which agrees with what we have proposed earlier. The congruence generated by this equation is called *β-conversion*—that is, the smallest equivalence relation which is compatible with the two construction rules for terms, application and abstraction, which contains (β). This congruence is denoted $M =_\beta N$. By definition, if $M =_\beta N$ we can replace M by N in any context. To be completely formal, we can say that β-conversion is characterized by the following properties:

$$\begin{array}{rcl} (\lambda x.M)N & =_\beta & M[N/x] \\ M & =_\beta & M \\ M =_\beta N & \Rightarrow & N =_\beta M \\ M =_\beta N \text{ and } N =_\beta R & \Rightarrow & M =_\beta R \\ M =_\beta N & \Rightarrow & \lambda x M =_\beta \lambda x N \\ M =_\beta M' \text{ and } N =_\beta N' & \Rightarrow & (MN) =_\beta (M'N') \end{array}$$

For instance, since, as we have seen, $\mathsf{k}_0\ fx =_\beta x$, we have:

$$\begin{array}{rcl} \mathsf{B}f(\mathsf{k}_0 f)x & =_\beta & f(\mathsf{k}_0 fx) \\ & =_\beta & fx \end{array}$$

From this, we can conclude that

$$\mathsf{k}_1 =_\beta \lambda fx.fx$$

In the same way, we compute k_2:

$$\begin{array}{rcl}\mathsf{k}_2 & = & \lambda fx.\mathbf{B}f(\mathsf{k}_1\ f)x \\ & =_\beta & \lambda fx.\mathbf{B}f((\lambda fx.fx)x \\ & =_\beta & \lambda fx.\mathbf{B}f(\lambda x.fx)x \\ & =_\beta & \lambda fx.f((\lambda x.fx)x) \\ & =_\beta & \lambda fx.f(fx)\end{array}$$

Since $\mathbf{B}f(\mathsf{k}_n f)x =_\beta f((\mathsf{k}_n f)x)$, we could have simplified the definition of k_{n+1} into $\mathsf{k}_{n+1} = \lambda fx.f(\mathsf{k}_n fx)$.

EXERCISE 2.1: *Show that for all n*

$$\mathsf{k}_n =_\beta \lambda fx.\ \underbrace{f(\cdots(f}_{n \text{ times}}\,x)\cdots)$$

Show that $\lambda fx.\mathsf{k}_n fx =_\beta \mathsf{k}_n$ for all n. Conclude that

$$\mathsf{mult}\ \mathsf{k}_n\ \mathsf{k}_1 =_\beta \mathsf{k}_n$$

Show that $\mathsf{succ}\ \mathsf{k}_1 =_\beta \mathsf{k}_2$*. Verify that* $\mathsf{plus}\ \mathsf{k}_1\ N =_\beta \mathsf{succ}\ N$*. Show that*

$$\mathsf{k}_n =_\beta \mathsf{k}_n\ \mathsf{succ}\ \mathsf{k}_0$$

for all n.

Observe that one consequence of this exercise is that:

$$\mathsf{k}_n =_\beta \underbrace{\mathsf{succ}(\cdots(\mathsf{succ}}_{n \text{ times}}\,\mathsf{k}_0)\cdots)$$

which is the usual representation of the integers. Often, in the λ-calculus, it is assumed that the integers are represented by terms of the form

$$\mathsf{c}_n =_{\text{def}} \lambda fx.\ \underbrace{f(\cdots(f}_{n \text{ times}}\,x)\cdots)$$

These are the *"Church integers"*. We shall see that $\mathsf{c}_n =_\beta \mathsf{k}_n$. The reason for preferring the Church integers to k_n is the following: the terms c_n are in *normal form* in the sense that they do not contain any subterms of the form $(\lambda xM)N$. In other words, there is no more computation left to be done in the c_ns. In the same way, it can be verified that all the terms that we have written for arithmetic operations, as well as the terms cond, $\mathbf{K}$ and $\mathbf{F}$, are in normal form and one can compute, for instance, that $\mathsf{cond}\ \mathbf{K} =_\beta \mathbf{K}$ and $\mathsf{cond}\ \mathbf{F} =_\beta \mathbf{F}$.

This suggests that to calculate means to find the normal form of a term, which can be treated as the result of the calculation. Unfortunately, this idea cannot be pushed very far. First of all, we have spoken of "the" normal form of a term, but it is not obvious that this is well-defined. For instance, when calculating k_2 earlier, we could have written $\mathsf{k}_2 =_\beta \lambda fx.f(\mathsf{k}_1 fx)$ rather than $\mathsf{k}_2 =_\beta \lambda fx.\mathbf{B}f((\lambda fx.fx)f)x$. We can verify that if we follow the computation further, we again reach the term $\lambda fx.f(fx)$. This is a general result—we shall see that the normal form of a term, if it exists, is unique. A second question is to know if the normal form of a term always exists. We shall see that this is not the case.

