Part I

The Lambda Calculus

This part deals with the lambda calculus. The introduction chapter is based on Jeremy Avigad's notes; part of it is now redundant and covered in later chapters. The chapters on syntax, Church–Rosser property, and lambda definability were produced by Zesen Qian during his Mitacs summer internship. They still have to be reviewed and revised.

Chapter 1

Introduction

This chapter consists of Jeremy's original concise notes on the lambda calculus. The sections need to be combined, and the material on lambda definability merged with the material in the separate, more detailed chapter on lambda definability.

1.1 Overview

lam:int:ovr:sec

The lambda calculus was originally designed by Alonzo Church in the early 1930s as a basis for constructive logic, and *not* as a model of the computable functions. But it was soon shown to be equivalent to other definitions of computability, such as the Turing computable functions and the partial recursive functions. The fact that this initially came as a small surprise makes the characterization all the more interesting.

Lambda notation is a convenient way of referring to a function directly by a symbolic expression which defines it, instead of defining a name for it. Instead of saying "let f be the function defined by $f(x) = x + 3$," one can say, "let f be the function $\lambda x.\,(x + 3)$." In other words, $\lambda x.\,(x + 3)$ is just a *name* for the function that adds three to its argument. In this expression, x is a dummy variable, or a placeholder: the same function can just as well be denoted by $\lambda y.\,(y+3)$. The notation works even with other parameters around. For example, suppose $g(x, y)$ is a function of two variables, and k is a natural number. Then $\lambda x.\,g(x, k)$ is the function which maps any x to $g(x, k)$.

This way of defining a function from a symbolic expression is known as *lambda abstraction*. The flip side of lambda abstraction is *application*: assuming one has a function f (say, defined on the natural numbers), one can *apply* it to any value, like 2. In conventional notation, of course, we write $f(2)$ for the result.

What happens when you combine lambda abstraction with application? Then the resulting expression can be simplified, by "plugging" the applicand

in for the abstracted variable. For example,

$$(\lambda x.\,(x+3))(2)$$

can be simplified to $2+3$.

Up to this point, we have done nothing but introduce new notations for conventional notions. The lambda calculus, however, represents a more radical departure from the set-theoretic viewpoint. In this framework:

1. Everything denotes a function.
2. Functions can be defined using lambda abstraction.
3. Anything can be applied to anything else.

For example, if F is a term in the lambda calculus, $F(F)$ is always assumed to be meaningful. This liberal framework is known as the *untyped* lambda calculus, where "untyped" means "no restriction on what can be applied to what."

digression

There is also a *typed* lambda calculus, which is an important variation on the untyped version. Although in many ways the typed lambda calculus is similar to the untyped one, it is much easier to reconcile with a classical set-theoretic framework, and has some very different properties.

Research on the lambda calculus has proved to be central in theoretical computer science, and in the design of programming languages. LISP, designed by John McCarthy in the 1950s, is an early example of a language that was influenced by these ideas.

1.2 The Syntax of the Lambda Calculus

lam:int:syn:sec

One starts with a sequence of variables x, y, z, ... and some constant symbols a, b, c, ... The set of terms is defined inductively, as follows:

1. Each variable is a term.
2. Each constant is a term.
3. If M and N are terms, so is (MN).
4. If M is a term and x is a variable, then $(\lambda x.\,M)$ is a term.

The system without any constants at all is called the *pure* lambda calculus. We'll mainly be working in the pure λ-calculus, so all lowerce letters will stand for variables. We use uppercase letters (M, N, etc.) to stand for terms of the λ-calculus.

We will follow a few notational conventions:

Convention 1. 1. When parentheses are left out, application takes place from left to right. For example, if M, N, P, and Q are terms, then $MNPQ$ abbreviates $(((MN)P)Q)$.

2. Again, when parentheses are left out, lambda abstraction is to be given the widest scope possible. From example, $\lambda x.\, MNP$ is read $(\lambda x.\, ((MN)P))$.

3. A lambda can be used to abstract multiple variables. For example, $\lambda xyz.\, M$ is short for $\lambda x.\, \lambda y.\, \lambda z.\, M$.

For example,

$$\lambda xy.\, xxyx\lambda z.\, xz$$

abbreviates

$$\lambda x.\, \lambda y.\, ((((xx)y)x)(\lambda z.\, (xz))).$$

You should memorize these conventions. They will drive you crazy at first, but you will get used to them, and after a while they will drive you less crazy than having to deal with a morass of parentheses.

Two terms that differ only in the names of the bound variables are called α-equivalent; for example, $\lambda x.\, x$ and $\lambda y.\, y$. It will be convenient to think of these as being the "same" term; in other words, when we say that M and N are the same, we also mean "up to renamings of the bound variables." Variables that are in the scope of a λ are called "bound", while others are called "free." There are no free variables in the previous example; but in

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

y and x are free, and z is bound.

1.3 Reduction of Lambda Terms

lam:int:red:sec

What can one do with lambda terms? Simplify them. If M and N are any lambda terms and x is any variable, we can use $M[N/x]$ to denote the result of substituting N for x in M, after renaming any bound variables of M that would interfere with the free variables of N after the substitution. For example,

$$(\lambda w.\, xxw)[yyz/x] = \lambda w.\, (yyz)(yyz)w.$$

Alternative notations for substitution are $[N/x]M$, $[x/N]M$, and also $M[x/N]$ digression Beware!

Intuitively, $(\lambda x.\, M)N$ and $M[N/x]$ have the same meaning; the act of replacing the first term by the second is called *β-contraction*. $(\lambda x.\, M)N$ is called a *redex* and $M[N/x]$ its *contractum*. Generally, if it is possible to change a term P to P' by β-contraction of some subterm, we say that P *β-reduces to P' in one step*, and write $P \to P'$. If from P we can obtain P' with some number of one-step reductions (possibly none), then P *β-reduces* to P'; in symbols, $P \twoheadrightarrow P'$. A term that cannot be β-reduced any further is called *β-irreducible*, or *β-normal*. We will say "reduces" instead of "β-reduces," etc., when the context is clear.

Let us consider some examples.

1. We have
$$\begin{aligned}(\lambda x.\, xxy)\lambda z.\, z &\to (\lambda z.\, z)(\lambda z.\, z)y\\ &\to (\lambda z.\, z)y\\ &\to y.\end{aligned}$$

2. "Simplifying" a term can make it more complex:
$$\begin{aligned}(\lambda x.\, xxy)(\lambda x.\, xxy) &\to (\lambda x.\, xxy)(\lambda x.\, xxy)y\\ &\to (\lambda x.\, xxy)(\lambda x.\, xxy)yy\\ &\to \dots\end{aligned}$$

3. It can also leave a term unchanged:
$$(\lambda x.\, xx)(\lambda x.\, xx) \to (\lambda x.\, xx)(\lambda x.\, xx).$$

4. Also, some terms can be reduced in more than one way; for example,
$$(\lambda x.\, (\lambda y.\, yx)z)v \to (\lambda y.\, yv)z$$
by contracting the outermost application; and
$$(\lambda x.\, (\lambda y.\, yx)z)v \to (\lambda x.\, zx)v$$
by contracting the innermost one. Note, in this case, however, that both terms further reduce to the same term, zv.

The final outcome in the last example is not a coincidence, but rather illustrates a deep and important property of the lambda calculus, known as the "Church–Rosser property."

1.4 The Church–Rosser Property

lam:int:cr:sec
lam:int:cr:thm:church-rosser

Theorem 1.1. *Let M, N_1, and N_2 be terms, such that $M \twoheadrightarrow N_1$ and $M \twoheadrightarrow N_2$. Then there is a term P such that $N_1 \twoheadrightarrow P$ and $N_2 \twoheadrightarrow P$.*

Corollary 1.2. *Suppose M can be reduced to normal form. Then this normal form is unique.*

Proof. If $M \twoheadrightarrow N_1$ and $M \twoheadrightarrow N_2$, by the previous theorem there is a term P such that N_1 and N_2 both reduce to P. If N_1 and N_2 are both in normal form, this can only happen if $N_1 \equiv P \equiv N_2$. □

Finally, we will say that two terms M and N are *β-equivalent*, or just *equivalent*, if they reduce to a common term; in other words, if there is some P such that $M \twoheadrightarrow P$ and $N \twoheadrightarrow P$. This is written $M \overset{\beta}{=} N$. Using Theorem 1.1, you can check that $\overset{\beta}{=}$ is an equivalence relation, with the additional property that for every M and N, if $M \twoheadrightarrow N$ or $N \twoheadrightarrow M$, then $M \overset{\beta}{=} N$. (In fact, one can show that $\overset{\beta}{=}$ is the *smallest* equivalence relation having this property.)

1.5 Currying

lam:rep:cur:sec

A λ-abstract $\lambda x.\, M$ represents a function of one argument, which is quite a limitation when we want to define function accepting multiple arguments. One way to do this would be by extending the λ-calculus to allow the formation of pairs, triples, etc., in which case, say, a three-place function $\lambda x.\, M$ would expect its argument to be a triple. However, it is more convenient to do this by *Currying*.

Let's consider an example. We'll pretend for a moment that we have a $+$ operation in the λ-calculus. The addition function is 2-place, i.e., it takes two arguments. But a λ-abstract only gives us functions of one argument: the syntax does not allow expressions like $\lambda(x, y).\, (x+y)$. However, we can consider the one-place function $f_x(y)$ given by $\lambda y.\, (x + y)$, which adds x to its single argument y. Actually, this is not a single function, but a family of different functions "add x," one for each number x. Now we can define another one-place function g as $\lambda x.\, f_x$. Applied to argument x, $g(x)$ returns the function f_x—so its values are other functions. Now if we apply g to x, and then the result to y we get: $(g(x))y = f_x(y) = x + y$. In this way, the one-place function g can do the same job as the two-place addition function. "Currying" simply refers to this trick for turning two-place functions into one place functions (whose values are one-place functions).

Here is an example properly in the syntax of the λ-calculus. How do we represent the function $f(x, y) = x$? If we want to define a function that accepts two arguments and returns the first, we can write $\lambda x.\, \lambda y.\, x$, which literally is a function that accepts an argument x and returns the function $\lambda y.\, x$. The function $\lambda y.\, x$ accepts another argument y, but drops it, and always returns x. Let's see what happens when we apply $\lambda x.\, \lambda y.\, x$ to two arguments:

$$\begin{aligned}(\lambda x.\, \lambda y.\, x)MN &\xrightarrow{\beta}(\lambda y.\, M)N\\ &\xrightarrow{\beta}M\end{aligned}$$

In general, to write a function with parameters $x_1, \ldots, x_n$ defined by some term N, we can write $\lambda x_1.\, \lambda x_2.\, \ldots \lambda x_n.\, N$. If we apply n arguments to it we get:

$$\begin{aligned}(\lambda x_1.\, \lambda x_2.\, \ldots \lambda x_n.\, N)M_1 \ldots M_n &\xrightarrow{\beta}\\ &\xrightarrow{\beta} ((\lambda x_2.\, \ldots \lambda x_n.\, N)[M_1/x_1])M_2 \ldots M_n\\ &\equiv (\lambda x_2.\, \ldots \lambda x_n.\, N[M_1/x_1])M_2 \ldots M_n\\ &\;\;\vdots\\ &\xrightarrow{\beta} P[M_1/x_1] \ldots [M_n/x_n]\end{aligned}$$

The last line literally means substituting M_i for x_i in the body of the function definition, which is exactly what we want when applying multiple arguments to a function.

1.6 λ-Definable Arithmetical Functions

lam:int:rep:sec

How can the lambda calculus serve as a model of computation? At first, it is not even clear how to make sense of this statement. To talk about computability on the natural numbers, we need to find a suitable representation for such numbers. Here is one that works surprisingly well.

Definition 1.3. For each natural number n, define the *Church numeral* $\overline{n}$ to be the lambda term $\lambda x.\, \lambda y.\, (x(x(x(\ldots x(y)))))$, where there are n x's in all.

The terms $\overline{n}$ are "iterators": on input f, $\overline{n}$ returns the function mapping y to $f^n(y)$. Note that each numeral is normal. We can now say what it means for a lambda term to "compute" a function on the natural numbers.

Definition 1.4. Let $f(x_0, \ldots, x_{k-1})$ be an n-ary partial function from $\mathbb{N}$ to $\mathbb{N}$. We say a λ-term F *λ-defines* f iff for every sequence of natural numbers $n_0, \ldots, n_{k-1}$,

$$F\,\overline{n_0}\,\overline{n_1}\ldots\overline{n_{k-1}} \twoheadrightarrow \overline{f(n_0, n_1, \ldots, n_{k-1})}$$

if $f(n_0, \ldots, n_{k-1})$ is defined, and $F, \overline{n_0}\,\overline{n_1}\ldots\overline{n_{k-1}}$ has no normal form otherwise.

Theorem 1.5. *A function f is a partial computable function if and only if it is λ-defined by a lambda term.*

lam:int:rep:thm:lambda-def

explanation

This theorem is somewhat striking. As a model of computation, the lambda calculus is a rather simple calculus; the only operations are lambda abstraction and application! From these meager resources, however, it is possible to implement any computational procedure.

1.7 λ-Definable Functions are Computable

lam:int:cmp:sec

Theorem 1.6. *If a partial function f is λ-defined by a lambda term, it is computable.*

lam:int:cmp:thm:lambda-computable

Proof. Suppose a function f is λ-defined by a lambda term X. Let us describe an informal procedure to compute f. On input $m_0, \ldots, m_{n-1}$, write down the term $X\overline{m}_0 \ldots \overline{m}_{n-1}$. Build a tree, first writing down all the one-step reductions of the original term; below that, write all the one-step reductions of those (i.e., the two-step reductions of the original term); and keep going. If you ever reach a numeral, return that as the answer; otherwise, the function is undefined.

An appeal to Church's thesis tells us that this function is computable. A better way to prove the theorem would be to give a recursive description of this search procedure. For example, one could define a sequence primitive recursive functions and relations, "IsASubterm," "Substitute," "ReducesToInOneStep," "ReductionSequence," "Numeral," etc. The partial recursive procedure for

computing $f(m_0, \ldots, m_{n-1})$ is then to search for a sequence of one-step reductions starting with $X\overline{m_0}\ldots\overline{m_{n-1}}$ and ending with a numeral, and return the number corresponding to that numeral. The details are long and tedious but otherwise routine. □

1.8 Computable Functions are λ-Definable

lam:int:lrp:sec
lam:int:lrp:thm:computable-lambda

Theorem 1.7. *Every computable partial function is λ-definable.*

Proof. We need to show that every partial computable function f is λ-defined by a lambda term F. By Kleene's normal form theorem, it suffices to show that every primitive recursive function is λ-defined by a lambda term, and then that the functions λ-definable are closed under suitable compositions and unbounded search. To show that every primitive recursive function is λ-defined by a lambda term, it suffices to show that the initial functions are λ-definable, and that the partial functions that are λ-definable are closed under composition, primitive recursion, and unbounded search. □

We will use a more conventional notation to make the rest of the proof more readable. For example, we will write $M(x, y, z)$ instead of $Mxyz$. While this is suggestive, you should remember that terms in the untyped lambda calculus do not have associated arities; so, for the same term M, it makes just as much sense to write $M(x, y)$ and $M(x, y, z, w)$. But using this notation indicates that we are treating M as a function of three variables, and helps make the intentions behind the definitions clearer. In a similar way, we will say "define M by $M(x, y, z) = \ldots$" instead of "define M by $M = \lambda x.\, \lambda y.\, \lambda z.\, \ldots$."

