PFPL Supplement: Church's λ-Calculus*

Robert Harper

Fall, 2025

1 Introduction

The λ-calculus is a remarkably simple and elegant model of computation formulated by Church in the early 1930's that is based on the concept of a *mathematical variable*, which is given meaning by *substitution*. Think back to when you first learned about polynomials, plugging in values for variables and simplifying them by equational deduction. What a pleasure that was! The λ-calculus has the same flavor, except that, rather than standing for real numbers, the variables in a λ-term stand for other λ-terms. And just as polynomials may be thought of as functions of their free variables, so λ-terms may be thought of as functions acting on λ-terms, one of which is the λ-term itself—the λ-calculus is inherently self-referential, which is the source of its expressive power.

2 λ-Terms

Expressions of the λ-calculus are called λ-*terms*. They have one of three forms:

1. A *variable*, x.
2. An *application*, $\mathtt{ap}(M_1\,;M_2)$, of λ-term M_1 to another λ-term M_2.
3. An *abstraction*, $\lambda(x.M)$, of a variable x within the λ-term M.

For the time being, it is best not to try to impose *meaning* on λ-terms, but rather just to concentrate on their *definition* as pieces of syntax. However, the idea is that the λ-calculus is a system of functions that may be applied to one another as arguments. In particular the λ-term $\lambda(x.M)$ defines a function with argument x and result given by the λ-term M, which may use x. The bewildering fact is that the λ-calculus has *no constants, no numbers, no primitive operations*, but *only* functions. Yet, amazingly, data structures such as numbers, lists, trees, and tuples are all *implicitly present* in that they are all definable as certain functions!

The description of what are the λ-terms is "self-referential" in that the second and third clauses describe terms that may be constructed from other terms. This description may be regarded as an *inductive definition* of the λ-terms in which variables are base cases given outright, and both application and abstraction are inductive cases that make use of "previously" given λ-terms. The

inductive definition of λ-terms is similar to that of natural numbers, which are defined by saying that 0 is a natural number, and that if n is a natural number, then so is $n+1$. Or, similarly, inductively defining a tree to be either empty, or, if t_1 and t_2 are trees, then so is $\mathsf{node}(\, t_1, t_2 \,)$.

The standard format for formulating such inductive definitions is by a collection of *rules* for defining one or more *judgments*, or *assertions*. In the present case the judgment $M\,\mathsf{tm}$ states that M is a λ-term. It is inductively defined by the following rules:

$$\frac{x\,\mathsf{var}}{x\,\mathsf{tm}} \tag{1a}$$

$$\frac{M_1\,\mathsf{tm} \quad M_2\,\mathsf{tm}}{\mathsf{ap}(M_1\,;M_2)\,\mathsf{tm}} \tag{1b}$$

$$\frac{x\,\mathsf{var} \quad M\,\mathsf{tm}}{\lambda(x.M)\,\mathsf{tm}} \tag{1c}$$

The definition makes use of an assumed judgment $x\,\mathsf{var}$ stating that x is a variable. It is essential that there be *infinitely many* variables, written x, y, z, x_1, x_2, x', and similarly.

To say that the judgment $M\,\mathsf{tm}$ is inductively defined by these rules means that the $M\,\mathsf{tm}$ is the *strongest* assertion that is *closed under*, or *obeys*, the rules given above. This means that $M\,\mathsf{tm}$ obeys the rules, and it implies any other assertion that also obeys the rules. Thus, the rules may be read as implications stating that "*if* the judgments above the line hold, *then* the judgment below the line also holds." For $M\,\mathsf{tm}$ to be the strongest judgment obeying the rules means that "*if* some assertion $\mathcal{P}$ obeys the rules, *then* $M\,\mathsf{tm}$ implies that $\mathcal{P}$ holds of M." For $\mathcal{P}$ to obey the rules means that (1) it holds for every variable x; (2) if it holds for M_1 and M_2, then it holds for $\mathsf{ap}(M_1\,;M_2)$; and (3) if it holds for M and x is a variable, then it holds for $\lambda(x.M)$. This is called the principle of *rule induction*.

