Even though the pure λ-calculus consists only of λ-terms, we can represent and manipulate common data objects like integers, booleans, lists, and trees. All these things can be encoded as λ-terms. We will now see how to encode several common datatypes. There are many reasonable encodings for numerous datatypes. We will explore *Church encodings*, named for Alonzo Church who first proposed them.

1 Booleans

Booleans are the easiest to encode, so we start with them. We would like to define λ-terms representing the constants true and false, and standard logical operators including if-then-else, $\wedge$ (and), $\vee$ (or), and $\neg$ (not) so that all of the above behave in the expected way.

For true and false, we consider two-argument functions, where true returns the first argument and false returns the second. That is,

$$\mathsf{true} \triangleq \lambda xy.\, x \qquad\qquad \mathsf{false} \triangleq \lambda xy.\, y.$$

To make use of these booleans, we need conditional statements. We would like if to take three arguments: a condition b that is a boolean (true or false), and two arbitrary λ-terms t and f. The function should return t whenever $b = \mathsf{true}$ and f whenever $b = \mathsf{false}$. In mathematical notation, that is

$$\mathsf{if} \;=\; \lambda btf.\, \begin{cases} t & \text{if } b = \mathsf{true} \\ f & \text{if } b = \mathsf{false} \end{cases}$$

Now it becomes clear why we defined true and false as above. Since $(\mathsf{true}\ t\ f) \longrightarrow^* t$ and $(\mathsf{false}\ t\ f) \longrightarrow^* f$, all if needs to do is apply its condition to the other two arguments:

$$\mathsf{if} \triangleq \lambda btf.\, b\ t\ f$$

We can then define other boolean operators in terms of if.

$$\mathsf{and} \triangleq \lambda b_1 b_2.\, \mathsf{if}\ b_1\ b_2\ \mathsf{false} \qquad \mathsf{or} \triangleq \lambda b_1 b_2.\, \mathsf{if}\ b_1\ \mathsf{true}\ b_2 \qquad \mathsf{not} \triangleq \lambda b.\, \mathsf{if}\ b\ \mathsf{false}\ \mathsf{true}$$

These operators work correctly when given boolean values as we have defined them, but all bets are off if they are applied to any other λ-term. There is no guarantee of any kind of reasonable behavior. Basically, with the untyped λ-calculus, it is *garbage in, garbage out*.

2 Natural Numbers

As with booleans, we will encode the natural numbers $\mathbb{N}$ as *Church numerals*. The Church numeral for $n \in \mathbb{N}$, which we denote $\overline{n}$, is the λ-term $\lambda fx.\, f^n\ x$ where $f^n = \underbrace{f \circ \cdots \circ f}_{n}$ is the n-fold composition of f with itself.

$$\begin{aligned}
\overline{0} &\triangleq \lambda fx.\, x \\
\overline{1} &\triangleq \lambda fx.\, f\ x \\
\overline{2} &\triangleq \lambda fx.\, f\ (f\ x) \\
&\vdots \\
\overline{n} &\triangleq \lambda fx.\, \underbrace{f\ (f\ (\ \cdots\ (f}_{n}\ x))) = \lambda fx.\, f^n\ x
\end{aligned}$$

Using this approach, we can define the successor function succ by

$$\mathsf{succ} \triangleq \lambda nfx.\, f\ (n\ f\ x).$$

That is, succ takes as input a Church numeral $\overline{n}$ and returns a function of two arguments, f and x, that uses $\overline{n}$ to compute the n-fold composition of f applied to x and then applies f to the result. This function therefore returns the $(n+1)$-fold composition of f applied to x, precisely the definition of $\overline{n+1}$. That is,

$$\begin{aligned}
\mathsf{succ}\ \overline{n} &= (\lambda nfx.\, f\ (n\ f\ x))\ \overline{n} \\
&\longrightarrow \lambda fx.\, f\ (\overline{n}\ f\ x) \\
&\longrightarrow \lambda fx.\, f\ (f^n\ x) \\
&= \lambda fx.\, f^{n+1}\ x \\
&= \overline{n+1}.
\end{aligned}$$

We can also perform basic arithmetic with Chruch numerals. For addition, we might define

$$\mathsf{add} \triangleq \lambda mnfx.\, m\ f\ (n\ f\ x).$$

That is, apply the m-fold composition of f to the result of applying the n-fold composition of f to x, thereby producing the $(m+n)$-fold composition of f applied to x. That is,

$$\begin{aligned}
\mathsf{add}\ \overline{m}\ \overline{n} &= (\lambda mnfx.\, m\ f\ (n\ f\ x))\ \overline{m}\ \overline{n} \\
&\longrightarrow^* \lambda fx.\, \overline{m}\ f\ (\overline{n}\ f\ x) \\
&\longrightarrow^* \lambda fx.\, f^m\ (f^n\ x) \\
&= \lambda fx.\, f^{m+n}\ x \\
&= \overline{m+n}.
\end{aligned}$$

Alternatively, Church numerals act on a function to apply that function repeatedly. Addition can be viewed as repeated application of the successor function, so we could more succinctly define it by combining these two facts:

$$\mathsf{add} \triangleq \lambda mn.\, m\ \mathsf{succ}\ n.$$

We can similarly define multiplication as repeated addition and exponentiation as repeated multiplication.

$$\mathsf{mult} \triangleq \lambda mn.\, m\ (\mathsf{add}\ n)\ \overline{0} \qquad\qquad \mathsf{exp} \triangleq \lambda mn.\, m\ (\mathsf{mult}\ n)\ \overline{1}$$

Interestingly, there is a much simpler (albeit less obvious) form for exponentiation: $\mathsf{exp} \triangleq \lambda mn.\, n\ m$.