1.9 The Basic Primitive Recursive Functions are λ-Definable

lam:int:bas:sec

Lemma 1.8. *The functions* zero, succ, *and* P_i^n *are λ-definable.*

Proof. zero is just $\lambda x.\, \lambda y.\, y$.

The successor function succ, is defined by $\mathrm{Succ}(u) = \lambda x.\, \lambda y.\, x(uxy)$. You should think about why this works; for each numeral $\overline{n}$, thought of as an iterator, and each function f, $\mathrm{Succ}(\overline{n}, f)$ is a function that, on input y, applies f n times starting with y, and then applies it once more.

There is nothing to say about projections: $\mathrm{Proj}_i^n(x_0, \ldots, x_{n-1}) = x_i$. In other words, by our conventions, Proj_i^n is the lambda term $\lambda x_0.\, \ldots \lambda x_{n-1}.\, x_i$.□

1.10 The λ-Definable Functions are Closed under Composition

lam:int:com:sec

Lemma 1.9. *The λ-definable functions are closed under composition.*

Proof. Suppose f is defined by composition from h, g_0, ..., g_{k-1}. Assuming h, g_0, ..., g_{k-1} are λ-defined by H, G_0, ..., G_{k-1}, respectively, we need to find a term F that λ-defines f. But we can simply define F by

$$F(x_0, \ldots, x_{l-1}) = H(G_0(x_0, \ldots, x_{l-1}), \ldots, G_{k-1}(x_0, \ldots, x_{l-1})).$$

In other words, the language of the lambda calculus is well suited to represent composition. □

1.11 λ-Definable Functions are Closed under Primitive Recursion

lam:int:pr:sec

When it comes to primitive recursion, we finally need to do some work. We will have to proceed in stages. As before, on the assumption that we already have terms G and H that λ-define functions g and h, respectively, we want a term H that λ-defines the function f defined by

$$\begin{aligned} f(0, \vec{z}) &= g(\vec{z}) \\ f(x+1, \vec{z}) &= h(z, f(x, \vec{z}), \vec{z}). \end{aligned}$$

So, in general, given lambda terms G' and H', it suffices to find a term F such that

$$\begin{aligned} F(\overline{0}, \vec{z}) &\equiv G(\vec{z}) \\ F(\overline{n+1}, \vec{z}) &\equiv H(\overline{n}, F(\overline{n}, \vec{z}), \vec{z}) \end{aligned}$$

for every natural number n; the fact that G' and H' λ-define g and h means that whenever we plug in numerals $\overline{\vec{m}}$ for $\vec{z}$, $F(\overline{n+1}, \overline{\vec{m}})$ will normalize to the right answer.

But for this, it suffices to find a term F satisfying

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n, where

$$\begin{aligned} G &= \lambda \vec{z}.\, G'(\vec{z}) \text{ and} \\ H(u, v) &= \lambda \vec{z}.\, H'(u, v(u, \vec{z}), \vec{z}). \end{aligned}$$

In other words, with lambda trickery, we can avoid having to worry about the extra parameters $\vec{z}$—they just get absorbed in the lambda notation.

Before we define the term F, we need a mechanism for handling ordered pairs. This is provided by the next lemma.

Lemma 1.10. *There is a lambda term D such that for each pair of lambda terms M and N, $D(M,N)(\overline{0}) \twoheadrightarrow M$ and $D(M,N)(\overline{1}) \twoheadrightarrow N$.*

Proof. First, define the lambda term K by

$$K(y) = \lambda x.\, y.$$

In other words, K is the term $\lambda y.\, \lambda x.\, y$. Looking at it differently, for every M, $K(M)$ is a constant function that returns M on any input.

Now define $D(x,y,z)$ by $D(x,y,z) = z(K(y))x$. Then we have

$$\begin{aligned} D(M,N,\overline{0}) &\twoheadrightarrow \overline{0}(K(N))M \twoheadrightarrow M \text{ and} \\ D(M,N,\overline{1}) &\twoheadrightarrow \overline{1}(K(N))M \twoheadrightarrow K(N)M \twoheadrightarrow N, \end{aligned}$$

as required. □

The idea is that $D(M,N)$ represents the pair $\langle M,N\rangle$, and if P is assumed to represent such a pair, $P(\overline{0})$ and $P(\overline{1})$ represent the left and right projections, $(P)_0$ and $(P)_1$. We will use the latter notations.

Lemma 1.11. *The λ-definable functions are closed under primitive recursion.*

Proof. We need to show that given any terms, G and H, we can find a term F such that

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n. The idea is roughly to compute sequences of *pairs*

$$\langle \overline{0}, F(\overline{0})\rangle, \langle \overline{1}, F(\overline{1})\rangle, \ldots,$$

using numerals as iterators. Notice that the first pair is just $\langle \overline{0}, G\rangle$. Given a pair $\langle \overline{n}, F(\overline{n})\rangle$, the next pair, $\langle \overline{n+1}, F(\overline{n+1})\rangle$ is supposed to be equivalent to $\langle \overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle$. We will design a lambda term T that makes this one-step transition.

The details are as follows. Define $T(u)$ by

$$T(u) = \langle S((u)_0), H((u)_0, (u)_1)\rangle.$$

Now it is easy to verify that for any number n,

$$T(\langle \overline{n}, M\rangle) \twoheadrightarrow \langle \overline{n+1}, H(\overline{n}, M)\rangle.$$

As suggested above, given G and H, define $F(u)$ by

$$F(u) = (u(T, \langle \overline{0}, G\rangle))_1.$$

In other words, on input $\overline{n}$, F iterates T n times on $\langle \overline{0}, G\rangle$, and then returns the second component. To start with, we have

1. $\overline{0}(T, \langle \overline{0}, G\rangle) \equiv \langle \overline{0}, G\rangle$
2. $F(\overline{0}) \equiv G$

By induction on n, we can show that for each natural number one has the following:

1. $\overline{n+1}(T, \langle \overline{0}, G\rangle) \equiv \langle \overline{n+1}, F(\overline{n+1})\rangle$
2. $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$

For the second clause, we have

$$\begin{aligned} F(\overline{n+1}) &\twoheadrightarrow (\overline{n+1}(T, \langle \overline{0}, G\rangle))_1 \\ &\equiv (T(\overline{n}(T, \langle \overline{0}, G\rangle)))_1 \\ &\equiv (T(\langle \overline{n}, F(\overline{n})\rangle))_1 \\ &\equiv (\langle \overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle)_1 \\ &\equiv H(\overline{n}, F(\overline{n})). \end{aligned}$$

Here we have used the induction hypothesis on the second-to-last line. For the first clause, we have

$$\begin{aligned} \overline{n+1}(T, \langle \overline{0}, G\rangle) &\equiv T(\overline{n}(T, \langle \overline{0}, G\rangle)) \\ &\equiv T(\langle \overline{n}, F(\overline{n})\rangle) \\ &\equiv \langle \overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle \\ &\equiv \langle \overline{n+1}, F(\overline{n+1})\rangle. \end{aligned}$$

Here we have used the second clause in the last line. So we have shown $F(\overline{0}) \equiv G$ and, for every n, $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$, which is exactly what we needed.□

1.12 Fixed-Point Combinators

lam:int:fix:sec

Suppose you have a lambda term g, and you want another term k with the property that k is β-equivalent to gk. Define terms

$$\mathrm{diag}(x) = xx$$

and

$$l(x) = g(\mathrm{diag}(x))$$

using our notational conventions; in other words, l is the term $\lambda x.\, g(xx)$. Let k be the term ll. Then we have

$$\begin{aligned} k &= (\lambda x.\, g(xx))(\lambda x.\, g(xx)) \\ &\twoheadrightarrow g((\lambda x.\, g(xx))(\lambda x.\, g(xx))) \\ &= gk. \end{aligned}$$

If one takes

$$Y = \lambda g.\,((\lambda x.\,g(xx))(\lambda x.\,g(xx)))$$

then Yg and $g(Yg)$ reduce to a common term; so $Yg \equiv_\beta g(Yg)$. This is known as "Curry's combinator." If instead one takes

$$Y = (\lambda xg.\,g(xxg))(\lambda xg.\,g(xxg))$$

then in fact Yg reduces to $g(Yg)$, which is a stronger statement. This latter version of Y is known as "Turing's combinator."

1.13 The λ-Definable Functions are Closed under Minimization

lam:int:min:sec

Lemma 1.12. *Suppose $f(x, y)$ is primitive recursive. Let g be defined by*

$$g(x) \simeq \mu y\ f(x, y).$$

Then g is λ-definable.

Proof. The idea is roughly as follows. Given x, we will use the fixed-point lambda term Y to define a function $h_x(n)$ which searches for a y starting at n; then $g(x)$ is just $h_x(0)$. The function h_x can be expressed as the solution of a fixed-point equation:

$$h_x(n) \simeq \begin{cases} n & \text{if } f(x, n) = 0 \\ h_x(n+1) & \text{otherwise.} \end{cases}$$

Here are the details. Since f is primitive recursive, it is λ-defined by some term F. Remember that we also have a lambda term D, such that $D(M, N, \bar{0}) \twoheadrightarrow M$ and $D(M, N, \bar{1}) \twoheadrightarrow N$. Fixing x for the moment, to λ-define h_x we want to find a term H (depending on x) satisfying

$$H(\overline{n}) \equiv D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})).$$

We can do this using the fixed-point term Y. First, let U be the term

$$\lambda h.\,\lambda z.\,D(z, (h(Sz)), F(x, z)),$$

and then let H be the term YU. Notice that the only free variable in H is x. Let us show that H satisfies the equation above.

By the definition of Y, we have

$$H = YU \equiv U(YU) = U(H).$$

In particular, for each natural number n, we have

$$\begin{aligned} H(\overline{n}) &\equiv U(H, \overline{n}) \\ &\twoheadrightarrow D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})), \end{aligned}$$

as required. Notice that if you substitute a numeral $\overline{m}$ for x in the last line, the expression reduces to $\overline{n}$ if $F(\overline{m}, \overline{n})$ reduces to $\overline{0}$, and it reduces to $H(S(\overline{n}))$ if $F(\overline{m}, \overline{n})$ reduces to any other numeral.

To finish off the proof, let G be $\lambda x.\, H(\overline{0})$. Then G λ-defines g; in other words, for every m, $G(\overline{m})$ reduces to reduces to $\overline{g(m)}$, if $g(m)$ is defined, and has no normal form otherwise. □

Chapter 2

Syntax

2.1 Terms

lam:syn:trm:sec

The terms of the lambda calculus are built up inductively from an infinite supply of variables v_0, v_1, ..., the symbol "λ", and parentheses. We will use x, y, z, ... to designate variables, and M, N, P, ... to desginate terms.

lam:syn:trm:defn:term

Definition 2.1 (Terms). The set of *terms* of the lambda calculus is defined inductively by:

lam:syn:trm:defn:term-var

1. If x is a variable, then x is a term.

lam:syn:trm:defn:term-abs

2. If x is a variable and M is a term, then $(\lambda x.\, M)$ is a term.

lam:syn:trm:defn:term-app

3. If both M and N are terms, then (MN) is a term.

If a term $(\lambda x.\, M)$ is formed according to (2) we say it is the result of an *abstraction*, and the x in λx is called a *parameter*. A term (MN) formed according to (3) is the result of an *application*.

The terms defined above are fully parenthesized. This can get rather cumbersome, as the term $(\lambda x.\, ((\lambda x.\, x)(\lambda x.\, (xx))))$ demnostrates. We will introduce conventions for avoiding parentheses. However, the official definition makes it easy to determine how a term is constructed according to Definition 2.1. For example, the last step of forming the term $(\lambda x.\, ((\lambda x.\, x)(\lambda x.\, (xx))))$ must be abstraction where the parameter is x. It results by abstraction from the term $((\lambda x.\, x)(\lambda x.\, (xx)))$, which is an application of two terms. Each of these two terms is the result of an abstraction, and so on.

Problem 2.1. Describe the formation of $(\lambda g.\, (\lambda x.\, (g(xx)))(\lambda x.\, (g(xx))))$.

2.2 Unique Readability

lam:syn:unq:sec

We may wonder if for each term there is a unique way of forming it, and there is. For each lambda term there is only one way to construct and interpret it.

In the following discussion, a *formation* is the procedure of constructing a term using the formation rules (one or several times) of Definition 2.1.

Lemma 2.2. *A term starts with either a variable or a parenthesis.* *lam:syn:unq: lem:term-start*

Proof. Something counts as a term only if it is constructed according to Definition 2.1. If it is the result of (1), it must be a variable. If it is the result of (2) or (3), it starts with a parenthesis. □

Lemma 2.3. *The result of an application starts with either two parentheses or a parenthesis and a variable.* *lam:syn:unq: lem:app-start*

Proof. If M is the result of an application, it is of the form (PQ), so it begins with a parenthesis. Since P is a term, by Lemma 2.2, it begins either with a parenthesis or a variable. □

Lemma 2.4. *No proper initial part of a term is itself a term.* *lam:syn:unq: lem:initial*

Problem 2.2. Prove Lemma 2.4 by induction on the length of terms.

Proposition 2.5 (Unique Readability). *There is a unique formation for each term. In other words, if a term M is formed by a formation, then it is the only formation that can form this term.* *lam:syn:unq: prop:unq*

Proof. We prove this by induction on the formation of terms.

1. M is of the form x, where x is some variable. Since the results of abstractions and applications always start with parentheses, they cannot have been used to construct M; Thus, the formation of M must be a single step of Definition 2.1(1).

2. M is of the form $(\lambda x.\, N)$, where x is some variable and N is a term. It could not have been constructed according to Definition 2.1(1), because it is not a single variable. It is not the result of an application, by Lemma 2.3. Thus M can only be the result of an abstraction on N. By inductive hypothesis we know that formation of N is itself unique.

3. M is of the form (PQ), where P and Q are terms. Since it starts with a parentheses, it cannot also be constructed by Definition 2.1(1). By Lemma 2.2, P cannot begin with λ, so (PQ) cannot be the result of an abstraction. Now suppose there were another way of constructing M by application, e.g., it is also of the form $(P'Q')$. Then P is a proper initial segment of P' (or vice versa), and this is impossible by Lemma 2.4. So P and Q are uniquely determined, and by inductive hypothesis we know that formations of P and Q is unique. □

A more readable paraphrase of the above proposition is as follows:

Proposition 2.6. *A term M can only be one of the following forms:*

1. *x, where x is a variable uniquely determined by M.*

2. *$(\lambda x.\, N)$, where x is a variable and N is another term, both of which is uniquely determined by M.*

3. *(PQ), where P and Q are two terms uniquely determined by M.*

2.3 Abbreviated Syntax

lam:syn:abb:sec

Terms as defined in Definition 2.1 are sometimes cumbersome to write, so it is useful to introduce a more concise syntax. We must of course be careful to make sure that the terms in the concise notation also are uniquely readable. One widely used version called *abbreviated terms* is as follows.

1. When parentheses are left out, application takes place from left to right. For example, if M, N, P, and Q are terms, then $MNPQ$ abbreviates $(((MN)P)Q)$.

2. Again, when parentheses are left out, lambda abstraction is given the widest scope possible. From example, $\lambda x.\, MNP$ is read as $(\lambda x.\, MNP)$.

3. A lambda can be used to abstract multiple variables. For example, $\lambda xyz.\, M$ is short for $\lambda x.\, \lambda y.\, \lambda z.\, M$.

For example,

$$\lambda xy.\, xxyx\lambda z.\, xz$$

abbreviates

$$(\lambda x.\, (\lambda y.\, ((((xx)y)x)(\lambda z.\, (xz))))).$$

Problem 2.3. Expand the abbreviated term $\lambda g.\, (\lambda x.\, g(xx))\lambda x.\, g(xx)$.