For example, define the judgment $\mathsf{size}(\, M, n \,)$, with the meaning that the size of the λ-term M is the natural number n, by the following rules:

$$\frac{x\,\mathsf{var}}{\mathsf{size}(\, x, 1 \,)} \tag{2a}$$

$$\frac{\mathsf{size}(\, M_1, n_1 \,) \quad \mathsf{size}(\, M_2, n_2 \,)}{\mathsf{size}(\, \mathsf{ap}(M_1\,;M_2), n_1+n_2+1 \,)} \tag{2b}$$

$$\frac{x\,\mathsf{var} \quad \mathsf{size}(\, M, n \,)}{\mathsf{size}(\, \lambda(x.M), n+1 \,)} \tag{2c}$$

In fact *every λ-term has a size*: *if $M\,\mathsf{tm}$, then there exists a natural number n such that $\mathsf{size}(\, M, n \,)$.* Let $\mathcal{P}(M)$ be the property *there exists an n such that $\mathsf{size}(\, M, n \,)$.* It suffices to show that $\mathcal{P}$ is closed under the rules defining $M\,\mathsf{tm}$:

1. If M is x where $x\,\mathsf{var}$, then take n to be 1 and note that $\mathsf{size}(\, x, 1 \,)$ by rule (2a).

2. If M is $\mathsf{ap}(M_1\,;M_2)$ where $M_1\,\mathsf{tm}$ and $M_2\,\mathsf{tm}$, then by induction there exists n_1 and n_2 such that $\mathsf{size}(\, M_1, n_1 \,)$ and $\mathsf{size}(\, M_2, n_2 \,)$. Let $n = n_1 + n_2 + 1$, and apply rule (2b).

3. If M is $\lambda(x.M_1)$, where $x\,\mathsf{var}$ and $M_1\,\mathsf{tm}$, then by induction there exists n_1 such that $\mathsf{size}(\, M_1, n_1 \,)$, and take n to be $n_1 + 1$, and apply rule (2c).

Furthermore, *the size of a λ-term is unique*, meaning that *if M tm, $\mathsf{size}(M, n_1)$, and $\mathsf{size}(M, n_2)$, then $n_1 = n_2$*. Consequently, *the size of a λ-term M* is well-defined.

Exercise 2.1. *1. Prove by rule induction that the size of a λ-terms is uniquely defined by rules (2).*

2. Give an inductive definition of the depth *of a λ-term to be the maximal nesting of λ-abstractions within it. Prove by rule induction that every λ-term has a unique depth.*

An abbreviated notation, called a *grammar*, is useful for defining the abstract syntax of a language. The grammar for λ-terms is as follows:

$$M ::= x \mid \mathtt{ap}(M_1 ; M_2) \mid \lambda(x.M)$$

It says, in words, a λ-term M is either a variable, x, or an application of two λ-terms M_1 and M_2 to each other, or an abstraction of a variable x in a λ-term M. Thus, the grammar is short-hand notation for rules (1). The principle of rule induction for a set of rules determined by a grammar is often called *structural induction*.

When writing examples it is helpful to use notational conventions that look better on the page or screen. For the λ-calculus these conventions are to write $M_1\, M_2$ for $\mathtt{ap}(M_1 ; M_2)$, and $\lambda x.M$ for $\lambda(x.M)$. When written in this form, application is *left-associative*; parentheses are used to override this convention when necessary. Moreover, applications have precedence over abstractions. Thus, one might write $(\lambda x.x\, x)\, (\lambda x.x\, x)$ instead of the official form, $\mathtt{ap}(\lambda(x.\, \mathtt{ap}(x ; x)) ; \lambda(x.\, \mathtt{ap}(x ; x)))$, and regard $M_1\, M_2\, M_3$ as short for $\mathtt{ap}(\mathtt{ap}(M_1 ; M_2) ; M_3)$.

3 Binding and Scope

The distinction between *free* and *bound* occurrences of variables in a λ-term is of the essence. You encountered this distinction already in differential calculus when you consider, for example, that $\partial/\partial x(x^2 + x\,y + y^2) = 2\,x + y$. Recall that the partial derivative with respect to a coordinate, or variable, x, is defined as the ordinary derivative of the function in which x varies while the other variables (y, in this case) are held fixed. The result is again a function in which y is held fixed, but again varying in the specified coordinate.

To be more explicit, write the above equation as

$$D(x \mapsto x^2 + x\,y + y^2) = x \mapsto 2\,x + y.$$

The idea is that the operator D takes a function as argument and yields a function as result, and the equation states that the two functions are equal (give the same result on all arguments). The variable y is fixed on both sides of the equation, which means that its *scope* (range of significance) lies outside of the derivative.