3 Pairing and Projection

Logic and arithmetic are good places to start, but we still would like to encode some useful data structures for specifying programs. One simple example is ordered pairs. It would be nice to have a pairing function pair with projections first and second that obey the following equational specifications:

$$\mathsf{first}\ (\mathsf{pair}\ e_1\ e_2) = e_1 \qquad \mathsf{second}\ (\mathsf{pair}\ e_1\ e_2) = e_2 \qquad \mathsf{pair}\ (\mathsf{first}\ p)\ (\mathsf{second}\ p) = p$$

provided p is a pair in the last equation.

We can take a hint from the boolean encodings in Section 1. Recall that if selects one of its two branches by simply applying a carefully-designed boolean argument to those branches. We can have pair do something similar, wrapping its two arguments for later extraction by some function f:

$$\mathsf{pair} \triangleq \lambda abf.\, f\ a\ b.$$

Thus, $\mathsf{pair}\ e_1\ e_2 \longrightarrow^* \lambda f.\, f\ e_1\ e_2$. To get e_1 back out, we can just apply a selector that takes two arguments and returns the first. Interestingly, we already have one: true.

$$(\mathsf{pair}\ e_1\ e_2)\ \mathsf{true} \longrightarrow^* (\lambda f.\, f\ e_1\ e_2)\ \mathsf{true} \longrightarrow \mathsf{true}\ e_1\ e_2 \longrightarrow^* e_1$$

Similarly, applying false extracts e_2. This observation allows us to simple definition of projections:

$$\mathsf{first} \triangleq \lambda p.\, p\ \mathsf{true} \qquad\qquad \mathsf{second} \triangleq \lambda p.\, p\ \mathsf{false}.$$

Again, if p is not of the form $\mathsf{pair}\ a\ b$, all bets are off and this might do anything.

4 Subtraction of Natural Numbers

Defining subtraction for Church-encoded natural numbers is surprisingly difficult. We can define it simply in terms of pred, just as we defined add in terms of succ, but pred itself is nontrivial. Given $\overline{n} = \lambda fx.\, f^n\ x$, there is no general way to construct f^{-1}, so we cannot define it as simply as we defined succ above.

Instead, the insight needed to define pred is to "count" from 0 to n, at each step keeping track of the previous value. When we get to n, we can return the previous value, which will be $n - 1$. More precisely, we will keep track of a pair of Church encodings $\left(\overline{i}, \overline{i-1}\right)$ and count from $i = 0$ to $i = n$.

At each step, we need to increment the pair, which we do with the following next_pair function.

$$\mathsf{next\_pair} \triangleq \lambda p.\, \mathsf{pair}\ (\mathsf{succ}\ (\mathsf{first}\ p))\ (\mathsf{first}\ p)$$

This function makes a pair consisting of the successor of the previous first element—counting up—followed by the previous first element—the old value.

To count from 0 to n, we need to start this pair at $(0, 0)$ and then apply next_pair n times. Luckily, when computing $\mathsf{pred}\ \overline{n}$, we have something that will apply a function n times: $\overline{n}$. We can therefore define

$$\mathsf{pred} \triangleq \lambda n.\, \mathsf{second}\ (n\ \mathsf{next\_pair}\ (\mathsf{pair}\ \overline{0}\ \overline{0})).$$

Note that the right side of the initial pair is only relevant because it defines the output of $\mathsf{pred}\ \overline{0}$. For all other inputs it will be thrown out and ignored. It is often convenient to define $\mathsf{pred}\ \overline{0} = \overline{0}$, so we use that option.

5 Local Variables

One feature common in functional programming languages that appears to be missing from λ-calculus is the ability to define local variables. Languages like OCaml, Haskell, and Lisp all have the ability to bind local variables using a let construct that looks something like this:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

Intuitively, this expression should evaluate e_1 to some value v, then replace all instances of x in e_2 with v, and evaluate the result. In other words, it should evaluate $e_2[x \mapsto v]$. But we already have a λ-term that behaves exactly like this: function application!

$$(\lambda x.\, e_2)\ e_1 \longrightarrow^* (\lambda x.\, e_2)\ v \longrightarrow e_2[x \mapsto v]$$

We can thus view $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as syntactic sugar for $(\lambda x.\, e_2)\ e_1$. This is a very common encoding.

6 Infinite Loops

So far, all of the examples we have seen eventually reach a term that has no β-redices—it cannot be further reduced. These terms are typically referred to as *values* or being in *normal form*.

We have claimed, however, that λ-calculus can represent any computable function—that it is Turing-complete. While the above encodings are all important for showing how it can compute interesting functions, they leave out a very important aspect: we must be able to represent infinite loops in a Turing-complete language. So that raises a question: how can λ-calculus encode an infinite loop?

To see how, we first consider a simple, but slightly odd, λ-term commonly denoted $\omega \triangleq \lambda x.\, x\ x$. This is a function that takes an argument and applies that argument to itself. By applying this function to itself, we construct an expression we call Ω:

$$\Omega \triangleq (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

What happens when we try to evaluate it?

$$\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \longrightarrow (x\ x)[x \mapsto \lambda x.\, x\ x] = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) = \Omega$$

We have just coded an infinite loop!

The term Ω has no value, and it also shows how we can make a function fail to terminate. If we want conditional non-termination, we can, for instance, put Ω in the branch of an if statement.