2.4 Free Variables

lam:syn:fv:sec

Lambda calculus is about functions, and lambda abstraction is how functions arise. Intuitively, $\lambda x.\, M$ is the function with values given by M when the argument to the function is assigned to x. But not every occurrence of x in M is relevant: if M contains another abstract $\lambda x.\, N$ then the occurrences of x in N are relevant to $\lambda x.\, N$ but not to $\lambda x.\, M$. So, a lambda abstract λx inside $\lambda x.\, M$ *binds* those occurrences of x in M that are not already bound by another lambda abstract—the *free* occurrences of x in M.

Definition 2.7 (Scope). If $\lambda x.\, M$ occurs inside a term N, then the corresponding occurrence of N is the *scope* of the λx.

Definition 2.8 (Free and bound occurrence). An occurrence of variable x in a term M is *free* if it is not in the scope of a λx, and *bound* otherwise. An occurrence of a variable x in $\lambda x.\, M$ is bound by the initial λx iff the occurrence of x in M is free.

Example 2.9. In $\lambda x.\, xy$, both x and y are in the scope of λx, so x is bound by λx. Since y is not in the scope of any λy, it is free. In $\lambda x.\, xx$, both occurrences of x are bound by λx, since both are free in xx. In $((\lambda x.\, xx)x)$, the last occurrence of x is free, since it is not in the scope of a λx. In $\lambda x.\, (\lambda x.\, x)x$, the scope of the first λx is $(\lambda x.\, x)x$ and the scope of the second λx is the second-to-last occurrence of x. In $(\lambda x.\, x)x$, the last occurrence of x is free, and the second-to-last is bound. Thus, the second-to-last occurrence of x in $\lambda x.\, (\lambda x.\, x)x$ is bound by the second λx, and the last occurrence by the first λx.

For a term P, we can check all variable occurrences in it and get a set of free variables. This set is denoted by $\mathrm{FV}(P)$ with a natural definition as follows:

Definition 2.10 (Free variables of a term). The set of *free variables* of a term is defined inductively by: lam:syn:fv: def:fv

1. $\mathrm{FV}(x) = \{x\}$ lam:syn:fv: def:fv1
2. $\mathrm{FV}(\lambda x.\, N) = \mathrm{FV}(N) \setminus \{x\}$ lam:syn:fv: def:fv2
3. $\mathrm{FV}(PQ) = \mathrm{FV}(P) \cup \mathrm{FV}(Q)$ lam:syn:fv: def:fv3

Problem 2.4.

1. Identify the scopes of λg and the two λx in this term: $\lambda g.\, (\lambda x.\, g(xx))\lambda x.\, g(xx)$.
2. In $\lambda g.\, (\lambda x.\, g(xx))\lambda x.\, g(xx)$, are all occurrences of variables bound? By which abstractions are they bound respectively?
3. Give $\mathrm{FV}(\lambda x.\, (\lambda y.\, (\lambda z.\, xy)z)y)$

explanation

A free variable is like a reference to the outside world (the *environment*), and a term containing free variables can be seen as a partially specified term, since its behaviour depends on how we set up the environment. For example, in the term $\lambda x.\, fx$, which accepts an argument x and returns f of that argument, the variable f is free. This value of the term is dependent on the environment it is in, in particular the value of f in that environment.

If we apply abstraction to this term, we get $\lambda f.\, \lambda x.\, fx$. This term is no longer dependent on the environment variable f, because it now designates a function that accepts two arguments and returns the result of applying the first to the second. Changing f in the environment won't have any effect on the behavior of this term, as the term will only use whatever is passed as an argument, and not the value of f in the environment.

Definition 2.11 (Closed term, combinator). A term with no free variables is called a *closed term*, or a *combinator*.

Lemma 2.12. *lam:syn:fv: lem:fv*

1. *If $y \neq x$, then $y \in \mathrm{FV}(\lambda x.\, N)$ iff $y \in \mathrm{FV}(N)$.* lam:syn:fv: lem:fv-abs
2. *$y \in \mathrm{FV}(PQ)$ iff $y \in \mathrm{FV}(P)$ or $y \in \mathrm{FV}(Q)$.* lam:syn:fv: lem:fv-app

Proof. Exercise. □

Problem 2.5. Prove Lemma 2.12.

2.5 Substitution

lam:syn:sub:sec

Free variables are references to environment variables, thus it makes sense to actually use a specific value in the place of a free variable. For example, we may want to replace f in $\lambda x.\, fx$ with a specific term, like the identity function $\lambda y.\, y$. This results in $\lambda x.\, (\lambda y.\, y)x$. The process of replacing free variables with lambda terms is called substitution. explanation

lam:syn:sub:defn:substitution

Definition 2.13 (Substitution). The *substitution* of a term N for a variable x in a term M, $M[N/x]$, is defined inductively by:

lam:syn:sub:defn:substitution-1

1. $x[N/x] = N$.

lam:syn:sub:defn:substitution-2

2. $y[N/x] = y$ if $x \neq y$.

lam:syn:sub:def:substitution-3

3. $PQ[N/x] = (P[N/x])(Q[N/x])$.

lam:syn:sub:defn:substitution-4

4. $(\lambda y.\, P)[N/x] = \lambda y.\, P[N/x]$, if $x \neq y$ and $y \notin \mathrm{FV}(N)$, otherwise undefined.

In Definition 2.13(4), we require $x \neq y$ because we don't want to replace *bound* occurrences of the variable x in M by N. For example, if we compute the substitution $\lambda x.\, x[y/x]$, the result should not be $\lambda x.\, y$ but simply $\lambda x.\, x$. explanation

When substituting N for x in $\lambda y.\, P$, we also require that $y \notin \mathrm{FV}(N)$. For example, we cannot substitute y for x in $\lambda y.\, x$, i.e., $\lambda y.\, x[y/x]$, because it would result in $\lambda y.\, y$, a term that stands for the function that accepts an argument and returns it directly. But the term $\lambda y.\, x$ stands for a function that always returns the term x (or whatever x refers to). So the result we actually want is a function that accepts an argument, drop it, and returns the environment variable y. To do this properly, we would first have to "rename" the bound variable y.

Problem 2.6. What is the result of the following substitutions?

1. $\lambda y.\, x(\lambda w.\, vwx)[(uv)/x]$
2. $\lambda y.\, x(\lambda x.\, x)[(\lambda y.\, xy)/x]$
3. $y(\lambda v.\, xv)[(\lambda y.\, vy)/x]$

lam:syn:sub:thm:notinfv

Theorem 2.14. *If $x \notin \mathrm{FV}(M)$, then $\mathrm{FV}(M[N/x]) = \mathrm{FV}(M)$, if the left-hand side is defined.*

Proof. By induction on the formation of M.

1. M is a variable: exercise.

2. M is of the form (PQ): exercise.

3. M is of the form $\lambda y.\, P$, and since $\lambda y.\, P[N/x]$ is defined, it has to be $\lambda y.\, P[N/x]$. Then $P[N/x]$ has to be defined; also, $x \neq y$ and $x \notin \mathrm{FV}(Q)$. Then:

$$\begin{aligned}
&\mathrm{FV}(\lambda y.\, P[N/x]) = \\
&\qquad = \mathrm{FV}(\lambda y.\, P[N/x]) && \text{by (4)} \\
&\qquad = \mathrm{FV}(P[N/x]) \setminus \{y\} && \text{by Definition 2.10(2)} \\
&\qquad = \mathrm{FV}(P) \setminus \{y\} && \text{by inductive hypothesis} \\
&\qquad = \mathrm{FV}(\lambda y.\, P) && \text{by Definition 2.10(2)}
\end{aligned}$$

□

Problem 2.7. Complete the proof of Theorem 2.14.

Theorem 2.15. *If $x \in \mathrm{FV}(M)$), then $\mathrm{FV}(M[N/x]) = (\mathrm{FV}(M) \setminus \{x\}) \cup \mathrm{FV}(N)$, provided the left hand is defined.* *lam:syn:sub: thm:infv*

Proof. By induction on the formation of M.

1. M is a variable: exercise.

2. M is of the form PQ: Since $(PQ)[N/y]$ is defined, it has to be $(P[N/x])(Q[N/x])$ with both substitution defined. Also, since $x \in \mathrm{FV}(PQ)$, either $x \in \mathrm{FV}(P)$ or $x \in \mathrm{FV}(Q)$ or both. The rest is left as an exercise.

3. M is of the form $\lambda y.\, P$. Since $\lambda y.\, P[N/x]$ is defined, it has to be $\lambda y.\, P[N/x]$, with $P[N/x]$ defined, $x \neq y$ and $y \notin \mathrm{FV}(N)$; also, since $y \in \mathrm{FV}(\lambda x.\, P)$, we have $y \in \mathrm{FV}(P)$ too. Now:

$$\begin{aligned}
&\mathrm{FV}((\lambda y.\, P)[N/x]) = \\
&\quad = \mathrm{FV}(\lambda y.\, P[N/x]) \\
&\quad = \mathrm{FV}(P[N/x]) \setminus \{y\} \\
&\quad = ((\mathrm{FV}(P) \setminus \{y\}) \cup (\mathrm{FV}(N) \setminus \{x\}) && \text{by inductive hypothesis} \\
&\quad = (\mathrm{FV}(P) \setminus \{x, y\}) \cup \mathrm{FV}(N) && x \notin \mathrm{FV}(N) \\
&\quad = (\mathrm{FV}(\lambda y.\, P) \setminus \{x\}) \cup \mathrm{FV}(N)
\end{aligned}$$

□

Problem 2.8. Complete the proof of Theorem 2.15.

Theorem 2.16. *$x \notin \mathrm{FV}(M[N/x])$, if the right-hand side is defined and $x \notin \mathrm{FV}(N)$.* *lam:syn:sub: thm:clr*

Proof. Exercise. □

Problem 2.9. Prove Theorem 2.16.

lam:syn:sub:thm:inv **Theorem 2.17.** *If $M[y/x]$ is defined and $y \notin \mathrm{FV}(M)$, then $M[y/x][x/y] = M$.*

Proof. By induction on the formation of M.

1. M is a variable z: Exercise.
2. M is of the form (PQ). Then:
$$\begin{aligned}(PQ)[y/x][x/y] &= ((P[y/x])(Q[y/x]))[x/y] \\ &= (P[y/x][x/y])(Q[y/x][x/y]) \\ &= (PQ) \text{ by inductive hypothesis}\end{aligned}$$
3. M is of the form $\lambda z.\, N$. Because $\lambda z.\, N[y/x]$ is defined, we know that $z \neq y$. So:
$$\begin{aligned}&(\lambda z.\, N)[y/x][x/y] \\ &\qquad = (\lambda z.\, N[y/x])[x/y] \\ &\qquad = \lambda z.\, N[y/x][x/y] \\ &\qquad = \lambda z.\, N \text{ by inductive hypothesis}\end{aligned}$$
□

Problem 2.10. Complete the proof of Theorem 2.17.

2.6 α-Conversion

lam:syn:alp:sec What is the relation between $\lambda x.\, x$ and $\lambda y.\, y$? They both represent the identity function. They are, of course, syntactically different terms. They differ only in the name of the bound variable, and one is the result of "renaming" the bound variable in the other. This is called *α-conversion*.

Definition 2.18 (Change of bound variable, $\xrightarrow{\alpha}$). If a term M contains an occurrence of $\lambda x.\, N$, $y \notin \mathrm{FV}(N)$, and $N[y/x]$ is defined, then replacing this occurrence by
$$\lambda y.\, N[y/x]$$
resulting in M' is called a *change of bound variable*, written as $M \xrightarrow{\alpha} M'$.

Definition 2.19 (Compatibility of relation). A relation R on terms is said to be *compatible* if it satisfies following conditions:

1. If RNN' then $R\lambda x.\, N\lambda x.\, N'$
2. If RPP' then $R(PQ)(P'Q)$
3. If RQQ' then $R(PQ)(PQ')$

Thus let's rephrase the definition:

Definition 2.20 (Change of bound variable, $\xrightarrow{\alpha}$). *Change of bound variable* ($\xrightarrow{\alpha}$) is the smallest compatible relation on terms satisfying following condition:

$$\lambda x.\, N \xrightarrow{\alpha} \lambda y.\, N[y/x] \qquad \text{if } x \neq y,\ y \notin \mathrm{FV}(N) \text{ and } N[y/x] \text{ is defined}$$

"Smallest" here means the relation contains only pairs that are required by compatibility and the additional condition, and nothing else. Thus this relation can also be defined as follows:

Definition 2.21 (Change of bound variable, $\xrightarrow{\alpha}$). *Change of bound variable* ($\xrightarrow{\alpha}$) is inductively defined as follows: lam:syn:alp: defn:aconvone

1. If $N \xrightarrow{\alpha} N'$ then $\lambda x.\, N \xrightarrow{\alpha} \lambda x.\, N'$ lam:syn:alp: defn:aconvone1
2. If $P \xrightarrow{\alpha} P'$ then $(PQ) \xrightarrow{\alpha} (P'Q)$ lam:syn:alp: defn:aconvone2
3. If $Q \xrightarrow{\alpha} Q'$ then $(PQ) \xrightarrow{\alpha} (PQ')$ lam:syn:alp: defn:aconvone3
4. If $x \neq y$, $y \notin \mathrm{FV}(N)$ and $N[y/x]$ is defined, then $\lambda x.\, N \xrightarrow{\alpha} \lambda y.\, N[y/x]$. lam:syn:alp: defn:aconvone4

The definitions are equivalent, but we leave the proof as an exercise. From now on we will use the inductive definition.

Definition 2.22 (α-conversion, $\xrightarrow{\alpha}\!\!\!\!\rightarrow$). *$\alpha$-conversion* ($\xrightarrow{\alpha}\!\!\!\!\rightarrow$) is the smallest reflexitive and transitive relation on terms containing $\xrightarrow{\alpha}$.

As above, "smallest" means the relation only contains pairs required by transitivity, and $\xrightarrow{\alpha}$, which leads to the following equivalent definition:

Definition 2.23 (α-conversion, $\xrightarrow{\alpha}\!\!\!\!\rightarrow$). *$\alpha$-conversion* ($\xrightarrow{\alpha}\!\!\!\!\rightarrow$) is inductively defined as follows: lam:syn:alp: defn:aconv

1. If $P \xrightarrow{\alpha}\!\!\!\!\rightarrow Q$ and $Q \xrightarrow{\alpha}\!\!\!\!\rightarrow R$, then $P \xrightarrow{\alpha}\!\!\!\!\rightarrow R$. lam:syn:alp: defn:aconv1
2. If $P \xrightarrow{\alpha} Q$, then $P \xrightarrow{\alpha}\!\!\!\!\rightarrow Q$. lam:syn:alp: defn:aconv2
3. $P \xrightarrow{\alpha}\!\!\!\!\rightarrow P$. lam:syn:alp: defn:aconv3

Example 2.24. $\lambda x.\, fx$ α-converts to $\lambda y.\, fy$, and conversely. Informally speaking, they are both functions that accept an argument and return f of that argument, refering to the environment variable f.

$\lambda x.\, fx$ does not α-convert to $\lambda x.\, gx$. Informally speaking, they refer to the environment variables f and g respectively, and this makes them different functions: they behave differently in environments where f and g are different.

Problem 2.11. Are the following pairs of terms α-convertible?

1. $\lambda x.\, \lambda y.\, x$ and $\lambda y.\, \lambda x.\, y$
2. $\lambda x.\, \lambda y.\, x$ and $\lambda c.\, \lambda b.\, a$
3. $\lambda x.\, \lambda y.\, x$ and $\lambda c.\, \lambda b.\, a$

lam:syn:alp: lem:fv-one

Lemma 2.25. *If $P \xrightarrow{\alpha} Q$ then $\mathrm{FV}(P) = \mathrm{FV}(Q)$.*

Proof. By induction on the derivation of $P \xrightarrow{\alpha} Q$.