It is usual in informal mathematics to drop the "$x \mapsto$" notation when it is "clear from context" what is intended. But (a) it is sometimes less than clear from context, and (b) computers are terrible at understanding informal conventions. To make this precise, it is important to observe that in the mapping notation $x \mapsto x^2 + x\,y + y^2$ the variable x is *local*, or *bound*, within the mapping, whereas the variable y is *unbound*, or *free*, within the mapping, which is to say that it remains constant while x varies. Because the variable is local to the mapping, the choice of variable name does not change the meaning: $x \mapsto x^2 + x\,y + y^2$ could equally well have been written $z \mapsto z^2 + z\,y + y^2$. More

precisely, any choice of variable is as good as any other, *except for* y, because y is already in use: the function $y \mapsto y^2 + y\,y + y^2$, that is $y \mapsto 3\,y^2$, is not the same function as the others. Choosing y as the mapping parameter is said to *capture* the separate use of y in the body of the mapping.

Although it is not at all concerned with the real numbers, the λ-calculus shares with the differential calculus the need to carefully distinguish between free and bound variables. The notation $\lambda(x.M)$ is analogous to the notation $x \mapsto x^2 + x\,y + 1$ for mappings in that the variable x is *bound* within the λ-term M. There is no difference if x is replaced by any other variable, provided that it does not already occur in M. Unlike differential calculus, where variables range over real numbers, in λ-calculus variables range over λ-terms, and so a λ-term may be *substituted* for a free variable in another λ-term, much as numbers may be plugged in for variables in algebra.

And that, fundamentally, is all there is to it.

To make this precise the collection of λ-terms is inductively defined by a collection of rules that serve as a specification for an implementation on a computer. First, if x is a variable and M is a λ-term, the judgment $x \in \mathsf{FV}(M)$, which states that x occurs free in M, is inductively defined by the following rules:

$$\frac{}{x \in \mathsf{FV}(x)} \tag{3a}$$

$$\frac{x \in \mathsf{FV}(M_1)}{x \in \mathsf{FV}(\mathtt{ap}(M_1\,;M_2))} \tag{3b}$$

$$\frac{x \in \mathsf{FV}(M_2)}{x \in \mathsf{FV}(\mathtt{ap}(M_1\,;M_2))} \tag{3c}$$

$$\frac{x \neq y \quad x \in \mathsf{FV}(M)}{x \in \mathsf{FV}(\lambda(y.M))} \tag{3d}$$

There are *two* rules for application, which states that x is free in $\mathtt{ap}(M_1\,;M_2)$ iff it is free in *either* M_1 *or* M_2. The rule for abstractions demands that x be different from y. For example, x is not free in $\lambda(x.x)$, but is free in $\lambda(y.\,\mathtt{ap}(y\,;x))$.

Exercise 3.1. *1. Give a direct inductive definition of the judgment $x \notin \mathsf{FV}(M)$, stating positively that x is not free in M.*

2. Prove that if x var and M tm, then either $x \in \mathsf{FV}(M)$ or $x \notin \mathsf{FV}(M)$. Proceed by structural induction on M, making use of the fact that if x var and y var, then either $x = y$ or $x \neq y$.

4 Substitution

Substitution is defined by an inductive definition of the relation $\mathsf{subst}(\,M, x, N, P\,)$, which states that P is the result of substituting M for free x's in N. It is common to write this relation as $[M/x]N = P$, once it is clear that the four-place relation defines P as a (partial) function of M, x, and N.

$$\frac{}{\mathsf{subst}(\,M, x, x, M\,)} \tag{4a}$$

$$\frac{x \neq y}{\mathsf{subst}(\,M, x, y, y\,)} \tag{4b}$$

$$\frac{\mathsf{subst}(\, M, x, N_1, P_1\,) \quad \mathsf{subst}(\, M, x, N_2, P_2\,)}{\mathsf{subst}(\, M, x, \mathsf{ap}(N_1\,;N_2), \mathsf{ap}(P_1\,;P_2)\,)} \tag{4c}$$

$$\frac{}{\mathsf{subst}(\, M, x, \lambda(x.N), \lambda(x.N)\,)} \tag{4d}$$

$$\frac{x \neq y \quad y \notin \mathsf{FV}(M) \quad \mathsf{subst}(\, M, x, N, P\,)}{\mathsf{subst}(\, M, x, \lambda(y.N), \lambda(y.P)\,)} \tag{4e}$$

The last two rules, defining substitution into a λ-abstraction, are easy to get wrong. First off, if the λ-abstraction binds the variable x, then there cannot also be free x's within it, and so substitution has no effect. If the λ-abstraction binds a different variable y, substitution replaces free occurrrences of x within its body, and then λ-abstracts y over the result. This seems a plausible definition (just "plug in" M for x in the body of N and re-abstract), but it is nevertheless incorrect.

