Chapter udf

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.

ldf.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 ldf.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 represention 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 ldf.2. Suppose $f\colon \mathbb{N}^k \to \mathbb{N}$. We say that a lambda term F *λ-defines* f if for all $n_0, \ldots, n_{k-1}$,

$$F\,\overline{n_0}\,\overline{m_1}\ldots\overline{n_{k-1}} \twoheadrightarrow \overline{f(n_0, n_1, \ldots, n_{k-1})}$$

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.

rep.2 λ-Definable Arithmetical Functions

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

Proposition rep.3. *The successor function* succ *is λ-definable.*

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

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

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., $\overline{n+1}$. □

Problem rep.1. The term

$$\text{Succ}' \equiv \lambda n.\, \lambda f x.\, n f (f x)$$

λ-defines the successor function. Explain why.

lam:rep:arf: prop:add-ld

Proposition rep.4. *The addition function* add *is λ-definable.*

Proof. Addition is λ-defined by the terms

$$\text{Add} \equiv \lambda a b.\, \lambda f x.\, a f (b f x)$$

or, alternatively,

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

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:

$$\begin{aligned}(\lambda a b.\, \lambda f x.\, a f (b f x))\overline{n}\,\overline{m} &\to \lambda f x.\, \overline{n}\, f (\overline{m}\, f x) \\ &\to \lambda f x.\, \overline{n}\, f (f^m x) \\ &\to \lambda f x.\, f^n (f^m x) \equiv \overline{n+m}.\end{aligned}$$

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

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

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

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

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 $\overline{n+m}$. □

lam:rep:arf: prop:mult-ld

Proposition rep.5. *Multiplication is λ-definable by the term*

$$\text{Mult} \equiv \lambda a b.\, \lambda f x.\, a (b f) x$$

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 $\lambda f x.\, \overline{n}(\overline{m}\, f)x$. 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:

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

Problem rep.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.

ldf.3 Pairs and Predecessor

lam:ldf:pai:sec

Definition ldf.6. The pair of M and N (written $\langle M, N\rangle$) is defined as follows:

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

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:

$$\text{Fst} \equiv \lambda p.\, p(\lambda mn.\, m)$$
$$\text{Snd} \equiv \lambda p.\, p(\lambda mn.\, n)$$

Problem ldf.3. Explain why the access functions Fst and Snd work.

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

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

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:

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

ldf.4 Truth Values and Relations

lam:ldf:tvr:sec

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

$$\mathrm{true} \equiv \lambda x.\, \lambda y.\, x$$
$$\mathrm{false} \equiv \lambda x.\, \lambda y.\, y$$

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 ldf.7. We call a relation $R \subseteq \mathbb{N}^n$ λ-definable if there is a term R such that

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

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

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

otherwise.

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

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

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:

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

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 ldf.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.

ldf.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 ldf.8. *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:

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

□

Lemma ldf.9. *Suppose the k-ary function f, and n-ary functions $g_0, \dots, g_{k-1}$, are λ-definable by terms F, G_0, …, G_k, and h is defined from them by composition. Then H is λ-definable* lam:ldf:prf:lem:comp

Proof. h can be λ-defined by the term

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

We leave verification of this fact as an exercise. □

Problem ldf.5. Complete the proof of Lemma ldf.9 by showing that $H\overline{n_0} \dots \overline{n_{n-1}} \twoheadrightarrow \overline{h(n_0, \dots, n_{n-1})}$.

Note that Lemma ldf.9 did not require that f and $g_0, \dots, g_{k-1}$ are primitive recursive; it is only required that they are total and λ-definable.

lam:ldf:prf: lem:prim **Lemma ldf.10.** *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 ldf.11. *Every primitive recursive function is λ-definable.*

Proof. By Lemma ldf.8, all basic functions are λ-definable, and by Lemma ldf.9 and Lemma ldf.10, the λ-definable functions are closed under composition and primitive recursion. □

ldf.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:

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

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

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

applied to the term f, i.e., $\mathrm{Fac}'\, f$, also represents the factorial function. That is, if we regard Fac' as a function accepting a function and returning a function, the value of $\mathrm{Fac}'\, f$ is just f, provided f is the factorial. A function f with the property that $\mathrm{Fac}'\, f \overset{\beta}{=} f$ 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 ldf.12. The *Y-combinator* is the term:

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

Theorem ldf.13. *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:

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

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

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

where N may contain g and x_1, …, 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

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

The Y combinator of Definition ldf.12 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.

ldf.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 ldf.14. *If $f(x_1, \dots, x_k, y)$ is regular and λ-definable, then g defined by*

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

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 Succ y. Thus $(Y\ \text{Search}) F \overline{n_1} \dots \overline{n_k}\, \overline{0}$ returns the least m for which $f(n_1, \dots, n_k, m) = 0$.

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

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

Proposition ldf.15. *Every general recursive function is λ-definable.*

Proof. By Lemma ldf.8, all basic functions are λ-definable, and by Lemma ldf.9, Lemma ldf.10, and Lemma ldf.14, the λ-definable functions are closed under composition, primitive recursion, and regular minimization. □

ldf.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 ldf.16. *All partial recursive functions are λ-definable.*

dfl.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 dfl.17. *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 as 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 lambda 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 ${}^{\ulcorner}F^{\#}$ 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

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

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

Photo Credits

Bibliography