1. If the last rule is (4), then P is of the form $\lambda x.\, N$ and Q of the form $\lambda y.\, N[y/x]$, with $x \neq y$, $y \notin \mathrm{FV}(N)$ and $N[y/x]$ defined. We distinguish cases according to whether $x \in \mathrm{FV}(N)$:

   a) If $x \in FV(N)$, then:

   $$\begin{aligned} \mathrm{FV}(\lambda y.\, N[y/x]) &= \mathrm{FV}(N[y/x]) \setminus \{y\} \\ &= ((\mathrm{FV}(N) \setminus \{x\}) \cup \{y\}) \setminus \{y\} \qquad \text{by Theorem 2.15} \\ &= \mathrm{FV}(N) \setminus \{x\} \\ &= \mathrm{FV}(\lambda x.\, N) \end{aligned}$$

   b) If $x \notin FV(N)$, then:

   $$\begin{aligned} \mathrm{FV}(\lambda y.\, N[y/x]) &= FVN[y/x] \setminus \{y\} \\ &= \mathrm{FV}(N) \setminus \{x\} \qquad \text{by Theorem 2.14} \\ &= \mathrm{FV}(\lambda x.\, N). \end{aligned}$$

2. The other three cases are left as exercises. □

Problem 2.12. Complete the proof of Lemma 2.25.

lam:syn:alp: lem:inv

Lemma 2.26. *If $P \xrightarrow{\alpha} Q$ then $Q \xrightarrow{\alpha} P$.*

Proof. Induction on the derivation of $P \xrightarrow{\alpha} Q$.

1. If the last rule is (4), then P is of the form $\lambda x.\, N$ and Q of the form $\lambda y.\, N[y/x]$, where $x \neq y$, $y \notin \mathrm{FV}(N)$ and $N[y/x]$ defined. First, we have $y \notin \mathrm{FV}(N[y/x])$ by Theorem 2.16. By Theorem 2.17 we have that $N[y/x][x/y]$ is not only defined, but also equal to N. Then by (4), we have $\lambda y.\, N[y/x] \xrightarrow{\alpha} \lambda x.\, N[y/x][x/y] = \lambda x.\, N$. □

Problem 2.13. Complete the proof of Lemma 2.26

Theorem 2.27. *α-Conversion is an equivalence relation on terms, i.e., it is reflexive, symmetric, and transitive.*

Proof. 1. For each term M, M can be changed to M by *zero* changes of bound variables.

2. If P is α-converts to Q by a series of changes of bound variables, then from Q we can just inverse these changes (by Lemma 2.26) in opposite order to obtain P.

3. If P α-converts to Q by a series of changes of bound variables, and Q to R by another series, then we can change P to R by first applying the first series and then the second series. □

From now on we say that M and N are *α-equivalent*, $M \stackrel{\alpha}{=} N$, iff M α-converts to N (which, as we've just shown, is the case iff N α-converts to M).

Theorem 2.28. *If $M \stackrel{\alpha}{=} N$, then $\mathrm{FV}(M) = \mathrm{FV}(N)$.* *lam:syn:alp: thm:fv*

Proof. Immediate from Lemma 2.25. □

Lemma 2.29. *If $R \stackrel{\alpha}{=} R'$ and $M[R/y]$ is defined, then $M[R'/y]$ is defined and α-equivalent to $M[R/y]$.* *lam:syn:alp: lem:sub:R*

Proof. Exercise. □

Problem 2.14. Prove Lemma 2.29.

Recall that in section 2.5, substitution is undefined in some cases; however, using α-conversion on terms, we can make substitution always defined by renaming bound variables. The result preserves α-equivalence, as shown in this theorem:

Theorem 2.30. *For any M, R, and y, there exists M' such that $M \stackrel{\alpha}{=} M'$ and $M'[R/y]$ is defined. Moreover, if there is another pair $M'' \stackrel{\alpha}{=} M$ and R'' where $M''[R''/y]$ is defined and $R'' \stackrel{\alpha}{=} R$, then $M'[R/y] \stackrel{\alpha}{=} M''[R''/y]$.* *lam:syn:alp: thm:sub*

Proof. By induction on the formation of M:

1. M is a variable z: Exercise.

2. Suppose M is of the form $\lambda x.\, N$. Select a variable z other than x and y and such that $z \notin \mathrm{FV}(N)$ and $z \notin \mathrm{FV}(R)$. By inductive hypothesis, we there is N' such that $N' \stackrel{\alpha}{=} N$ and $N'[z/x]$ is defined. Then $\lambda x.\, N \stackrel{\alpha}{=} \lambda x.\, N'$ too, by Definition 2.21(1). Now $\lambda x.\, N' \stackrel{\alpha}{=} \lambda z.\, N'[z/x]$ by Definition 2.21(4). We can do this because $z \neq x$, $z \notin FV(N')$ and $N'[z/x]$ is defined. Finally, $\lambda z.\, N'[z/x][R/y]$ is defined, because $z \neq y$ and $z \notin FV(R)$.

Moreover, if there is another N'' and R'' satisfying the same conditions,

$$\begin{aligned}(\lambda z.\, N''[z/x])[R''/y] = & \\ &= \lambda z.\, N''[z/x][R''/y] \\ &= \lambda z.\, N''[z/x][R/y] && \text{by Lemma 2.29} \\ &= \lambda z.\, N'[z/x][R/y] && \text{by inductive hypothesis} \\ &= (\lambda z.\, N'[z/x])[R/y]\end{aligned}$$

3. M is of the form (PQ): Exercise. □

Problem 2.15. Complete the proof of Theorem 2.30.

lam:syn:alp:cor:sub **Corollary 2.31.** *For any M, R, and y, there exists a pair of M' and R' such that $M' \stackrel{\alpha}{=} M$, $R \stackrel{\alpha}{=} R'$ and $M'[R'/y]$ is defined. Moreover, if there is another pair $M'' \stackrel{\alpha}{=} M$ and R'' with $M'[R'/y]$ defined, then $M'[R'/y] \stackrel{\alpha}{=} M''[R''/y]$.*

Proof. Immediate from Theorem 2.30. □

2.7 The De Bruijn Index

lam:syn:deb:sec α-Equivalence is very natural, as terms that are α-equivalent "mean the same." In fact, it is possible to give a syntax for lambda terms which does not distinguish terms that can be α-converted to each other. The best known replaces variables by their *De Bruijn index*.

When we write $\lambda x.\, M$, we explicitly state that x is the parameter of the function, so that we can use x in M to refer to this parameter. In the de Bruijn index, however, parameters have no name and reference to them in the function body is denoted by a number denoting the levels of abstraction between them. For example, consider the example of $\lambda x.\, \lambda y.\, yx$: the outer abstraction is on binds the variable x; the inner abstraction binds the variable is y; the sub-term yx lies in the scope of the inner abstraction: there is no abstraction between y and its abstract λy, but one abstract between x and its abstract λx. Thus we write $0\,1$ for yx, and $\lambda.\, \lambda.\, 0\,1$ for the entire term.

Definition 2.32. De Bruijn terms are inductively defines as follows:

1. n, where n is any natural number.
2. PQ, where P and Q are both De Bruijn terms.
3. $\lambda.\, N$, where N is a De Bruijn term.

A formalized translation from ordinary lambda terms to De Bruijn indexed terms is as follows:

Definition 2.33.

$$\begin{aligned} F_\Gamma(x) &= \Gamma(x) \\ F_\Gamma(PQ) &= F_\Gamma(P)F_\Gamma(Q) \\ F_\Gamma(\lambda x.\, N) &= \lambda.\, F_{x,\Gamma}(N) \end{aligned}$$

where Γ is a list of variables indexed from zero, and $\Gamma(x)$ denotes the position of the variable x in Γ. For example, if Γ is x, y, z, then $\Gamma(x)$ is 0 and $\Gamma(z)$ is 2.

x, Γ denotes the list resulted from pushing x to the head of Γ; for instance, continuing the Γ in last example, w, Γ is w, x, y, z.

Recovering a standard lambda term from a de Bruijn term is done as follows:

Definition 2.34.

$$\begin{aligned} G_\Gamma(n) &= \Gamma[n] \\ G_\Gamma(PQ) &= G_\Gamma(P)G_\Gamma(Q) \\ G_\Gamma(\lambda.\, N) &= \lambda x.\, G_{x,\Gamma}(N) \end{aligned}$$

where Γ is again a list of variables indexed from zero, and $\Gamma[n]$ denotes the variable in position n. For example, if Γ is x, y, z, then $\Gamma[1]$ is y.

The variable x in last equation is chosen to be any variable that not in Γ.

Here we give some results without proving them:

Proposition 2.35. *If $M \xrightarrow{\alpha} M'$, and Γ is any list containing $\mathrm{FV}(M)$, then $F_\Gamma(M) \equiv F_\Gamma(M')$.*

2.8 Terms as α-Equivalence Classes

lam:syn:tr:sec

From now on, we will consider terms up to α-equivalence. That means when we write a term, we mean its α-equivalance class it is in. For example, we write $\lambda a.\, \lambda b.\, ac$ for the set of all terms α-equivalent to it, such as $\lambda a.\, \lambda b.\, ac$, $\lambda b.\, \lambda a.\, bc$, etc.

Also, while in previous sections letters such as N, Q are used to denote a term, from now on we use them to denote a class, and it is these classes instead of terms that will be our subjects of study in what follows. Letters such as x, y continues to denote a variable.

We also adopt the notation $\underline{M}$ to denote an arbitrary element of the class M, and $\underline{M}_0, \underline{M}_1, etc.$ if we need more than one.

We reuse the notations from terms to simplify our wording. We have following definition on classes:

Definition 2.36.

1. $\lambda x.\, N$ is defined as the class containing $\lambda x.\, \underline{N}$.
2. PQ is defined to be the class containing $\underline{P}\,\underline{Q}$.

It is not hard to see that they are well defined, because α-conversion is compatible.

lam:syn:tr: def:fv **Definition 2.37.** The *free variables* of an α-equivalence class M, or $FV(M)$, is defined to be $FV(\underline{M})$.

This is well defined since $FV(\underline{M}_0) = FV(\underline{M}_1)$, as shown in Theorem 2.28.

We also reuse the notation for substition into classes:

lam:syn:tr: def:sub **Definition 2.38.** The *substitution* of R for y in M, or $M[R/y]$, is defined to be $\underline{M}_{\underline{R}/y}$, for any $\underline{M}$ and $\underline{R}$ making the substition defined.

This is also well defined as shown in Corollary 2.31.

Note how this definition significantly simplifies our reasoning. For example:

lam:syn:tr: eq:1 lam:syn:tr: eq:2
$$\begin{aligned} \lambda x.\, x[y/x] &= && (2.1)\\ &= \lambda z.\, z[y/x] && (2.2)\\ &= \lambda z.\, z[y/x] && (2.3)\\ &= \lambda z.\, z && (2.4) \end{aligned}$$

eq. (2.1) is undefined if we still regard it as substitution on terms; but as mentioned earlier, we now consider it a substitution on classes, which is why eq. (2.2) can happen: we can replace $\lambda x.\, x$ with $\lambda z.\, z$ because they belong to the same class.

For the same reason, from now on we will assume that the representatives we choose always satisfy the conditions needed for substitution. For example, when we see $\lambda x.\, N[R/y]$, we will assume the representative $\lambda x.\, N$ is chosen so that $x \neq y$ and $x \notin FV(R)$.

Since it is a bit strange to call $\lambda x.\, x$ a "class", let's call them Λ-terms (or simply "terms" in the rest of the part) from now on, to distinguish them from λ-terms that we are familiar with.

> We cannot say goodbye to terms yet: the whole definition of Λ-terms is based on λ-terms, and we haven't provided a method to define functions on Λ-terms, which means all such functions have to be first defined on λ-terms, and then "projected" to Λ-terms, as we did for substitutions. However we assume the reader can intuitively understand how we can define functions on Λ-terms.

2.9 β-reduction

lam:int:bet: sec When we see $(\lambda m.\, (\lambda y.\, y)m)$, it is natural to conjecture that it has some connection with $\lambda m.\, m$, namely the second term should be the result of "simplifying" the first. The notion of *β-reduction* captures this intuition formally.

Definition 2.39 (β-contraction, $\xrightarrow{\beta}$). *The β-contraction* ($\xrightarrow{\beta}$) is the smallest compatible relation on terms satisfying the following condition: lam:int:bet: defn:betacontr

$$(\lambda x.\, N)Q \xrightarrow{\beta} N[Q/x]$$

We say P is *β-contracted* to Q if $P \xrightarrow{\beta} Q$. A term of the form $(\lambda x.\, N)Q$ is called a *redex*.

Problem 2.16. Spell out the equivalent inductive definitions of β-contraction as we did for change of bound variable in Definition 2.21. lam:int:bet: prob:def

Definition 2.40 (β-reduction, $\xrightarrow{\beta}\!\!\!\!\rightarrow$). *$\beta$-reduction* ($\xrightarrow{\beta}\!\!\!\!\rightarrow$) is the smallest reflexive, transitive relation on terms containing $\xrightarrow{\beta}$. We say P is β-reduced to Q if $P \xrightarrow{\beta}\!\!\!\!\rightarrow Q$. lam:int:bet: defn:betared

We will write $\to$ instead of $\xrightarrow{\beta}$, and $\twoheadrightarrow$ instead of $\xrightarrow{\beta}\!\!\!\!\rightarrow$ when context is clear.

Informally speaking, $M \xrightarrow{\beta}\!\!\!\!\rightarrow N$ if and only if M can be changed to N by zero or several steps of β-contraction.

Definition 2.41 (β-normal). A term that cannot be β-contracted any further is said to be *β-normal*.

If $M \xrightarrow{\beta}\!\!\!\!\rightarrow N$ and N is β-normal, then we say N is a *normal form* of M. One may ask if the normal form of a term is unique, and the answer is yes, as we will see later.

Let us consider some examples.

1. We have

$$\begin{aligned}(\lambda x.\, xxy)\lambda z.\, z &\to (\lambda z.\, z)(\lambda z.\, z)y\\ &\to (\lambda z.\, z)y\\ &\to y\end{aligned}$$

2. "Simplifying" a term can actually make it more complex:

$$\begin{aligned}(\lambda x.\, xxy)(\lambda x.\, xxy) &\to (\lambda x.\, xxy)(\lambda x.\, xxy)y\\ &\to (\lambda x.\, xxy)(\lambda x.\, xxy)yy\\ &\to \dots\end{aligned}$$

3. It can also leave a term unchanged:

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

4. Also, some terms can be reduced in more than one way; for example,

$$(\lambda x.\,(\lambda y.\,yx)z)v \to (\lambda y.\,yv)z$$

by contracting the outermost application; and

$$(\lambda x.\,(\lambda y.\,yx)z)v \to (\lambda x.\,zx)v$$

by contracting the innermost one. Note, in this case, however, that both terms further reduce to the same term, zv.

The final outcome in the last example is not a coincidence, but rather illustrates a deep and important property of the lambda calculus, known as the Church–Rosser property.