To see what is wrong, ignore for the moment the second premise of rule (4e), and derive

$$\mathsf{subst}(\, y, x, \lambda(y.\,\mathsf{ap}(y\,;x)), \lambda(y.\,\mathsf{ap}(y\,;y))\,).$$

But this is not correct! When substituting y for x in $\lambda(y.\,\mathsf{ap}(y\,;x))$, the bound variable y is local to the λ-abstraction, and should not be confused with the y being plugged in for x. Doing so is said to incur *capture*: the binding for y governs the y being plugged, changing its meaning. Thus, the purpose of the second premise of rule (4e) is to ensure that the incorrect result cannot be derived: substitution is *undefined* if capture would otherwise occur. Thus, in the foregoing example, there is no P such that $\mathsf{subst}(\, y, x, \lambda(y.\,\mathsf{ap}(y\,;x)), P\,)$, because the bound variable y of the abstraction occurs free substituting term, namely y itself. However, for any M, x, and N, there is *at most one* P such that $\mathsf{subst}(\, M, x, N, P\,)$; that is, the result of substitution is a *partial function* of its arguments.

Exercise 4.1. *Prove that if M tm, x var, and N tm, then if* subst$(\, M, x, N, P_1\,)$, *and* subst$(\, M, x, N, P_2\,)$, *then $P_1 = P_2$.*

5 α-Equivalence

Because the substitution judgment defines a partial function, it is tempting to write $[M/x]N$ for the unique P such that $\mathsf{subst}(\, M, x, N, P\,)$, *if it exists*, much as in math one may write $1/x$ for the unique y, *if it exists*, such that $x\,y = 1$. In a sense it is "cheating" to write $[M/x]N$, but, with care, nothing can go wrong, just as writing $1/x$ is ok as long as $x \neq 0$. Nevertheless, it would be better if substitution were *always* defined so that there are no worries, but can that be achieved?

Consider again the example in the last section for which substitution is undefined. Let N be the λ-term $\lambda(y.\,\mathsf{ap}(y\,;x))$ considered above, and let N' be the λ-term $\lambda(y'.\,\mathsf{ap}(y'\,;x))$, which differs from N only in that the bound variable y has been renamed to y', which has been chosen so that $y' \notin \mathsf{FV}(M)$. Now there *is* a (unique) P' such that $\mathsf{subst}(\, M, x, N', P'\,)$, even though "morally" there is no difference between N and N'. Notice that y is free in P', as it should be, because it is meaningful in the surrounding context of the substitution and should not be confused with a bound variable that would change its meaning. This shows that any undefinedness in substitution may always be *removed* by simply changing the names of the bound variables in N so as to avoid capture before it would occur.

Define the equivalence relation, $M \equiv_\alpha M'$, on λ-terms, called *α-equivalence*, that equates two terms that differ only in their bound variable names, as follows:

$$\frac{}{M \equiv_\alpha M} \tag{5a}$$

$$\frac{M \equiv_\alpha M'}{M' \equiv_\alpha M} \tag{5b}$$

$$\frac{M \equiv_\alpha M' \quad M' \equiv_\alpha M''}{M \equiv_\alpha M''} \tag{5c}$$

$$\frac{M_1 \equiv_\alpha M_1' \quad M_2 \equiv_\alpha M_2'}{\mathtt{ap}(M_1\,;M_2) \equiv_\alpha \mathtt{ap}(M_1'\,;M_2')} \tag{5d}$$

$$\frac{M \equiv_\alpha M'}{\lambda(x.M) \equiv_\alpha \lambda(x.M')} \tag{5e}$$

$$\frac{\mathsf{subst}(\,x', x, M, M'\,) \quad (x' \notin \mathsf{FV}(M))}{\lambda(x.M) \equiv_\alpha \lambda(x'.M')} \tag{5f}$$

Rule (5f) states that a bound variable x may be renamed to x' whenever doing so would not incur capture.

