

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:

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

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 $b = \mathbf{true}$ and f if $b = \mathbf{false}$.

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

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

$$\mathbf{if} \triangleq \lambda btf. btf$$

The other Boolean operators can be defined from **if**:

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

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

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

$$\begin{aligned}
\bar{0} &\triangleq \lambda f x. f^0 x = \lambda f x. x \\
\bar{1} &\triangleq \lambda f x. f^1 x = \lambda f x. f x \\
\bar{2} &\triangleq \lambda f x. f^2 x = \lambda f x. f(f x) \\
\bar{3} &\triangleq \lambda f x. f^3 x = \lambda f x. f(f(f x)) \\
&\vdots \\
\bar{n} &\triangleq \lambda f x. f^n x = \lambda f x. \underbrace{f(f(\dots(f x)\dots))}_n
\end{aligned}$$

We can define the successor function **succ** as

$$\mathbf{succ} \triangleq \lambda n f x. f (n f x).$$

That is, **succ** on input \bar{n} returns a function that takes a function f as input, applies \bar{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

$$\begin{aligned}
\mathbf{succ} \bar{n} &= (\lambda n f x. f(n f x)) \bar{n} \\
&\xrightarrow{1} \lambda f x. f(\bar{n} f x) \\
&\xrightarrow{1} \lambda f x. f(f^n x) \\
&= \lambda f x. f^{n+1} x \\
&= \overline{n+1}.
\end{aligned}$$

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

$$\mathbf{add} \triangleq \lambda m n f x. m f (n f x).$$

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

$$\begin{aligned}
(\lambda m n f x. m f (n f x)) \bar{m} \bar{n} &\xrightarrow{1} \lambda f x. \bar{m} f (\bar{n} f x) \\
&\xrightarrow{1} \lambda f x. f^m(f^n x) \\
&= \lambda f x. f^{m+n} x \\
&= \overline{m+n}.
\end{aligned}$$

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

$$\mathbf{add} \triangleq \lambda m n. m \mathbf{succ} n.$$

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

$$\mathbf{mul} \triangleq \lambda m n. m(\mathbf{add} n) \bar{0} \qquad \mathbf{exp} \triangleq \lambda m n. m(\mathbf{mul} n) \bar{1}.$$

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:

$$\text{first}(\text{pair } e_1 e_2) = e_1 \quad \text{second}(\text{pair } e_1 e_2) = e_2 \quad \text{pair}(\text{first } p)(\text{second } p) = p,$$

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 :

$$\text{pair} \triangleq \lambda abf. fab.$$

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

$$\text{first} \triangleq \lambda p. p \text{true} \quad \text{second} \triangleq \lambda p. p \text{false}.$$

Again, if p is not a term of the form `pair a b`, expect the unexpected.

1.4 Lists

One can define lists $[x_1; \dots; x_n]$ 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:

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

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.