In general, there is more than one way to β-reduce a term, thus many reduction strategies have been invented, among which the most common is the *natural strategy*. The natural strategy always contracts the *left-most* redex, where the position of a redex is defined as its starting point in the term. The natural strategy has the useful property that a term can be reduced to a normal form by some strategy iff it can be reduced to normal form using the natural strategy. In what follows we will use the natural stratuegy unless otherwise specified. digression

Definition 2.42 (β-equivalence, $=$). *β-Equivalence* $(=)$ is the relation inductively defined as follows:

1. $M = M$.
2. If $M = N$, then $N = M$.
3. If $M = N$, $N = O$, then $M = O$.
4. If $M = N$, then $PM = PN$.
5. If $M = N$, then $MQ = NQ$.
6. If $M = N$, then $\lambda x.\,M = \lambda x.\,N$.
7. $(\lambda x.\,N)Q = N[Q/x]$.

The first three rules make the relation an equivalence relation; the next three make it compatible; the last ensures that it contains β-contraction.

Informally speaking, two terms are β-equivalent if and only if one of them can be changed to the other in zero or more steps of β-contraction, or "inverse" of β-contraction. The inverse of β-contraction is defined so that M inverse-β-contracts to N iff N β-contracts to M.

Besides the above rules, we will extend the relation with more rules, and denote the extended equivalence relation as $\overset{X}{=}$, where X is the extending rule.

2.10 η-conversion

There is another relation on λ terms. In section 2.4 we used the example $\lambda x.\,(fx)$, which accepts an argument and applies f to it. In other words, it is the same function as f: $\lambda x.\,(fx)N$ and fN both reduce to fN. We use η-reduction (and η-extension) to capture this idea. lam:syn:eta:sec

Definition 2.43 (η-contraction, $\xrightarrow{\eta}$). *η-contraction* ($\xrightarrow{\eta}$) is the smallest compatible relation on terms satisfying the following condition: lam:syn:eta:defn:beredone

$$\lambda x.\,Mx \xrightarrow{\eta} M \text{ provided } x \notin FV(M)$$

Definition 2.44 ($\beta\eta$-reduction, $\xrightarrow{\beta\eta}\!\!\!\!\!\rightarrow$). *$\beta\eta$-reduction* ($\xrightarrow{\beta\eta}\!\!\!\!\!\rightarrow$) is the smallest reflexive, transitive relation on terms containing $\xrightarrow{\beta}$ and $\xrightarrow{\eta}$, i.e., the rules of reflexivity and transitive plus the following two rules: lam:syn:eta:defn:bered

1. If $M \xrightarrow{\beta} N$ then $M \xrightarrow{\beta\eta}\!\!\!\!\!\rightarrow N$. lam:syn:eta:defn:bered3
2. If $M \xrightarrow{\eta} N$ then $M \xrightarrow{\beta\eta}\!\!\!\!\!\rightarrow N$. lam:syn:eta:defn:bered4

Definition 2.45. We extend the equivalence relation $=$ with the η-conversion rule:

$$\lambda x.\,fx = f$$

and denote the extended relation as $\overset{\eta}{=}$.

η-equivalence is important because it is related to extensionality of lambda terms:

Definition 2.46 (Extensionality). We extend the equivalence relation $=$ with the (ext) rule:

$$\text{If } Mx = Nx \text{ then } M = N, \text{ provided } x \notin FV(MN).$$

and denote the extended relation as $\overset{ext}{=}$.

Roughly speaking, the rule states that two terms, viewed as functions, should be considered equal if they behave the same for the same argument.

We now prove that the η rule provides exactly the extensionality, and nothing else.

Theorem 2.47. *$M \overset{ext}{=} N$ if and only if $M \overset{\eta}{=} N$.*

Proof. First we prove that $\overset{\eta}{=}$ is closed under the extensionality rule. That is, *ext* rule doesn't add anything to $\overset{\eta}{=}$. We then have $\overset{\eta}{=}$ contains $\overset{ext}{=}$, and if $M \overset{ext}{=} N$, then $M \overset{\eta}{=} N$.

To prove $\overset{\eta}{=}$ is closed under *ext*, note that for any $M = N$ derived by the *ext* rule, we have $Mx \overset{\eta}{=} Nx$ as premise. Then we have $\lambda x.\, Mx \overset{\eta}{=} \lambda x.\, Nx$ by a rule of $=$, applying η on both side gives us $M \overset{\eta}{=} N$.

Similarly we prove that the η rule is contained in $\overset{ext}{=}$. For any $\lambda x.\, Mx$ and M with $x \notin FV(M)$, we have that $(\lambda x.\, Mx)x \overset{ext}{=} Mx$, giving us $\lambda x.\, Mx \overset{ext}{=} M$ by the *ext* rule. □

Chapter 3

The Church–Rosser Property

3.1 Definition and Properties

In this chapter we introduce the concept of Church–Rosser property and some common properties of this property. lam:cr:dap:sec

Definition 3.1 (Church–Rosser property, CR). A relation $\xrightarrow{X}$ on terms is said to satisfy the *Church–Rosser property* iff, whenever $M \xrightarrow{X} P$ and $M \xrightarrow{X} Q$, then there exists some N such that $P \xrightarrow{X} N$ and $Q \xrightarrow{X} N$.

We can view the lambda calculus as a model of computation in which terms in normal form are "values" and a reducibility relation on terms are the "calculation rules." The Church–Rosser property states is that when there is more than one way to proceed with a calculation, there is still only a single value of the expression.

To take an example from elementary algebra, there's more than one way to calculate $4 \times (1 + 2) + 3$. It can either be reduced to $4 \times 3 + 3$ (if we first reduce $1 + 2$ to 3) or to $4 \times 1 + 4 \times 2 + 3$ (if we first reduce $4 \times (1 + 2)$ using distributivity). Both of these, however, can be further reduced to $12 + 3$.

If we take $\xrightarrow{X}$ to be β-reduction, we easily see that a consequence of the Church–Rosser property is that if a term has a normal form, then it is unique. For suppose M can be reduced to P and Q, both of which are normal forms. By the Church–Rosser property, there exists some N such that both P and Q reduce to it. Since by assumption P and Q are normal forms, the reduction of P and Q to N can only be the trivial reduction, i.e., P, Q, and N are identical. This justifies our speaking of *the* normal form of a term.

In viewing the lambda calculus as a model of computation, then, the normal form of a term can be thought of as the "final result" of the computation starting with that term. The above corollary means there's only one, if any, final result of a computation, just like there is only one result of computing $4 \times (1 + 2) + 3$, namely 15.

lam:cr:dap: thm:str **Theorem 3.2.** *If a relation $\xrightarrow{X}$ satisfies the Church–Rosser property, and $\xrightarrow[X]{}\!\!\!\!\!\rightarrow$ is the smallest transitive relation containing $\xrightarrow{X}$, then $\xrightarrow[X]{}\!\!\!\!\!\rightarrow$ satisfies the Church–Rosser property too.*

Proof. Suppose

$$M \xrightarrow{X} P_1 \xrightarrow{X} \dots \xrightarrow{X} P_m \text{ and}$$
$$M \xrightarrow{X} Q_1 \xrightarrow{X} \dots \xrightarrow{X} Q_n.$$

We will prove the theorem by constructing a grid N of terms of height is $m+1$ and width $n+1$. We use $N_{i,j}$ to denote the term in the i-th row and j-th column.

We construct N in such a way that $N_{i,j} \xrightarrow{X} N_{i+1,j}$ and $N_{i,j} \xrightarrow{X} N_{i,j+1}$. It is defined as follows:

$$\begin{aligned} N_{0,0} &= M \\ N_{i,0} &= P_i && \text{if } 1 \leq i \leq m \\ N_{0,j} &= Q_j && \text{if } 1 \leq j \leq n \end{aligned}$$

and otherwise:

$$N_{i,j} = R$$

where R is a term such that $N_{i-1,j} \xrightarrow{X} R$ and $N_{i,j-1} \xrightarrow{X} R$. By the Church–Rosser property of $\xrightarrow{X}$, such a term always exists.

Now we have $N_{m,0} \xrightarrow{X} \dots \xrightarrow{X} N_{m,n}$ and $N_{0,n} \xrightarrow{X} \dots \xrightarrow{X} N_{m,n}$. Note $N_{m,0}$ is P and $N_{0,n}$ is Q. By definition of $\xrightarrow[X]{}\!\!\!\!\!\rightarrow$ the theorem follows. □

3.2 Parallel β-reduction

lam:cr:pb: sec We introduce the notion of *parallel β-reduction*, and prove the it has the Church–Rosser property.

lam:cr:pb: defn:bredpar **Definition 3.3 (parallel β-reduction, $\overset{\beta}{\Longrightarrow}$).** Parallel reduction ($\overset{\beta}{\Longrightarrow}$) of terms is inductively defined as follows:

lam:cr:pb: defn:bredpar1 1. $x \overset{\beta}{\Longrightarrow} x$.

lam:cr:pb: defn:bredpar2 2. If $N \xrightarrow{\beta} N'$ then $\lambda x.\, N \overset{\beta}{\Longrightarrow} \lambda x.\, N'$.

lam:cr:pb: defn:bredpar3 3. If $P \overset{\beta}{\Longrightarrow} P'$ and $Q \overset{\beta}{\Longrightarrow} Q'$ then $PQ \overset{\beta}{\Longrightarrow} P'Q'$.

lam:cr:pb: defn:bredpar4 4. If $N \overset{\beta}{\Longrightarrow} N'$ and $Q \overset{\beta}{\Longrightarrow} Q'$ then $(\lambda x.\, N)Q \overset{\beta}{\Longrightarrow} N'[Q'/x]$.

Parallel β-reduction allows us to reduce any number of redices in a term in one step. It is different from β-reduction in the sense that we can only contract redices that occur in the original term, but not redices arising from parallel β-reduction. For example, the term $(\lambda f.\, fx)(\lambda y.\, y)$ can only be parallel β-reduced to itself or to $(\lambda y.\, y)x$, but not further to x, although it β-reduces to x, because this redex arises only after one step of parallel β-reduction. A second parallel β-reduction step yields x, though.

Theorem 3.4. $M \overset{\beta}{\Longrightarrow} M$. *lam:cr:pb: thm:refl*

Proof. Exercise. □

Problem 3.1. Prove Theorem 3.4.

Definition 3.5 (β-complete development). The *β-complete development* $M^{*\beta}$ of M is defined inductively as follows: *lam:cr:pb: defn:bcd*

$$x^{*\beta} = x \tag{3.1}$$ *lam:cr:pb: defn:bcd1*

$$(\lambda x.\, N)^{*\beta} = \lambda x.\, N^{*\beta} \tag{3.2}$$ *lam:cr:pb: defn:bcd2*

$$(PQ)^{*\beta} = P^{*\beta}Q^{*\beta} \qquad \text{if } P \text{ is not a } \lambda\text{-abstract} \tag{3.3}$$ *lam:cr:pb: defn:bcd3*

$$((\lambda x.\, N)Q)^{*\beta} = N^{*\beta}[Q^{*\beta}/x] \tag{3.4}$$ *lam:cr:pb: defn:bcd4*

The β-complete development of a term, as its name suggests, is a "complete parallel reduction." While for parallel β-reduction we still can choose to not contract a redex, for complete development we have no choice but to contract all of them. Thus the complete development of $(\lambda f.\, fx)(\lambda y.\, y)$ is $(\lambda y.\, y)x$, not itself.

This definition has the problem that we haven't introduced how to define functions on (λ-)terms recursively. Will fix in future.

Lemma 3.6. *If $M \overset{\beta}{\Longrightarrow} M'$ and $R \overset{\beta}{\Longrightarrow} R'$, then $M[R/y] \overset{\beta}{\Longrightarrow} M'[R'/y]$.* *lam:cr:pb: lem:comp*

Proof. By induction on the derivation of $M \overset{\beta}{\Longrightarrow} M'$.

1. The last step is (1): Exercise.

2. The last step is (2): Then M is $\lambda x.\, N$ and M' is $\lambda x.\, N'$, where $N \overset{\beta}{\Longrightarrow} N'$. We want to prove that $(\lambda x.\, N)[R/y] \overset{\beta}{\Longrightarrow} (\lambda x.\, N')[R'/y]$, i.e., $\lambda x.\, N[R/y] \overset{\beta}{\Longrightarrow} \lambda x.\, N'[R/y]$. This follows immediately by (2) and the induction hypothesis.

3. The last step is (3): Exercise.

4. The last step is (4): M is $(\lambda x.\, N)Q$ and M' is $N'[Q'/x]$. We want to prove that $((\lambda x.\, N)Q)[R/y] \overset{\beta}{\Longrightarrow} N'[Q'/x][R'/y]$, i.e., $(\lambda x.\, N[R/y])Q[R/y] \overset{\beta}{\Longrightarrow} N'[R'/y][Q'[R'/y]/x]$. This follows by (4) and the induction hypothesis. □

Problem 3.2. Complete the proof of Lemma 3.6.

lam:cr:pb: lem:cont **Lemma 3.7.** *If $M \overset{\beta}{\Longrightarrow} M'$ then $M' \overset{\beta}{\Longrightarrow} M^{*\beta}$.*

Proof. By induction on the derivation of $M \overset{\beta}{\Longrightarrow} M'$.

1. The last rule is (1): Exercise.

2. The last rule is (2): M is $\lambda x.\, N$ and M' is $\lambda x.\, N'$ with $N \overset{\beta}{\Longrightarrow} N'$. We want to show that $\lambda x.\, N' \overset{\beta}{\Longrightarrow} (\lambda x.\, N)^{*\beta}$, i.e., $\lambda x.\, N' \overset{\beta}{\Longrightarrow} \lambda x.\, N^{*\beta}$ by eq. (3.2). It follows by (2) and the induction hypothesis.

3. The last rule is (3):M is PQ and M' is $P'Q'$ for some P, Q, P' and Q', with $P \overset{\beta}{\Longrightarrow} P'$ and $Q \overset{\beta}{\Longrightarrow} Q'$. By induction hypothesis, we have $P' \overset{\beta}{\Longrightarrow} P^{*\beta}$ and $Q' \overset{\beta}{\Longrightarrow} Q^{*\beta}$.

   a) If P is $\lambda x.\, N$ for some x and N, then P' must be $\lambda x.\, N'$ for some N' with $N \overset{\beta}{\Longrightarrow} N'$. By induction hypothesis we have $N' \overset{\beta}{\Longrightarrow} N^{*\beta}$ and $Q' \overset{\beta}{\Longrightarrow} Q^{*\beta}$. Then $(\lambda x.\, N')Q' \overset{\beta}{\Longrightarrow} N^{*\beta}[Q^{*\beta}/x]$ by (4).

   b) If P is not a λ-abstract, then $P'Q' \overset{\beta}{\Longrightarrow} P^{*\beta}Q^{*\beta}$ by (3), and the right-hand side is $PQ^{*\beta}$ by eq. (3.3).

4. The last rule is (4): M is $(\lambda x.\, N)Q$ and M' is $N'[Q'/x]$ for some x, N, Q, N', and Q', with $N \overset{\beta}{\Longrightarrow} N'$ and $Q \overset{\beta}{\Longrightarrow} Q'$. By induction hypothesis we know $N' \overset{\beta}{\Longrightarrow} N^{*\beta}$ and $Q' \overset{\beta}{\Longrightarrow} Q^{*\beta}$. By Lemma 3.6 we have $N'[Q'/x] \overset{\beta}{\Longrightarrow} N^{*\beta}[Q^{*\beta}/x]$, the right-hand side of which is exactly $((\lambda x.\, N)Q)^{*\beta}$. □

Problem 3.3. Complete the proof of Lemma 3.7.

lam:cr:pb: thm:cr **Theorem 3.8.** *$\overset{\beta}{\Longrightarrow}$ has the Church–Rosser property.*

Proof. Immediate from Lemma 3.7. □

3.3 β-reduction

lam:cr:b:sec
lam:cr:b:lem:one-par

Lemma 3.9. *If $M \xrightarrow{\beta} M'$, then $M \xRightarrow{\beta} M'$.*

Proof. If $M \xrightarrow{\beta} M'$, then M is $(\lambda x.\, N)Q$, M' is $N[Q/x]$, for some x, N, and Q. Since $N \xRightarrow{\beta} N$ and $Q \xRightarrow{\beta} Q$ by Theorem 3.4, we immediately have $(\lambda x.\, N)Q \xRightarrow{\beta} N[Q/x]$ by Definition 3.3(4). □

lam:cr:b:lem:par-red

Lemma 3.10. *If $M \xRightarrow{\beta} M'$, then $M \xrightarrow{\beta}\!\!\!\!\rightarrow M'$.*

Proof. By induction on the derivation of $M \xRightarrow{\beta} M'$.