Exercise 5.1. *Show that substitution is well-defined up to α-equivalence: for any M and x and N, there exists N' such that $N' \equiv_\alpha N$ and $\mathsf{subst}(\,M, x, N', P'\,)$ for some P'; moreover, if $N'' \equiv_\alpha N'$ and $\mathsf{subst}(\,M, x, N'', P''\,)$, then $P'' \equiv_\alpha P'$.*

From now on λ-terms are *identified up to α-equivalence*, meaning that terms that differ only in the names of their bound variables are not distinguished from one another. Under this convention, substitution is always defined, and so it is appropriate to write $[M/x]N$ for the unique-up-to-α-equivalence result of substitution of M into *some* α-variant of N chosen to avoid capture of free variables in M. As was shown in the preceding exercise, the result is insensitive to the choice of α-variants of N, the result being the same up to α-equivalence.

This policy is convenient not only in informal work, but is also a boon to implementation. The convenience amounts to this:

> A bound variable is automatically "fresh" in the sense of being different from any other variable in any a given context.

By the miracle of α-equivalence, a bound variable is "never what one thinks it is." It is instead a "fresh" version that does not conflict with any other variable in use.

6 β-Equivalence: Calculation in λ-Calculus

The λ-calculus is so-called because it is a system of calculation, in fact a model of computation based on equations. In this regard it is similar to algebra, but quite far removed from machine models of computation, which emphasize the step-by-step process of computation. It is possible, and essential for implementation, to formulate a deterministic method of calculation, but for the present purposes it is enough to define the rules of equational reasoning in the λ-calculus.

The judgment $M \equiv_\beta M'$ states that the λ-terms M and M' are *β-equivalent* to one another using rules that are reminiscent of those you learned in algebra. This judgment is a form of equality for λ-terms that enjoys these properties:

1. It is an *equivalence relation*: it is reflexive, symmetric, and transitive.

2. It is *compatible* meaning that equals may be replaced by equals to get equals within any term.

3. It encodes *computation* by simplification of the application of an abstraction to an argument.

The first two properties are so familiar from algebra that they are often not even mentioned. The third is much the same: if I define a function of x, say $x \mapsto x^2 + 2\,x + 1$, then I can apply it to some number, say 2, by "plugging it in" for x to obtain $2^2 + 2\,2 + 1$, which is of course 9.

Here is an inductive definition of β-equivalence given by a set of rules:

$$\frac{}{M \equiv_\beta M} \tag{6a}$$

$$\frac{M \equiv_\beta M'}{M' \equiv_\beta M} \tag{6b}$$

$$\frac{M \equiv_\beta M' \quad M' \equiv_\beta M''}{M \equiv_\beta M''} \tag{6c}$$

$$\frac{M_1 \equiv_\beta M_1' \quad M_2 \equiv_\beta M_2'}{\mathtt{ap}(M_1\,;M_2) \equiv_\beta \mathtt{ap}(M_1'\,;M_2')} \tag{6d}$$

$$\frac{M \equiv_\beta M'}{\lambda(x.M) \equiv_\beta \lambda(x.M')} \tag{6e}$$

$$\frac{}{\mathtt{ap}(\lambda(x.N)\,;M) \equiv_\beta [M/x]N} \tag{6f}$$

Rules (6a), (6b), and (6c) specify that β-equivalence is reflexive, symmetric, and transitive. Rules (6d) and (6e) specify that it is compatible with application and abstraction. And rule (6f) defines how to apply a function to an argument: substitute the argument for the parameter in the body.

Exercise 6.1. *Define the following λ-terms, which are called* combinators*:*

$$\begin{aligned}\boldsymbol{I} &\triangleq \lambda x.x\\ \boldsymbol{K} &\triangleq \lambda x.\lambda y.x\\ \boldsymbol{S} &\triangleq \lambda x.\lambda y.\lambda z.(x\,z)\,(y\,z)\end{aligned}$$

The $\boldsymbol{I}$ combinator is the identity function, the $\boldsymbol{K}$ combinator generates konstant functions, and the $\boldsymbol{S}$ combinator is a "distributor" that "sends" z to both x and y, and then applies the results. Prove that $\boldsymbol{S}\,\boldsymbol{K}\,\boldsymbol{K} \equiv_\beta \boldsymbol{I}$ by exhibiting a derivation of it using the rules of β-equivalence.