To see this, let us return for a moment to the definition mechanism which we have adopted. When we write $\mathsf{f}x_1 \cdots x_n =_{\mathrm{def}} E$, nothing prevents the expression E from containing the symbol f. In such a case, we have a *recursive definition* for the function f. Can we still represent such a function f by a λ-term? Clearly, it is not sufficient to write $\mathsf{f} =_{\mathrm{def}} \lambda x_1 \dots x_n.E$, as this will force us to expand f on the right hand side, leading to an infinitely long expression. To see how to do this, let us write the expression E, which contains f, in the form $E = [\cdots \mathsf{f} \cdots]$ (the symbol f may appear more than once in E). The "trick", to find a λ-term which satisfies the equation $\mathsf{f}x_1 \cdots x_n =_{\mathrm{def}} E$, is to consider the expression E^* obtained by replacing f in E by ff, where f is a variable—that is, $E* = [\cdots ff \cdots]$. Now, write

$$F =_{\mathrm{def}} \lambda f \lambda x_1 \dots x_n.E* = \lambda f \lambda x_1 \dots x_n.[\cdots ff \cdots]$$

Then it is easy to see that

$$FF =_\beta \lambda x_1 \dots x_n [\cdots FF \cdots]$$

which means that FF satisfies the defining equation for f. We can now write

$$f =_{\mathrm{def}} FF \text{ where } F = \lambda f \lambda x_1 \dots x_n.E^*$$

For instance, we can use this "trick" (which permits us to rephrase (simple) recursive functional equations in terms of the λ-calculus) in order to define the factorial function, which is given by:

$$\mathsf{fact}\ x =_{\mathrm{def}} (\mathsf{if}\ x = \mathsf{k}_0\ \mathsf{then}\ \mathsf{k}_1\ \mathsf{else}\ \mathsf{mult}\ x(\mathsf{fact}\ (\mathsf{pred}\ x)))$$

Actually, we need to know how to define a "test for 0" (which is part of a later exercise), and also how to define the predecessor function pred in order to define the factorial function. Let us look at another, simpler, example which uses the previous technique: suppose that we wish to define a functional f which satisfies the equation

$$\mathsf{f}x =_{\mathrm{def}} x(\mathsf{f}x)$$

Since here the expression E^* is $x(ffx)$, we obtain $F = \lambda fx.x(ffx)$, and it is sufficient to write

$$\mathsf{f} =_{\mathrm{def}} FF = (\lambda fx.x(ffx))(\lambda fx.x(ffx))$$

It is easy to verify that $\mathsf{f} =_\beta \lambda x.x(\mathsf{f}x)$, and thus $\mathsf{f}X =_\beta X(\mathsf{f}X)$ for all X. This functional is known as the *fixed point operator*, because the definition of a fixed point of a function f is an operator p such that $f(\mathsf{p}) = \mathsf{p}$. This fixed point operator, due to Turing, is traditionally denoted Θ.

We have shown that all λ-terms M, regarded as functions of one argument, have a fixed point ΘM. This can appear surprising. For instance, what could possibly be the fixed point of the successor function? Let us calculate:

$$\begin{array}{rcl}\Theta\ \mathsf{succ} & = & \mathsf{succ}(\Theta\ \mathsf{succ}) \\ & =_\beta & \lambda fx.f(\Theta\ \mathsf{succ} fx) \\ & =_\beta & \lambda fx.f(\mathsf{succ}(\Theta\ \mathsf{succ})fx) \\ & =_\beta & \lambda fx.f((\lambda fx.f(\Theta\ \mathsf{succ} fx))fx) \\ & =_\beta & \lambda fx.f(f(\Theta\ \mathsf{succ} fx)) \\ & \vdots & \end{array}$$