1. The last rule is (1): Then M and M' are just x, and $x \xrightarrow{\beta}\!\!\!\!\rightarrow x$.
2. The last rule is (2): M is $\lambda x.\, N$ and M' is $\lambda x.\, N'$ for some x, N, N', where $N \xRightarrow{\beta} N'$. By induction hypothesis we have $N \xrightarrow{\beta}\!\!\!\!\rightarrow N'$. Then $\lambda x.\, N \xrightarrow{\beta}\!\!\!\!\rightarrow \lambda x.\, N'$ (by the same series of $\xrightarrow{\beta}$ contractions as $N \xrightarrow{\beta}\!\!\!\!\rightarrow N'$).
3. The last rule is (3): M is PQ and M' is $P'Q'$ for some P, Q, P', Q', where $P \xRightarrow{\beta} P'$ and $Q \xRightarrow{\beta} Q'$. By induction hypothesis we have $P \xrightarrow{\beta}\!\!\!\!\rightarrow P'$ and $Q \xrightarrow{\beta}\!\!\!\!\rightarrow Q'$. So $PQ \xrightarrow{\beta}\!\!\!\!\rightarrow P'Q'$ by the reduction sequence $P \xrightarrow{\beta}\!\!\!\!\rightarrow P'$ followed by the reduction $Q \xrightarrow{\beta}\!\!\!\!\rightarrow Q'$.
4. The last rule is (4): M is $(\lambda x.\, N)Q$ and M' is $N'[Q'/x]$ for some x, N, M', Q, Q', where $N \xRightarrow{\beta} N'$ and $Q \xRightarrow{\beta} Q'$. By induction hypothesis we get $Q \xrightarrow{\beta}\!\!\!\!\rightarrow Q'$ and $N \xrightarrow{\beta}\!\!\!\!\rightarrow N'$. So $(\lambda x.\, N)Q \xrightarrow{\beta}\!\!\!\!\rightarrow N'[Q'/x]$ by $N \xrightarrow{\beta}\!\!\!\!\rightarrow N'$ followed by $Q \xrightarrow{\beta}\!\!\!\!\rightarrow Q'$ and finally contraction of $(\lambda x.\, N')Q'$ to $N'[Q'/x]$. □

lam:cr:b:lem:str

Lemma 3.11. *$\xrightarrow{\beta}\!\!\!\!\rightarrow$ is the smallest transitive relation containing $\xRightarrow{\beta}$.*

Proof. Let $\xrightarrow{X}\!\!\!\!\rightarrow$ be the smallest transitive relation containing $\xRightarrow{\beta}$.

$\xrightarrow{\beta}\!\!\!\!\rightarrow \subseteq \xrightarrow{X}\!\!\!\!\rightarrow$: Suppose $M \xrightarrow{\beta}\!\!\!\!\rightarrow M'$, i.e., $M \equiv M_1 \xrightarrow{\beta} \ldots \xrightarrow{\beta} M_k \equiv M'$. By Lemma 3.9, $M \equiv M_1 \xRightarrow{\beta} \ldots \xRightarrow{\beta} M_k \equiv M'$. Since is $\xrightarrow{X}\!\!\!\!\rightarrow$ contains $\xRightarrow{\beta}$ and is transitive, $M \xrightarrow{X}\!\!\!\!\rightarrow M'$.

$\xrightarrow{X}\!\!\!\!\rightarrow \subseteq \xrightarrow{\beta}\!\!\!\!\rightarrow$: Suppose $M \xrightarrow{X}\!\!\!\!\rightarrow M'$, i.e., $M \equiv M_1 \xRightarrow{\beta} \ldots \xRightarrow{\beta} M_k \equiv M'$. By Lemma 3.10, $M \equiv M_1 \xrightarrow{\beta}\!\!\!\!\rightarrow \ldots \xrightarrow{\beta}\!\!\!\!\rightarrow M_k \equiv M'$. Since $\xrightarrow{\beta}\!\!\!\!\rightarrow$ is transitive, $M \xrightarrow{\beta}\!\!\!\!\rightarrow M'$. □

lam:cr:b:thm:cr

Theorem 3.12. *$\xrightarrow{\beta}\!\!\!\!\rightarrow$ satisfies the Church–Rosser property.*

Proof. Immediate from Theorem 3.2, Theorem 3.8, and Lemma 3.11. □

3.4 Parallel $\beta\eta$-reduction

lam:cr:pbe:sec

In this section we prove the Church-Rosser property for parallel $\beta\eta$-reduction, the parallel reduction notion corresponding to $\beta\eta$-reduction.

lam:cr:pbe:defn:beredpar

Definition 3.13 (Parallel $\beta\eta$-reduction, $\overset{\beta\eta}{\Longrightarrow}$). *Parallel $\beta\eta$-reduction* ($\overset{\beta\eta}{\Longrightarrow}$) on terms is inductively defined as follows:

lam:cr:pbe:defn:beredpar1
1. $x \overset{\beta\eta}{\Longrightarrow} x$.

lam:cr:pbe:defn:beredpar2
2. If $N \overset{\beta}{\longrightarrow} N'$ then $\lambda x.\, N \overset{\beta\eta}{\Longrightarrow} \lambda x.\, N'$.

lam:cr:pbe:defn:beredpar3
3. If $P \overset{\beta\eta}{\Longrightarrow} P'$ and $Q \overset{\beta\eta}{\Longrightarrow} Q'$ then $PQ \overset{\beta\eta}{\Longrightarrow} P'Q'$.

lam:cr:pbe:defn:beredpar4
4. If $N \overset{\beta\eta}{\Longrightarrow} N'$ and $Q \overset{\beta\eta}{\Longrightarrow} Q'$ then $(\lambda x.\, N)Q \overset{\beta\eta}{\Longrightarrow} N'[Q'/x]$.

lam:cr:pbe:defn:beredpar5
5. If $N \overset{\beta\eta}{\Longrightarrow} N'$ then $\lambda x.\, Nx \overset{\beta\eta}{\Longrightarrow} N'$, provided $x \notin FV(N)$.

lam:cr:pbe:thm:refl

Theorem 3.14. $M \overset{\beta\eta}{\Longrightarrow} M$.

Proof. Exercise. □

Problem 3.4. Prove Theorem 3.14.

lam:cr:pbe:defn:becd

Definition 3.15 ($\beta\eta$-complete development). The *$\beta\eta$-complete development* $M^{*\beta\eta}$ of M is defined as follows:

lam:cr:pbe:defn:becd1
$$x^{*\beta\eta} = x \tag{3.5}$$

lam:cr:pbe:defn:becd2
$$(\lambda x.\, N)^{*\beta\eta} = \lambda x.\, N^{*\beta\eta} \tag{3.6}$$

lam:cr:pbe:defn:becd3
$$(PQ)^{*\beta\eta} = P^{*\beta\eta}Q^{*\beta\eta} \qquad \text{if } P \text{ is not a } \lambda\text{-abstract} \tag{3.7}$$

lam:cr:pbe:defn:becd4
$$((\lambda x.\, N)Q)^{*\beta\eta} = N^{*\beta\eta}[Q^{*\beta\eta}/x] \tag{3.8}$$

lam:cr:pbe:defn:becd5
$$(\lambda x.\, Nx)^{*\beta\eta} = N^{*\beta\eta} \qquad \text{if } x \notin FV(N) \tag{3.9}$$

lam:cr:pbe:lem:comp

Lemma 3.16. *If $M \overset{\beta\eta}{\Longrightarrow} M'$ and $R \overset{\beta\eta}{\Longrightarrow} R'$, then $M[R/y] \overset{\beta\eta}{\Longrightarrow} M'[R'/y]$.*

Proof. By induction on the derivation of $M \overset{\beta\eta}{\Longrightarrow} M'$.

The first four cases are exactly like those in Lemma 3.6. If the last rule is (5), then M is $\lambda x.\, Nx$, M' is N' for some x and N' where $x \notin FV(N)$, and $N \overset{\beta\eta}{\Longrightarrow} N'$. We want to show that $(\lambda x.\, Nx)[R/y] \overset{\beta\eta}{\Longrightarrow} N'[R'/y]$, i.e., $\lambda x.\, N[R/y]x \overset{\beta\eta}{\Longrightarrow} N'[R'/y]$. It follows by Definition 3.13(5) and the induction hypothesis. □

lam:cr:pbe:lem:cont

Lemma 3.17. *If $M \overset{\beta\eta}{\Longrightarrow} M'$ then $M' \overset{\beta\eta}{\Longrightarrow} M^{*\beta\eta}$.*

Proof. By induction on the derivation of $M \overset{\beta\eta}{\Longrightarrow} M'$.

The first four cases are like those in Lemma 3.7. If the last rule is (5), then M is $\lambda x.\, Nx$ and M' is N' for some x, N, N' where $x \notin FV(N)$ and $N \overset{\beta\eta}{\Longrightarrow} N'$. We want to show that $N' \overset{\beta\eta}{\Longrightarrow} (\lambda x.\, Nx)^{*\beta\eta}$, i.e., $N' \overset{\beta\eta}{\Longrightarrow} N^{*\beta\eta}$, which is immediate by induction hypothesis. □

Theorem 3.18. *$\overset{\beta\eta}{\Longrightarrow}$ has the Church-Rosser property.*

lam:cr:pbe:
thm:cr

Proof. Immediate from Lemma 3.17. □

3.5 $\beta\eta$-reduction

The Church–Rosser property holds for $\beta\eta$-reduction ($\overset{\beta\eta}{\longrightarrow\!\!\!\!\!\rightarrow}$).

lam:cr:be:
sec

Lemma 3.19. *If $M \overset{\beta\eta}{\longrightarrow} M'$, then $M \overset{\beta\eta}{\Longrightarrow} M'$.*

lam:cr:be:
lem:one-par

Proof. By induction on the derivation of $M \overset{\beta\eta}{\longrightarrow} M'$. If $M \overset{\beta}{\longrightarrow} M'$ by η-conversion (i.e., Definition 2.43), we use Theorem 3.14. The other cases are as in Lemma 3.9. □

Lemma 3.20. *If $M \overset{\beta\eta}{\Longrightarrow} M'$, then $M \overset{\beta\eta}{\longrightarrow\!\!\!\!\!\rightarrow} M'$.*

lam:cr:be:
lem:par-red

Proof. Induction on the derivation of $M \overset{\beta\eta}{\Longrightarrow} M'$.

If the last rule is (5), then M is $\lambda x.\, Nx$ and M' is N' for some x, N, N' where $x \notin FV(N)$ and $N \overset{\beta\eta}{\Longrightarrow} N'$. Thus we can first reduce $\lambda x.\, Nx$ to N by η-conversion, followed by the series of $\overset{\beta\eta}{\longrightarrow}$ steps that show that $N \overset{\beta\eta}{\longrightarrow\!\!\!\!\!\rightarrow} N'$, which holds by induction hypothesis. □

Lemma 3.21. *$\overset{\beta\eta}{\longrightarrow\!\!\!\!\!\rightarrow}$ is the smallest transitive relation containing $\overset{\beta\eta}{\Longrightarrow}$.*

lam:cr:be:
lem:str

Proof. As in Lemma 3.11 □

Theorem 3.22. *$\overset{\beta\eta}{\longrightarrow\!\!\!\!\!\rightarrow}$ satisfies Church–Rosser property.*

lam:cr:be:
thm:cr

Proof. By Theorem 3.2, Theorem 3.18 and Lemma 3.21. □

Chapter 4

Lambda Definability

This chapter is experimental. It needs more explanation, and the material should be structured better into definitions and propositions with proofs, and more examples.

4.1 Introduction

lam:ldf:int:sec

At first glance, the lambda calculus is just a very abstract calculus of expressions that represent functions and applications of them to others. Nothing in the syntax of the lambda calculus suggests that these are functions of particular kinds of objects, in particular, the syntax includes no mention of natural numbers. Its basic operations—application and lambda abstractions—are operations that apply to any function, not just functions on natural numbers.

Nevertheless, with some ingenuity, it is possible to define arithmetical functions, i.e., functions on the natural numbers, in the lambda calculus. To do this, we define, for each natural number $n \in \mathbb{N}$, a special λ-term $\overline{n}$, the *Church numeral* for n. (Church numerals are named for Alonzo Church.)

Definition 4.1. If $n \in \mathbb{N}$, the corresponding *Church numeral* $\overline{n}$ represents n:

$$\overline{n} \equiv \lambda f x.\, f^n(x)$$

Here, $f^n(x)$ stands for the result of applying f to x n times. For example, $\overline{0}$ is $\lambda f x.\, x$, and $\overline{3}$ is $\lambda f x.\, f(f(f\, x))$.

The Church numeral $\overline{n}$ is encoded as a lambda term which represents a function accepting two arguments f and x, and returns $f^n(x)$. Church numerals are evidently in normal form.

A representation of natural numbers in the lambda calculus is only useful, of course, if we can compute with them. Computing with Church numerals in the lambda calculus means applying a λ-term F to such a Church numeral,

and reducing the combined term $F\,\overline{n}$ to a normal form. If it always reduces to a normal form, and the normal form is always a Church numeral $\overline{m}$, we can think of the output of the computation as being the number m. We can then think of F as defining a function $f\colon \mathbb{N} \to \mathbb{N}$, namely the function such that $f(n) = m$ iff $F\,\overline{n} \twoheadrightarrow \overline{m}$. Because of the Church–Rosser property, normal forms are unique if they exist. So if $F\,\overline{n} \twoheadrightarrow \overline{m}$, there can be no other term in normal form, in particular no other Church numeral, that $F\,\overline{n}$ reduces to.

Conversely, given a function $f\colon \mathbb{N} \to \mathbb{N}$, we can ask if there is a term F that defines f in this way. In that case we say that F *λ-defines* f, and that f is λ-definable. We can generalize this to many-place and partial functions.

Definition 4.2. Suppose $f\colon \mathbb{N}^k \to \mathbb{N}$. We say that a lambda term F *λ-defines* f if for all $n_0, \ldots, n_{k-1}$,

$$F\,\overline{n_0}\,\overline{n_1}\ldots\overline{n_{k-1}} \twoheadrightarrow \overline{f(n_0, n_1, \ldots, n_{k-1})}$$

if $f(n_0, \ldots, n_{k-1})$ is defined, and $F\,\overline{n_0}\,\overline{n_1}\ldots\overline{n_{k-1}}$ has no normal form otherwise.

A very simple example are the constant functions. The term $C_k \equiv \lambda x.\,\overline{k}$ λ-defines the function $c_k\colon \mathbb{N} \to \mathbb{N}$ such that $c(n) = k$. For $C_k\,\overline{n} \equiv (\lambda x.\,\overline{k})\overline{n} \to \overline{k}$ for any n. The identity function is λ-defined by $\lambda x.\,x$. More complex functions are of course harder to define, and often require a lot of ingenuity. So it is perhaps surprising that every computable function is λ-definable. The converse is also true: if a function is λ-definable, it is computable.