The *β-reduction* judgment, $M \succ_\beta N$, is defined by the same rules as for β-equivalence, but without the rules of reflexivity or symmetry. Thus $M \succ_\beta N$ expresses that M simplifies to N by applying rule (6f) one or more times anywhere in M. Thus, for example,

$$\mathbf{K\,I\,I} \succ_\beta \mathbf{I\,I} \succ_\beta \mathbf{I},$$

but not the other way around.

A major result about the λ-calculus, called the *Church-Rosser Theorem*, states that $M \equiv_\beta M'$ iff M and M' have a common reduct: there exists P such that $M \succ_\beta P$ and $M' \succ_\beta P$. Thus, to determine whether $M \equiv_\beta M'$, it suffices to simplify both sides until a common reduct is reached.

Exercise 6.2. *A λ-term N is called a β-*normal form *iff it is β-irreducible in that there is no N' such that $N \succ_\beta N'$. Show that there is a λ-term M without a normal form, that is such that there is no N such that $M \succ_\beta N$ and $N \not\succ_\beta$.*

7 Church's Law

Unless you have prior experience with it, the λ-calculus must seem to be an arbitrary and mysterious formalism. In fact it was introduced by Church as a universal model of computation, which means that it is a fully general programming language. *Church's Law* is the the assertion that the λ-calculus can express *any* computable function on the natural numbers. Because it makes a prediction about the nature of such computable functions, namely that they are always programmable in the λ-calculus, Church's Law is a scientific law on par with Newton's Law relating forces to accelerations. And, like Newton's $F = m\,a$, it has been empirically verified so thoroughly that it is accepted as an eternal truth, within its range of applicability.

How is the λ-calculus supposed to compute on the natural numbers? Apparently, it doesn't even *have* natural numbers on which to compute! Appearances are deceiving, though: the natural numbers are *definable* as certain λ-terms. For the time being, assume given the following primitive constructs pertaining to the natural numbers:

1. $\mathbf{0}$, representing the number 0.
2. $\mathbf{succ}(M)$, representing the successor of the number given by M.
3. $\mathbf{ifz}(M, M_0, M_1)$, representing a case analysis on whether a natural number M is zero or a successor.

The computation steps corresponding to the case analysis are as follows:

$$\overline{\mathbf{ifz}(\mathbf{0}, M_0, M_1) \equiv_\beta M_0} \tag{7a}$$

$$\overline{\mathbf{ifz}(\mathbf{succ}(M), M_0, M_1) \equiv_\beta M_1\,M} \tag{7b}$$

Notice that in the second case, the argument M_1 is applied to the predecessor M.

Using these primitives it is almost possible to define, say, addition of natural numbers, except for the fact that there is apparently no way for a function to "call itself." In fact, there is a way,

using a programming trick, called the ***Y*** *combinator*, which was invented by Church and Kleene in the 1930's.

Begin by defining addition using the following recursion equations as a guide:

$$x + 0 = x$$
$$x + (y + 1) = (x + y) + 1$$

This suggests the following "definition," which refers to itself at the underlined spot:

$$\mathbf{add} \triangleq \lambda x.\lambda y.\mathbf{ifz}(y, x, \lambda z.\mathbf{succ}(\underline{\mathbf{add}}\, x\, z))$$

In programming languages such as ML this would be considered a valid definition, because self-reference is implicitly taken care of. But it is interesting to consider how to obtain **add** without assuming the problem has already been solved.

What is required is a *solution* to the recursion equation

$$\mathbf{add} \equiv_\beta \lambda x.\lambda y.\mathbf{ifz}(y, x, \lambda z.\mathbf{succ}(\underline{\mathbf{add}}\, x\, z)).$$

Kleene's invention was to obtain a solution in two steps. First, define a "prototype" of **add** that takes an additional argument, the function to be called in lieu of calling itself.

$$\mathbf{addproto} \triangleq \lambda f.\lambda x.\lambda y.\mathbf{ifz}(y, x, \lambda z.\mathbf{succ}(f\, x\, z))$$

The idea is that **add** will be defined to be a *fixed point* of **addproto**, which is to say that **add** is a solution to the equation

$$\mathbf{add} \equiv_\beta \mathbf{addproto}\, \mathbf{add}$$

Expanding the right-hand side yields the equation

$$\mathbf{add} \equiv_\beta \lambda x.\lambda y.\mathbf{ifz}(y, x, \lambda z.\mathbf{succ}(\mathbf{add}\, x\, z)).$$

This is self-referential in that the function **add** itself has been substituted for the variable f in the body of **addproto**.