We see that the computation continues indefinitely, and even if the (infinite) "term" which appears

$$\mathsf{c}_\infty = \lambda fx.f(f(\cdots f(f(\cdots$$

seems to signify an infinite number, we have to conclude that the term Θ succ does not have a normal form, or, alternatively, that computing this term does not yield a result. Another example which is much simpler is the fixed point of the identity function $\Theta\ \mathbf{I}$. There again, if we try to compute this term, we find that the computation does not terminate, because all that we can say is that $\Theta\ \mathbf{I} =_\beta \mathbf{I}(\Theta\ \mathbf{I}) =_\beta \Theta\ \mathbf{I}$. Despite this, the situation is different, because while Θ succ appears to have some meaning, there appears to be no way to give a sensible interpretation to $\Theta\ \mathbf{I}$ (observe that any term is a fixed point of the identity). Yet another example: When using our technique for rephrasing recursive equations, it is possible to rephrase an equation which appears absurd—the equation which defines the constant f by $\mathsf{f} =_{\text{def}} \mathsf{f}$. It suffices to write $\Delta =_{\text{def}} \lambda f.ff$, and the solution is $\Delta\Delta$. The term Δ is the duplication operator, or *duplicator*, and the term $\Delta\Delta$ is normally denoted Ω. All that we can say of this term is that $\Omega =_\beta \Omega$.

It could be tempting to declare that all terms whose computation does not terminate—that is, all terms which do not have a normal form—are equivalent or identical in a certain sense: they all represent "undefined" terms. We have already noted, with the example of Θ succ (as compared to Ω in the preceding example) that this is not the best thing to do. In fact, this point of view is inconsistent. To see, this, let us formalize the preceding idea in the following way: suppose that the "semantics" of the λ-calculus is a congruence $M \approx N$ which identifies all "undefined" terms, and which contains the "fundamental law", and thus the computation relation, namely β-conversion $=_\beta$. It turns out that such a "semantics" identifies *all* terms—$X \approx Y$ for all X and Y. Consider the function f defined by the equation

$$\mathsf{f}xb =_{\text{def}} \mathsf{cond}\ bx(\mathsf{f}xb)$$

This function can be defined in the λ-calculus by

$$\mathsf{f} =_{\text{def}} FF \text{ where } F = \lambda fxb.\mathsf{cond}\ bx(ffxb)$$

To see its significance, let us compute $\mathsf{f}X\mathbf{K}$ and $\mathsf{f}X\mathbf{F}$.

$$\begin{array}{rcl}\mathsf{f}X\mathbf{K} & =_\beta & (\lambda xb.\mathsf{cond}\ bx(\mathsf{f}xb))X\mathbf{K} \\ & =_\beta & \mathsf{cond}\ \mathbf{K}X(\mathsf{f}X\mathbf{K}) \\ & =_\beta & X \end{array}$$

and

$$\begin{aligned} \mathsf{f}X\mathbf{F} &=_\beta (\lambda xb.\mathsf{cond}\ bx(\mathsf{f}xb))X\mathbf{F} \\ &=_\beta \mathsf{cond}\ \mathbf{F}X(\mathsf{f}X\mathbf{F}) \\ p &=_\beta \mathsf{f}X\mathbf{F} \end{aligned}$$

In a certain sense, we can say that f is the function $\lambda xb.(\mathsf{if}\ b\ \mathsf{then}\ x)$, which gives its first argument if its second one is "true" and which gives nothing if its second is "false". Another way of saying this is that f has the same meaning as $\lambda xb.\mathsf{cond}\ bx\Omega =_\beta \lambda xb.bx\Omega$ (a similar, but more interesting, example is dealt with in an exercise below). Now, suppose that $\approx$ is a "semantics" in the sense which we have defined. We then have:

$$\begin{aligned} fZ &=_\beta (\lambda xb.\mathsf{cond} bx(\mathsf{f}xb))Z \\ &=_\beta \lambda b.\mathsf{cond} bZ(\mathsf{f}Zb) \\ &=_\beta \lambda b.\mathsf{cond} bZG,\ \text{where } G = \mathsf{cond}\ bZ(\mathsf{f}Zb) \\ &=_\beta \lambda b.\mathsf{cond}\ bZ(\mathsf{cond}\ bZG) \\ &\ \ \vdots \end{aligned}$$

Since the computation of $\mathsf{f}Z$ does not terminate for any Z, we are forced to write $\mathsf{f}X \approx \mathsf{f}Y$ for all X and Y. And, since $\mathsf{f}X \approx \mathsf{f}Y \Rightarrow \mathsf{f}XM \approx \mathsf{f}YM$ for all M, we then have $\mathsf{f}X\mathbf{K} \approx \mathsf{f}Y\mathbf{K}$ for all X and all Y. But then $X \approx Y$, since $\mathsf{f}Z\mathbf{K} \approx Z$ for all Z. What we have shown is summarized in the following proposition.

Proposition: *Let $\approx$ be an equivalence relation on λ-terms with the following properties:*

(i) $\approx$ satisfies the fundamental law—that is, $(\lambda xM)N \approx M[N/x]$.

