Chapter udf

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.

int.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.

int.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.

int.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."

int.4 The Church–Rosser Property

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

Theorem int.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 int.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 int.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.)

rep.5 Currying

lam:rep:curr: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.

int.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 int.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 int.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 int.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.

int.7 λ-Definable Functions are Computable

lam:int:cmp:sec

Theorem int.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. □

int.8 Computable Functions are λ-Definable

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

Theorem int.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 $M = \lambda x.\, \lambda y.\, \lambda z.\, \ldots$."

int.9 The Basic Primitive Recursive Functions are λ-Definable

lam:int:bas:sec

Lemma int.8. *The functions* zero, succ, *and* P^n_i *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: $\mathrm{Proj}^n_i(x_0, \ldots, x_{n-1}) = x_i$. In other words, by our conventions, Proj^n_i is the lambda term $\lambda x_0.\, \ldots \lambda x_{n-1}.\, x_i$.□

int.10 The λ-Definable Functions are Closed under Composition

lam:int:com:sec

Lemma int.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. □

int.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 int.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 int.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.□

int.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."

int.13 The λ-Definable Functions are Closed under Minimization

lam:int:min:sec

Lemma int.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. □

Photo Credits

Bibliography