But why should there be such a function **add** at all? That is, why should **addproto** have a fixed point? The second, and most intriguing, part of Kleene's invention is that the desired fixed point may be obtained using a technique called *self-application*. Define the auxiliary

$$\mathbf{addself} \triangleq \lambda \mathit{this}.\mathbf{addproto}\, (\mathit{this}\, \mathit{this}),$$

then define

$$\mathbf{add} \triangleq \mathbf{addself}\, \mathbf{addself},$$

which is the application of **addself** to itself. Calculate:

$$\begin{aligned} \mathbf{add} &\triangleq \mathbf{addself}\, \mathbf{addself} \\ &\equiv_\beta \mathbf{addproto}\, (\mathbf{addself}\, \mathbf{addself}) \\ &\equiv_\beta \mathbf{addproto}\, \mathbf{add} \end{aligned}$$

But this (ahem) is exactly what is wanted!

What is going on here? The main idea is adopt a programming convention for defining recursive functions. Specifically, to define a recursive function, give it an "extra" first argument with which it may refer to itself, then ensure that whenever that function is called, it is implicitly applied to the function itself everywhere it is used, including the internal call sites at which a function calls itself.

Exercise 7.1. *The foregoing calculation is so slick that it is easy to miss what is really going on here.*

1. *Write out explicitly what is* ***addself****, namely* $\lambda this. \ldots$*, where you are to fill in the dots by expanding* ***addproto*** *and reducing the resulting application.*
2. *Then verify that* ***addself addself*** *expands to the desired form, using self-application to effect the recursion.*

For the *coup de grace* abstract the entire discussion from addition, and define the ***Y*** *combinator* to be the following λ-term:

$$\mathbf{Y} \triangleq \lambda proto.(\underbrace{\lambda this.proto\,(this\,this)}_{M_{proto}})\,(\underbrace{\lambda this.proto\,(this\,this)}_{M_{proto}}).$$

Then define

$$\mathbf{add} \triangleq \mathbf{Y}\,\mathbf{addproto}$$

and check that

$$\mathbf{add} \equiv_\beta \mathbf{addproto}\,\mathbf{add}.$$

Exercise 7.2. *Define the function to add up the first n natural numbers.*

Now show that the natural numbers are definable as λ-terms in such a way that it is possible to program with case analysis. The original formulation was given by Church in the form of what are called the *Church numerals* whereby the natural number n is represented by the λ-term $\lambda b.\lambda s.s^{(n)}\,b$, the n-fold application of s, the inductive step, to b, the base case. Using this representation it is rather difficult, but possible, to define the case analysis primitive.[1] An alternative is to use the *Barendregt numerals*, named after the author of the definitive book on the λ-calculus.

The first step is to define the *Church booleans* using *binary decision diagrams* represented as follows:

$$\mathbf{T} \triangleq \lambda x.\lambda y.x$$
$$\mathbf{F} \triangleq \lambda x.\lambda y.y$$
$$\mathbf{if}(M, M_0, M_1) \triangleq M\,M_0\,M_1$$

When expanded, $\mathbf{if}(\mathbf{T}, M_0, M_1) \equiv_\beta M_0$ and $\mathbf{if}(\mathbf{F}, M_0, M_1) \equiv_\beta M_1$. That is, *the boolean itself* chooses between the two alternatives! Rather than being "passive data," the booleans are active programs that make the appropriate choice.

Now define the *Church pair* $\langle M_0, M_1\rangle$ by the λ-term $\lambda z.z\,M_0\,M_1$, which takes in a function and applies it to the two components of the pair.[2] Check that under this definition $\langle M_0, M_1\rangle\,\mathbf{T} \equiv_\beta M_0$ and $\langle M_0, M_1\rangle\,\mathbf{F} \equiv_\beta M_1$. That is, to obtain the first component of a pair, apply it to $\mathbf{T}$, and to obtain the second, apply it to $\mathbf{F}$.

Barendregt's idea is that the natural number $n \geq 0$ is represented by a sequence of $\mathbf{F}$'s ending with $\mathbf{I}$, the identity combinator. *This is a very cool hack!*

$$\mathbf{0} \triangleq \mathbf{I}$$
$$\mathbf{succ}(M) \triangleq \langle \mathbf{F}, M\rangle$$

[1] If you want a hard programming challenge, try to do it!