4.2 λ-Definable Arithmetical Functions

lam:rep:arf:sec
lam:rep:arf:prop:succ-ld

Proposition 4.3. *The successor function* succ *is λ-definable.*

Proof. A term that λ-defines the successor function is

$$\mathrm{Succ} \equiv \lambda a.\,\lambda f x.\,f(a f x).$$

Given our conventions, this is short for

$$\mathrm{Succ} \equiv \lambda a.\,\lambda f.\,\lambda x.\,(f((a f)x)).$$

Succ is a function that accepts as argument a number a, and evaluates to another function, $\lambda f x.\,f(a f x)$. That function is not itself a Church numeral. However, if the argument a is a Church numeral, it reduces to one. Consider:

$$(\lambda a.\,\lambda f x.\,f(a f x))\,\overline{n} \to \lambda f x.\,f(\overline{n} f x).$$

The embedded term $\overline{n} f x$ is a redex, since $\overline{n}$ is $\lambda f x.\,f^n x$. So $\overline{n} f x \to f^n x$ and so, for the entire term we have

$$\mathrm{Succ}\,\overline{n} \twoheadrightarrow \lambda f x.\,f(f^n(x)),$$

i.e., $\overline{n+1}$. □

Example 4.4. Let's look at what happens when we apply Succ to $\overline{0}$, i.e., $\lambda f x.\, x$. We'll spell the terms out in full:

$$\begin{aligned}\operatorname{Succ}\overline{0} &\equiv (\lambda a.\, \lambda f.\, \lambda x.\, (f((af)x)))(\lambda f.\, \lambda x.\, x)\\ &\to \lambda f.\, \lambda x.\, (f(((\lambda f.\, \lambda x.\, x)f)x))\\ &\to \lambda f.\, \lambda x.\, (f((\lambda x.\, x)x))\\ &\to \lambda f.\, \lambda x.\, (fx) \equiv \overline{1}\end{aligned}$$

Problem 4.1. The term

$$\operatorname{Succ}' \equiv \lambda n.\, \lambda f x.\, nf(fx)$$

λ-defines the successor function. Explain why.

lam:rep:arf:prop:add-ld

Proposition 4.5. *The addition function* add *is λ-definable.*

Proof. Addition is λ-defined by the terms

$$\operatorname{Add} \equiv \lambda ab.\, \lambda f x.\, af(bfx)$$

or, alternatively,

$$\operatorname{Add}' \equiv \lambda ab.\, a \operatorname{Succ} b.$$

The first addition works as follows: Add first accept two numbers a and b. The result is a function that accepts f and x and returns $af(bfx)$. If a and b are Church numerals $\overline{n}$ and $\overline{m}$, this reduces to $f^{n+m}(x)$, which is identical to $f^n(f^m(x))$. Or, slowly:

$$\begin{aligned}(\lambda ab.\, \lambda f x.\, af(bfx))\overline{n}\,\overline{m} &\to \lambda f x.\, \overline{n}\, f(\overline{m}\, fx)\\ &\to \lambda f x.\, \overline{n}\, f(f^m x)\\ &\to \lambda f x.\, f^n(f^m x) \equiv \overline{n+m}.\end{aligned}$$

The second representation of addition Add' works differently: Applied to two Church numerals $\overline{n}$ and $\overline{m}$,

$$\operatorname{Add}'\overline{n}\,\overline{m} \to \overline{n} \operatorname{Succ} \overline{m}.$$

But $\overline{n}fx$ always reduces to $f^n(x)$. So,

$$\overline{n} \operatorname{Succ} \overline{m} \twoheadrightarrow \operatorname{Succ}^n(\overline{m}).$$

And since Succ λ-defines the successor function, and the successor function applied n times to m gives $n+m$, this in turn reduces to $\overline{n+m}$. □

Proposition 4.6. *Multiplication is λ-definable by the term* lam:rep:arf:prop:mult-ld

$$\text{Mult} \equiv \lambda ab.\, \lambda fx.\, a(bf)x$$

Proof. To see how this works, suppose we apply Mult to Church numerals $\overline{n}$ and $\overline{m}$: $\text{Mult}\,\overline{n}\,\overline{m}$ reduces to $\lambda fx.\, \overline{n}(\overline{m}\, f)x$. The term $\overline{m}f$ defines a function which applies f to its argument m times. Consequently, $\overline{n}(\overline{m}f)x$ applies the function "apply f m times" itself n times to x. In other words, we apply f to x, $n \cdot m$ times. But the resulting normal term is just the Church numeral $\overline{nm}$. □

> We can actually simplify this term further by η-reduction:
>
> $$\text{Mult} \equiv \lambda ab.\, \lambda f.\, a(bf).$$
>
> But then we first have to explain η-reduction.

Problem 4.2. Multiplication can be λ-defined by the term

$$\text{Mult}' \equiv \lambda ab.\, a(\text{Add}\, a)\overline{0}.$$

Explain why this works.

The definition of exponentiation as a λ-term is surprisingly simple:

$$\text{Exp} \equiv \lambda be.\, eb.$$

The first argument b is the base and the second e is the exponent. Intuitively, ef is f^e by our encoding of numbers. If you find it hard to understand, we can still define exponentiation also by iterated multiplication:

$$\text{Exp}' \equiv \lambda be.\, e(\text{Mult}\, b)\overline{1}.$$

Predecessor and subtraction on Church numeral is not as simple as we might think: it requires encoding of pairs.

4.3 Pairs and Predecessor

Definition 4.7. The pair of M and N (written $\langle M, N\rangle$) is defined as follows: lam:ldf:pai:sec

$$\langle M, N\rangle \equiv \lambda f.\, fMN.$$

Intuitively it is a function that accepts a function, and applies that function to the two elements of the pair. Following this idea we have this constructor, which takes two terms and returns the pair containing them:

$$\text{Pair} \equiv \lambda mn.\, \lambda f.\, fmn$$

Given a pair, we also want to recover its elements. For this we need two access functions, which accept a pair as argument and return the first or second elements in it:

$$\begin{aligned}\mathrm{Fst} &\equiv \lambda p.\, p(\lambda mn.\, m)\\ \mathrm{Snd} &\equiv \lambda p.\, p(\lambda mn.\, n)\end{aligned}$$

Problem 4.3. Explain why the access functions Fst and Snd work.

Now with pairs we can λ-define the predecessor function:

$$\mathrm{Pred} \equiv \lambda n.\, \mathrm{Fst}(n(\lambda p.\, \langle \mathrm{Snd}\, p, \mathrm{Succ}(\mathrm{Snd}\, p)\rangle)\langle \overline{0}, \overline{0}\rangle)$$

Remember that $\overline{n}\, fx$ reduces to $f^n(x)$; in this case f is a function that accepts a pair p and returns a new pair containing the second component of p and the successor of the second component; x is the pair $\langle 0, 0\rangle$. Thus, the result is $\langle 0, 0\rangle$ for $n = 0$, and $\langle \overline{n-1}, \overline{n}\rangle$ otherwise. Pred then returns the first component of the result.

Subtraction can be defined as Pred applied to a, b times:

$$\mathrm{Sub} \equiv \lambda ab.\, b\mathrm{Pred}\, a.$$

4.4 Truth Values and Relations

lam:ldf:tvr:sec

We can encode truth values in the pure lambda calculus as follows:

$$\begin{aligned}\mathrm{true} &\equiv \lambda x.\, \lambda y.\, x\\ \mathrm{false} &\equiv \lambda x.\, \lambda y.\, y\end{aligned}$$

Truth values are represented as *selectors*, i.e., functions that accept two arguments and returning one of them. The truth value true selects its first argument, and false its second. For example, $\mathrm{true}\, MN$ always reduces to M, while $\mathrm{false}\, MN$ always reduces to N.

Definition 4.8. We call a relation $R \subseteq \mathbb{N}^n$ λ-definable if there is a term R such that

$$R\,\overline{n_1}\ldots\overline{n_k} \xrightarrow{\beta}\!\!\!\!\twoheadrightarrow \mathrm{true}$$

whenever $R(n_1, \ldots, n_k)$ and

$$R\,\overline{n_1}\ldots\overline{n_k} \xrightarrow{\beta}\!\!\!\!\twoheadrightarrow \mathrm{false}$$

otherwise.

For instance, the relation IsZero $= \{0\}$ which holds of 0 and 0 only, is λ-definable by

$$\text{IsZero} \equiv \lambda n.\, n(\lambda x.\, \text{false})\, \text{true}.$$

How does it work? Since Church numerals are defined as iterators (functions which apply their first argument n times to the second), we set the initial value to be true, and for every step of iteration, we return false regardless of the result of the last iteration. This step will be applied to the initial value n times, and the result will be true if and only if the step is not applied at all, i.e., when $n = 0$.

On the basis of this representation of truth values, we can further define some truth functions. Here are two, the representations of negation and conjunction:

$$\text{Not} \equiv \lambda x.\, x\, \text{false}\, \text{true}$$
$$\text{And} \equiv \lambda x.\, \lambda y.\, xy\, \text{false}$$

The function "Not" accepts one argument, and returns true if the argument is false, and false if the argument is true. The function "And" accepts two truth values as arguments, and should return true iff both arguments are true. Truth values are represented as selectors (described above), so when x is a truth value and is applied to two arguments, the result will be the first argument if x is true and the second argument otherwise. Now And takes its two arguments x and y, and in return passes y and false to its first argument x. Assuming x is a truth value, the result will evaluate to y if x is true, and to false if x is false, which is just what is desired.

Note that we assume here that only truth values are used as arguments to And. If it is passed other terms, the result (i.e., the normal form, if it exists) may well not be a truth value.

Problem 4.4. Define the functions Or and Xor representing the truth functions of inclusive and exclusive disjunction using the encoding of truth values as λ-terms.

4.5 Primitive Recursive Functions are λ-Definable

Recall that the primitive recursive functions are those that can be defined from the basic functions zero, succ, and P_i^n by composition and primitive recursion. lam:ldf:prf:sec

Lemma 4.9. *The basic primitive recursive functions* zero, succ, *and projections* P_i^n *are λ-definable.* lam:ldf:prf:lem:basic

Proof. They are λ-defined by the following terms:

$$\text{Zero} \equiv \lambda a.\, \lambda f x.\, x$$
$$\text{Succ} \equiv \lambda a.\, \lambda f x.\, f(a f x)$$
$$\text{Proj}_i^n \equiv \lambda x_0 \ldots x_{n-1}.\, x_i$$

□

lam:ldf:prf: lem:comp **Lemma 4.10.** *Suppose the k-ary function f, and n-ary functions $g_0, \ldots, g_{k-1}$, are λ-definable by terms F, G_0, $\ldots$, G_k, and h is defined from them by composition. Then H is λ-definable.*

Proof. h can be λ-defined by the term

$$H \equiv \lambda x_0 \ldots x_{n-1}.\, F\,(G_0 x_0 \ldots x_{n-1}) \ldots (G_{k-1} x_0 \ldots x_{n-1})$$

We leave verification of this fact as an exercise. □

Problem 4.5. Complete the proof of Lemma 4.10 by showing that $H\overline{n_0} \ldots \overline{n_{n-1}} \twoheadrightarrow \overline{h(n_0, \ldots, n_{n-1})}$.

Note that Lemma 4.10 did not require that f and $g_0, \ldots, g_{k-1}$ are primitive recursive; it is only required that they are total and λ-definable.

lam:ldf:prf: lem:prim **Lemma 4.11.** *Suppose f is an n-ary function and g is an $n+2$-ary function, they are λ-definable by terms F and G, and the function h is defined from f and g by primitive recursion. Then h is also λ-definable.*

Proof. Recall that h is defined by

$$\begin{aligned} h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n, y+1) &= h(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)). \end{aligned}$$

Informally speaking, the primitive recursive definition iterates the application of the function h y times and applies it to $f(x_1, \ldots, x_n)$. This is reminiscent of the definition of Church numerals, which is also defined as a iterator.

For simplicity, we give the definition and proof for a single additional argument x. The function h is λ-defined by:

$$H \equiv \lambda x.\, \lambda y.\, \mathrm{Snd}(y D \langle \overline{0}, F x \rangle)$$

where

$$D \equiv \lambda p.\, \langle \mathrm{Succ}(\mathrm{Fst}\, p), (G x (\mathrm{Fst}\, p)(\mathrm{Snd}\, p)) \rangle$$

The iteration state we maintain is a pair, the first of which is the current y and the second is the corresponding value of h. For every step of iteration we create a pair of new values of y and h; after the iteration is done we return the second part of the pair and that's the final h value. We now prove this is indeed a representation of primitive recursion.

We want to prove that for any n and m, $H\,\overline{n}\,\overline{m} \twoheadrightarrow \overline{h(n,m)}$. To do this we first show that if $D_n \equiv D[\overline{n}/x]$, then $D_n^m \langle \overline{0}, F\,\overline{n} \rangle \twoheadrightarrow \langle \overline{m}, \overline{h(n,m)} \rangle$ We proceed by induction on m.

If $m = 0$, we want $D_n^0 \langle \overline{0}, F\,\overline{n} \rangle \twoheadrightarrow \langle \overline{0}, \overline{h(n,0)} \rangle$. But $D_n^0 \langle \overline{0}, F\,\overline{n} \rangle$ just is $\langle \overline{0}, F\,\overline{n} \rangle$. Since F λ-defines f, this reduces to $\langle \overline{0}, \overline{f(n)} \rangle$, and since $f(n) = h(n,0)$, this is $\langle \overline{0}, \overline{h(n,0)} \rangle$

Now suppose that $D_n^m\langle\overline{0}, F\,\overline{n}\rangle \twoheadrightarrow \langle\overline{m}, \overline{h(n,m)}\rangle$. We want to show that $D_n^{m+1}\langle\overline{0}, F\,\overline{n}\rangle \twoheadrightarrow \langle\overline{m+1}, \overline{h(n,m+1)}\rangle$.

$$\begin{aligned}
D_n^{m+1}\langle\overline{0}, F\,\overline{n}\rangle &\equiv D_n(D_n^m\langle\overline{0}, F\,\overline{n}\rangle)\\
&\twoheadrightarrow D_n\,\langle\overline{m}, \overline{h(n,m)}\rangle \qquad \text{(by IH)}\\
&\equiv (\lambda p.\,\langle \mathrm{Succ}(\mathrm{Fst}\,p), (G\,\overline{n}(\mathrm{Fst}\,p)(\mathrm{Snd}\,p))\rangle)\langle\overline{m}, \overline{h(n,m)}\rangle\\
&\to \langle \mathrm{Succ}(\mathrm{Fst}\,\langle\overline{m}, \overline{h(n,m)}\rangle),\\
&\qquad (G\,\overline{n}(\mathrm{Fst}\,\langle\overline{m}, \overline{h(n,m)}\rangle)(\mathrm{Snd}\,\langle\overline{m}, \overline{h(n,m)}\rangle)\rangle\\
&\twoheadrightarrow \langle \mathrm{Succ}\,\overline{m}, (G\,\overline{n}\,\overline{m}\,\overline{h(n,m)})\rangle\\
&\twoheadrightarrow \langle\overline{m+1}, \overline{g(n,m,h(n,m))}\rangle
\end{aligned}$$

Since $g(n,m,h(n,m)) = h(n,m+1)$, we are done.

