Even though the pure λ-calculus consists only of λ-terms, we can represent and manipulate common data objects like integers, Boolean values, lists, and trees. All these things can be encoded as λ-terms.

1 Encoding Common Datatypes

1.1 Booleans

The Booleans are the easiest to encode, so let us start with them. We would like to define λ-terms to represent the Boolean constants true and false and the usual Boolean operators $\Rightarrow$ (if-then), $\wedge$ (and), $\vee$ (or), and $\neg$ (not) so that they behave in the expected way. There are many reasonable encodings. One good one is to define true and false by:

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

Now we would like to define a conditional test if. We would like if to take three arguments b, t, f, where b is a Boolean value (either true or false) and t, f are arbitrary λ-terms. The function should return t if $b = \mathsf{true}$ and f if $b = \mathsf{false}$.

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

Now the reason for defining true and false the way we did becomes clear. Since $\mathsf{true}\, t\, f \xrightarrow{1} t$ and $\mathsf{false}\, t\, f \xrightarrow{1} f$, all if has to do is apply its Boolean argument to the other two arguments:

$$\mathsf{if} \quad \triangleq \quad \lambda btf.\, btf$$

The other Boolean operators can be defined from if:

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

Whereas these operators work correctly when given Boolean values as we have defined them, 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.*

1.2 Natural Numbers

We will encode natural numbers $\mathbb{N}$ using *Church numerals.* This is the same encoding that Alonzo Church used, although there are other reasonable encodings. The Church numeral for the number $n \in \mathbb{N}$ is denoted

$\overline{n}$. It is the λ-term $\lambda fx.\, f^n\, x$, where f^n denotes the n-fold composition of f with itself:

$$\begin{array}{rcl} \overline{0} & \triangleq & \lambda fx.\, f^0 x \;=\; \lambda fx.\, x \\ \overline{1} & \triangleq & \lambda fx.\, f^1 x \;=\; \lambda fx.\, fx \\ \overline{2} & \triangleq & \lambda fx.\, f^2 x \;=\; \lambda fx.\, f(fx) \\ \overline{3} & \triangleq & \lambda fx.\, f^3 x \;=\; \lambda fx.\, f(f(fx)) \\ & \vdots & \\ \overline{n} & \triangleq & \lambda fx.\, f^n x \;=\; \lambda fx.\, \underbrace{f(f(\ldots(f}_{n}\, x)\ldots)) \end{array}$$

We can define the successor function succ as

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

That is, succ on input $\overline{n}$ returns a function that takes a function f as input, applies $\overline{n}$ to it to get the n-fold composition of f with itself, then composes that with one more f to get the $(n+1)$-fold composition of f with itself. Then

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

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

$$\mathsf{add} \;\triangleq\; \lambda mnfx.\, mf(nfx).$$

On input $\overline{m}$ and $\overline{n}$, this function returns

$$\begin{array}{rcl} (\lambda mnfx.\, mf(nfx))\,\overline{m}\,\overline{n} & \xrightarrow{1} & \lambda fx.\, \overline{m}f(\overline{n}fx) \\ & \xrightarrow{1} & \lambda fx.\, f^m(f^n x) \\ & = & \lambda fx.\, f^{m+n} x \\ & = & \overline{m+n}. \end{array}$$

Here we are composing f^m with f^n to get f^{m+n}.

Alternatively, recall that Church numerals act on a function to apply that function repeatedly, and addition can be viewed as repeated application of the successor function, so we could define

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

Similarly, multiplication is just iterated addition, and exponentiation is iterated multiplication:

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

1.3 Pairing and Projections

Logic and arithmetic are good places to start, but we still are lacking any useful data structures. For example, consider ordered pairs. It would be nice to have a pairing function pair with projections first and second that obeyed 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. We can take a hint from if. Recall that if selects one of its two branch options depending on its Boolean argument. pair can do something similar, wrapping its two arguments for later extraction by some function f:

$$\mathsf{pair} \;\triangleq\; \lambda abf\,.\,fab.$$

Thus $\mathsf{pair}\,e_1\,e_2 \;\rightarrow\; \lambda f\,.\,fe_1e_2$. To get e_1 back out, we can just apply this to true: $(\lambda f\,.\,fe_1e_2)\,\mathsf{true} \rightarrow \mathsf{true}\,e_1\,e_2 \rightarrow e_1$, and similarly applying it to false extracts e_2. Thus we can define

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

Again, if p is not a term of the form $\mathsf{pair}\,a\,b$, expect the unexpected.

1.4 Lists

One can define lists $[x_1;\,\ldots;\,x_n]$ and list operators corresponding to the OCaml `::`, `List.hd`, and `List.tl` in the λ-calculus. We leave these constructions as exercises.

1.5 Local Variables

One feature that seems to be missing is the ability to declare local variables. For example, in OCaml, we can introduce a new local variable with the let expression:

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

Intuitively, we expect this expression to evaluate e_1 to some value v and then to replace occurrences of x inside e_2 with v. In other words, it should evaluate to $e_2\,\{v/x\}$. But we can construct a λ-term that behaves the same way:

$$(\lambda x\,.\,e_2)\,e_1 \;\rightarrow\; (\lambda x\,.\,e_2)\,v \;\xrightarrow{1}\; e_2\,\{v/x\}.$$

We can thus view a let expression as syntactic sugar for an application of a λ-abstraction.

References

[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 2nd edition, 1984.