[2] The argument z to the pair is sometimes called a *visitor*, and the programming method is nowadays called, rather grandly, the *visitor pattern*.

The Barendregt numeral for n, written $\overline{n}$, is defined to be $\mathbf{succ}(\ldots\mathbf{succ}(\mathbf{0}))$, with n iterations of the successor ending with zero.

Exercise 7.3. *What is the normal form of $\overline{n}$ for $n \geq 0$?*

To appreciate the "hack value" of the Barendregt numerals, be sure to do the following exercise!

Exercise 7.4. *Define $\mathbf{ifz}(M, M_0, M_1)$ in terms of the Barendregt numerals.*

8 Normalization

The β-equivalence relation for the λ-calculus defines *when* two terms are equal by virtue of calculation, but it does not provide a guide for *how* to determine when two terms are β-equivalent. A natural approach is to do what you did in school: to see if two polynomials are the same, simplify both sides as much as possible and check whether the results have the same form when written in decreasing order of degree. Unlike in school, however, a λ-term may not have a simplest form—the simplification process can go on forever. But there is no better way to proceed than to try, with the understanding that it may not converge to an answer.

In the λ-calculus each step of simplification is called a *β-reduction*; it consists of replacing $(\lambda x.M)\,N$ by $[N/x]M$ anywhere within a term. When a term cannot be β-reduced, it is said to be in *β-normal form*. The process of finding a β-normal form for a term, if it has one, is called *β-normalization*. The *Church-Rosser Theorem* states that a term has at most one β-normal form, which is obtained by some sequence of β-reductions. Unfortunately, not every sequence of β-reductions will lead to a β-normal form—it is possible to charge off to infinity performing reductions that do not lead to a normal form, even though some other reductions would.

Put another way, how can one write a program to compute the β-normal form of a term, if it has one? That is, the program should not reduce forever needlessly if there is a way to reduce to a normal form. At first this may sound like a wildly underdetermined problem, but in fact there is a quite simple way to achieve it. The main idea is that any β-reductions that can be performed at the outermost level of a term *must* be performed if a normal form is to be reached. *(Weak) head reduction*, written $M \longmapsto_\beta M'$, is the process of performing one step of β-reduction at the outermost level of a term. It is defined by the rules in Figure 1.

If N is in head normal form, then it has one of two forms:

1. $\lambda x.P$ where P is some λ-term, or
2. $x\,M_1 \ldots M_k$, where $M_1, \ldots, M_k$ with $k \geq 0$ are some λ-terms.

To normalize a term M, first compute its head normal form, N, and proceed by cases:

1. If N is $\lambda x.P$, normalize P to P', and yield $\lambda x.P'$.
2. If N is $x\,M_1 \ldots M_k$, normalize each of $M_1, \ldots, M_k$ to $M_1', \ldots, M_k'$, respectively, and yield $x\,M_1' \ldots M_k'$.

The judgment M norm N, stating that the normal form of M is N, is defined in Figure 2. It makes use of the hnorm judgment defined in Figure 1.

$$\frac{}{\mathtt{ap}(\lambda(x.M)\,;M_2) \longmapsto_\beta [M_2/x]M} \tag{8a}$$

$$\frac{M_1 \longmapsto_\beta N_1}{\mathtt{ap}(M_1\,;M_2) \longmapsto_\beta \mathtt{ap}(N_1\,;M_2)} \tag{8b}$$

$$\frac{M \not\longmapsto_\beta}{M \mathsf{\ hnorm\ } M} \tag{9a}$$

$$\frac{M \longmapsto_\beta M' \quad M' \mathsf{\ hnorm\ } N'}{M \mathsf{\ hnorm\ } N'} \tag{9b}$$

Figure 1: Head Reduction and Head Normalization

$$\frac{M \mathsf{\ hnorm\ } x}{M \mathsf{\ norm\ } x} \tag{10a}$$

$$\frac{M \mathsf{\ hnorm\ } \mathtt{ap}(M_1\,;M_2) \quad M_1 \mathsf{\ norm\ } N_1 \quad M_2 \mathsf{\ norm\ } N_2}{M \mathsf{\ norm\ } \mathtt{ap}(N_1\,;N_2)} \tag{10b}$$

$$\frac{M \mathsf{\ hnorm\ } \lambda(x.M') \quad M' \mathsf{\ norm\ } N'}{M \mathsf{\ norm\ } \lambda(x.N')} \tag{10c}$$

Figure 2: Normalization

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.