Finally, consider

$$\begin{aligned}
H\,\overline{n}\,\overline{m} &\equiv \lambda x.\,\lambda y.\,\mathrm{Snd}(y(\lambda p.\langle \mathrm{Succ}(\mathrm{Fst}\,p), (G\,x\,(\mathrm{Fst}\,p)\,(\mathrm{Snd}\,p))\rangle)\langle\overline{0}, Fx\rangle)\\
&\quad \overline{n}\,\overline{m}\\
&\twoheadrightarrow \mathrm{Snd}(\overline{m}\ \underbrace{(\lambda p.\langle \mathrm{Succ}(\mathrm{Fst}\,p), (G\,\overline{n}\,(\mathrm{Fst}\,p)(\mathrm{Snd}\,p))\rangle)}_{D_n}\langle\overline{0}, F\overline{n}\rangle)\\
&\equiv \mathrm{Snd}(\overline{m}\,D_n\,\langle\overline{0}, F\overline{n}\rangle)\\
&\twoheadrightarrow \mathrm{Snd}\,(D_n^m\langle\overline{0}, F\overline{n}\rangle)\\
&\twoheadrightarrow \mathrm{Snd}\,\langle\overline{m}, \overline{h(n,m)}\rangle\\
&\twoheadrightarrow \overline{h(n,m)}.
\end{aligned}$$

□

Proposition 4.12. *Every primitive recursive function is λ-definable.*

Proof. By Lemma 4.9, all basic functions are λ-definable, and by Lemma 4.10 and Lemma 4.11, the λ-definable functions are closed under composition and primitive recursion. □

4.6 Fixpoints

lam:ldf:fp:sec

Suppose we wanted to define the factorial function by recursion as a term Fac with the following property:

$$\mathrm{Fac} \equiv \lambda n.\,\mathrm{IsZero}\,n\,\overline{1}(\mathrm{Mult}\,n(\mathrm{Fac}(\mathrm{Pred}\,n)))$$

That is, the factorial of n is 1 if $n = 0$, and n times the factorial of $n-1$ otherwise. Of course, we cannot define the term Fac this way since Fac itself occurs in the right-hand side. Such recursive definitions involving self-reference are not part of the lambda calculus. Defining a term, e.g., by

$$\mathrm{Mult} \equiv \lambda ab.\,a(\mathrm{Add}\,a)0$$

only involves previously defined terms in the right-hand side, such as Add. We can always remove Add by replacing it with its defining term. This would give the term Mult as a pure lambda term; if Add itself involved defined terms (as, e.g., Add′ does), we could continue this process and finally arrive at a pure lambda term.

However this is not true in the case of recursive definitions like the one of Fac above. If we replace the occurrence of Fac on the right-hand side with the definition of Fac itself, we get:

$$\begin{aligned}\text{Fac} \equiv \lambda n.\, &\text{IsZero}\, n\, \overline{1} \\ &(\text{Mult}\, n((\lambda n.\, \text{IsZero}\, n\, \overline{1}\, (\text{Mult}\, n\, (\text{Fac}(\text{Pred}\, n))))(\text{Pred}\, n)))\end{aligned}$$

and we still haven't gotten rid of Fac on the right-hand side. Clearly, if we repeat this process, the definition keeps growing longer and the process never results in a pure lambda term. Thus this way of defining factorial (or more generally recursive functions) is not feasible.

The recursive definition does tell us something, though: If f were a term representing the factorial function, then the term

$$\text{Fac}' \equiv \lambda g.\, \lambda n.\, \text{IsZero}\, n\, \overline{1}\, (\text{Mult}\, n\, (g(\text{Pred}n)))$$

applied to the term f, i.e., $\text{Fac}'\, f$, also represents the factorial function. That is, if we regard Fac′ as a function accepting a function and returning a function, the value of $\text{Fac}'\, f$ is just f, provided f is the factorial. A function f with the property that $\text{Fac}'\, f \overset{\beta}{=} f$ is called a *fixpoint* of Fac′. So, the factorial is a fixpoint of Fac′.

There are terms in the lambda calculus that compute the fixpoints of a given term, and these terms can then be used to turn a term like Fac′ into the definition of the factorial.

lam:ldf:fp:defn:Turing-Y

Definition 4.13. The *Y-combinator* is the term:

$$Y \equiv (\lambda ux.\, x(uux))(\lambda ux.\, x(uux)).$$

Theorem 4.14. *Y has the property that $Yg \twoheadrightarrow g(Yg)$ for any term g. Thus, Yg is always a fixpoint of g.*

Proof. Let's abbreviate $(\lambda ux.\, x(uux))$ by U, so that $Y \equiv UU$. Then

$$\begin{aligned}Yg &\equiv (\lambda ux.\, x(uux))U\, g \\ &\twoheadrightarrow (\lambda x.\, x(UUx))g \\ &\twoheadrightarrow g(UUg) \equiv g(Yg).\end{aligned}$$

Since $g(Yg)$ and Yg both reduce to $g(Yg)$, $g(Yg) \overset{\beta}{=} Yg$, so Yg is a fixpoint of g. □

Of course, since Yg is a redex, the reduction can continue indefinitely:

$$\begin{aligned} Yg &\twoheadrightarrow g(Yg) \\ &\twoheadrightarrow g(g(Yg)) \\ &\twoheadrightarrow g(g(g(Yg))) \\ &\dots \end{aligned}$$

So we can think of Yg as g applied to itself infinitely many times. If we apply g to it one additional time, we—so to speak—aren't doing anything extra; g applied to g applied infinitely many times to Yg is still g applied to Yg infinitely many times.

Note that the above sequence of β-reduction steps starting with Yg is infinite. So if we apply Yg to some term, i.e., consider $(Yg)N$, that term will also reduce to infinitely many different terms, namely $(g(Yg))N$, $(g(g(Yg)))N$, ... It is nevertheless possible that some *other* sequence of reduction steps does terminate in a normal form.

Take the factorial for instance. Define Fac as $Y\,\mathrm{Fac}'$ (i.e., a fixpoint of Fac'). Then:

$$\begin{aligned} \mathrm{Fac}\,\overline{3} &\twoheadrightarrow Y\,\mathrm{Fac}'\,\overline{3} \\ &\twoheadrightarrow \mathrm{Fac}'(Y\,\mathrm{Fac}')\,\overline{3} \\ &\equiv (\lambda x.\,\lambda n.\,\mathrm{IsZero}\,n\,\overline{1}\,(\mathrm{Mult}\,n\,(x(\mathrm{Pred}\,n))))\,\mathrm{Fac}\,\overline{3} \\ &\twoheadrightarrow \mathrm{IsZero}\,\overline{3}\,\overline{1}\,(\mathrm{Mult}\,\overline{3}\,(\mathrm{Fac}(\mathrm{Pred}\,\overline{3}))) \\ &\twoheadrightarrow \mathrm{Mult}\,\overline{3}\,(\mathrm{Fac}\,\overline{2}). \end{aligned}$$

Similarly,

$$\begin{aligned} \mathrm{Fac}\,\overline{2} &\twoheadrightarrow \mathrm{Mult}\,\overline{2}\,(\mathrm{Fac}\,\overline{1}) \\ \mathrm{Fac}\,\overline{1} &\twoheadrightarrow \mathrm{Mult}\,\overline{1}\,(\mathrm{Fac}\,\overline{0}) \end{aligned}$$

but

$$\begin{aligned} \mathrm{Fac}\,\overline{0} &\twoheadrightarrow \mathrm{Fac}'(Y\,\mathrm{Fac}')\,\overline{0} \\ &\equiv (\lambda x.\,\lambda n.\,\mathrm{IsZero}\,n\,\overline{1}\,(\mathrm{Mult}\,n\,(x(\mathrm{Pred}\,n))))\,\mathrm{Fac}\,\overline{0} \\ &\twoheadrightarrow \mathrm{IsZero}\,\overline{0}\,\overline{1}\,(\mathrm{Mult}\,\overline{0}\,(\mathrm{Fac}(\mathrm{Pred}\,\overline{0}))). \\ &\twoheadrightarrow \overline{1}. \end{aligned}$$

So together

$$\mathrm{Fac}\,\overline{3} \twoheadrightarrow \mathrm{Mult}\,\overline{3}\,(\mathrm{Mult}\,\overline{2}\,(\mathrm{Mult}\,\overline{1}\,\overline{1})).$$

What goes for Fac' goes for any recursive definition. Suppose we have a recursive equation

$$g\,x_1 \dots x_n \overset{\beta}{=} N$$

where N may contain g and $x_1, \ldots, x_n$. Then there is always a term $G \equiv (Y\lambda g.\, \lambda x_1 \ldots x_n.\, N)$ such that

$$G\, x_1 \ldots x_n \overset{\beta}{=} N[G/g].$$

For by the fixpoint theorem,

$$\begin{aligned} G \equiv (Y\lambda g.\, \lambda x_1 \ldots x_n.\, N) &\twoheadrightarrow \lambda g.\, \lambda x_1 \ldots x_n.\, N(Y\lambda g.\, \lambda x_1 \ldots x_n.\, N) \\ &\equiv (\lambda g.\, \lambda x_1 \ldots x_n.\, N)\, G \end{aligned}$$

and consequently

$$\begin{aligned} G\, x_1 \ldots x_n &\twoheadrightarrow (\lambda g.\, \lambda x_1 \ldots x_n.\, N)\, G\, x_1 \ldots x_n \\ &\twoheadrightarrow (\lambda x_1 \ldots x_n.\, N[G/g])\, x_1 \ldots x_n \\ &\twoheadrightarrow N[G/g]. \end{aligned}$$

The Y combinator of Definition 4.13 is due to Alan Turing. Alonzo Church had proposed a different version which we'll call Y_C:

$$Y_C \equiv \lambda g.\, (\lambda x.\, g(xx))(\lambda x.\, g(xx)).$$

Church's combinator is a bit weaker than Turing's in that $Yg \overset{\beta}{=} g(Yg)$ but not $Yg \overset{\beta}{\twoheadrightarrow} g(Yg)$. Let V be the term $\lambda x.\, g(xx)$, so that $Y_C \equiv \lambda g.\, VV$. Then

$$\begin{aligned} VV \equiv (\lambda x.\, g(xx))V &\twoheadrightarrow g(VV) \text{ and thus} \\ Y_C g \equiv (\lambda g.\, VV)g &\twoheadrightarrow VV \twoheadrightarrow g(VV), \text{ but also} \\ g(Y_C g) \equiv g((\lambda g.\, VV)g) &\twoheadrightarrow g(VV). \end{aligned}$$

In other words, $Y_C g$ and $g(Y_C g)$ reduce to a common term $g(VV)$; so $Y_C g \overset{\beta}{=} g(Y_C g)$. This is often enough for applications.

4.7 Minimization

lam:ldf:min:sec

The general recursive functions are those that can be obtained from the basic functions zero, succ, P_i^n by composition, primitive recursion, and regular minimization. To show that all general recursive functions are λ-definable we have to show that any function defined by regular minimization from a λ-definable function is itself λ-definable.

lam:ldf:min:lem:min

Lemma 4.15. *If $f(x_1, \ldots, x_k, y)$ is regular and λ-definable, then g defined by*

$$g(x_1, \ldots, x_k) = \mu y\ f(x_1, \ldots, x_k, y) = 0$$

is also λ-definable.

Proof. Suppose the lambda term F λ-defines the regular function $f(\vec{x}, y)$. To λ-define h we use a search function and a fixpoint combinator:

$$\begin{aligned} \text{Search} &\equiv \lambda g.\, \lambda f\, \vec{x}\, y.\, \text{IsZero}(f\, \vec{x}\, y)\, y\, (g\, \vec{x}(\text{Succ}\, y)) \\ H &\equiv \lambda \vec{x}.\, (Y\, \text{Search}) F\, \vec{x}\, \overline{0}, \end{aligned}$$

where Y is any fixpoint combinator. Informally speaking, Search is a self-referencing function: starting with y, test whether $f\, \vec{x}\, y$ is zero: if so, return y, otherwise call itself with $\text{Succ}\, y$. Thus $(Y\, \text{Search}) F \overline{n_1} \ldots \overline{n_k}\, \overline{0}$ returns the least m for which $f(n_1, \ldots, n_k, m) = 0$.

Specifically, observe that

$$(Y\, \text{Search}) F \overline{n_1} \ldots \overline{n_k}\, \overline{m} \twoheadrightarrow \overline{m}$$

if $f(n_1, \ldots, n_k, m) = 0$, or

$$\twoheadrightarrow (Y\, \text{Search}) F\, \overline{n_1} \ldots \overline{n_k}\, \overline{m+1}$$

otherwise. Since f is regular, $f(n_1, \ldots, n_k, y) = 0$ for some y, and so

$$(Y\, \text{Search}) F \overline{n_1} \ldots \overline{n_k}\, \overline{0} \twoheadrightarrow \overline{h(n_1, \ldots, n_k)}.$$ □

Proposition 4.16. *Every general recursive function is λ-definable.*

Proof. By Lemma 4.9, all basic functions are λ-definable, and by Lemma 4.10, Lemma 4.11, and Lemma 4.15, the λ-definable functions are closed under composition, primitive recursion, and regular minimization. □

4.8 Partial Recursive Functions are λ-Definable

lam:ldf:par:sec

Partial recursive functions are those obtained from the basic functions by composition, primitive recursion, and unbounded minimization. They differ from general recursive function in that the functions used in unbounded search are not required to be regular. Not requiring regularity means that functions defined by minimization may sometimes not be defined.

At first glance it might seem that the same methods used to show that the (total) general recursive functions are all λ-definable can be used to prove that all partial recursive functions are λ-definable. For instance, the composition of f with g is λ-defined by $\lambda x.\, F(Gx)$ if f and g are λ-defined by terms F and G, respectively. However, when the functions are partial, this is problematic. When $g(x)$ is undefined, meaning Gx has no normal form. In most cases this means that $F(Gx)$ has no normal forms either, which is what we want. But consider when F is $\lambda x.\, \lambda y.\, y$, in which case $F(Gx)$ does have a normal form $(\lambda y.\, y)$.

This problem is not insurmountable, and there are ways to λ-define all partial recursive functions in such a way that undefined values are represented

by terms without a normal form. These ways are, however, somewhat more complicated and less intuitive than the approach we have taken for general recursive functions. We record the theorem here without proof:

Theorem 4.17. *All partial recursive functions are λ-definable.*

4.9 λ-Definable Functions are Recursive

lam:dfl:ldr:sec

Not only are all partial recursive functions λ-definable, the converse is true, too. That is, all λ-definable functions are partial recursive.

lam:dfl:ldr:thm:lambda-computable

Theorem 4.18. *If a partial function f is λ-definable, it is partial recursive.*

Proof. We only sketch the proof. First, we arithmetize λ-terms, i.e., systematially assign Gödel numbers to λ-terms, using the usual power-of-primes coding of sequences. Then we define a partial recursive function normalize(t) operating on the Gödel number t of a lambda term as argument, and which returns the Gödel number of the normal form if it has one, or is undefined otherwise. Then define two partial recursive functions toChurch and fromChurch that maps natural numbers to and from the Gödel numbers of the corresponding Church numeral.

Using these recursive functions, we can define the function f as a partial recursive function. There is a λ-term F that λ-defines f. To compute $f(n_1, \ldots, n_k)$, first obtain the Gödel numbers of the corresponding Church numerals using toChurch(n_i), append these to ${}^{\ulcorner}F^{\urcorner}$ to obtain the Gödel number of the term $F\overline{n_1}\ldots\overline{n_k}$. Now use normalize on this Gödel number. If $f(n_1, \ldots, n_k)$ is defined, $F\overline{n_1}\ldots\overline{n_k}$ has a normal form (which must be a Church numeral), and otherwise it has no normal form (and so

$$\text{normalize}({}^{\ulcorner}F\overline{n_1}\ldots\overline{n_k}{}^{\urcorner})$$

is undefined). Finally, use fromChurch on the Gödel number of the normalized term. □

Photo Credits

Bibliography