(ii) $\approx$ identifies all terms which do not have a normal form—that is, if M and N do not have normal forms, then $M \approx N$.

(iii) Functions which are equivalent under $\approx$ yield equivalent results when applied to the same argument—that is, $M \approx N \Rightarrow (\forall R)MR \approx NR$.

Then, $\approx$ is inconsistent: $X \approx Y$ for all X and Y.

We see that the notion of a normal form cannot really take the place of the notion of a result and serve as the foundation of a coherent semantics for the λ-calculus—which must satisfy, at the very least, the conditions we have demanded of $\approx$. One reason is that if we consider all terms without normal forms to be "undefined" and equal, then an "undefined" function can well give a well-defined result when applied to certain arguments, as we have seen with $\lambda xb.\mathsf{cond}\ bx\Omega$, and this is inconsistent. The core of this course will be to study the following question: what is a coherent semantics for the λ-calculus? Can we give meaning to "infinite objects" which appear to make sense, such as $\Theta\ \mathsf{succ}$, and also give a coherent status to undefined objects, the prototypical one being Ω? Can we give meaning to functions like $\lambda xb.\mathsf{cond}\ bx\Omega$, whose only defect is that they do not have a normal form? Note that β-conversion is not the solution, because we would like to identify, for instance, Ω and $\Theta\ \mathbf{I}$, while $\Omega \neq_\beta \Theta\mathbf{I}$.

Exercise 2.2: *We define the* pair *formed by two terms M and N as follows:*

$$\langle M, N\rangle =_{\text{def}} \lambda x.xMN$$

(i) *Define a combinator* P *such that* $\mathsf{P}MN =_\beta \langle M, N\rangle$ *for all* M *and* N.

(ii) *Define projection combinators* proj_0 *and* proj_1 *such that* $\mathsf{proj}_i\langle M_0, M_1\rangle =_\beta M_i$. *Hint: The usual projection functions are functions* $\pi_i : A_0 \times A_1 \to A_i$ *such that* $\pi_i(x_0, x_1) = x_i$. *To find* proj_0 *and* proj_1, *recall that one can transform* π_i *into* $\pi_i^* : A_0 \to (A_i)^{A_1}$.

(iii) *The currying of a function* $f : A \times B \to C$ *is the function* $f^* : A \to C^B$ *such that* $f^*(x)(y) = f(x, y)$. *Find an expression for this functional—that is, a combinator* $*$ *such that* $(*F)MN =_\beta F\langle M, N\rangle$ *for all* F, M *and* N.

Exercise 2.3: *Write a* λ*-term* D *which represents logical disjunction—that is, which satisfies* $\mathsf{DT}X =_\beta \mathsf{T}$ *and* $\mathsf{DF}X =_\beta X$. *Hint: Experiment first with the disjunction of the form "if . . . then . . . else".*

Write a λ*-term for conjunction* C *which satisfies* $\mathsf{CT}X =_\beta X$ *and* $\mathsf{CF}X =_\beta \mathsf{F}$.

Exercise 2.4: *Define a "test for 0"—that is, an expression* Z *such that* $\mathsf{Z}\ \mathsf{k}_n XY =_\beta X$ *if* $n = 0$ *and* $\mathsf{Z}\ \mathsf{k}_n XY =_\beta Y$ *otherwise. Hint: Set* $\mathsf{Z} = \lambda zxy.zEF$. *To find the appropriate expressions* E *and* F, *notice that* k_0 *is, with a change of variable, the same object as the selector* F, *and that* $\mathsf{k}_{n+1} =_\beta \lambda fx.fN$ *where* $N = \mathsf{k}_n fx$.

Exercise 2.5: *Compute* $\mathsf{K}^n f x_1 \cdots x_n$ *and* $\mathsf{B}^n fgx_1 \cdots x_n$. *Define*

$$\Phi =_{\text{def}} \lambda fghx.f(gx)(hx)$$

Compute $\Phi^n fghx_1 \cdots x_n$.

Exercise 2.6: *Compute* $\Theta\mathsf{F}$. *Write a* λ*-term* Ξ *which satifies the equation* $\Xi x = \Xi$. *Calculate* $(\Theta\mathsf{K})X_1 \cdots X_n$. *Find a* λ*-term for the function* f *defined by the equation*

$$\mathsf{f}xb =_{\text{def}} \mathsf{cond}\ bx(\lambda b.\mathsf{f}\ xb)$$

Compute $\mathsf{f}X\mathsf{K}$, $\mathsf{f}X\mathsf{FK}$ *and* $\mathsf{f}X\mathsf{FFK}$.