

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,

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

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 $b = \text{true}$ and f whenever $b = \text{false}$. In mathematical notation, that is

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

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

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

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

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

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 \bar{n} , is the λ -term $\lambda fx. \underbrace{f^n}_n x$ where $f^n = f \circ \dots \circ f$ is the n -fold composition of f with itself.

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

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

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

That is, `succ` takes as input a Church numeral \bar{n} and returns a function of two arguments, f and x , that uses \bar{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,

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

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

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

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,

$$\begin{aligned} \text{add } \bar{m} \bar{n} &= (\lambda m n f x. m f (n f x)) \bar{m} \bar{n} \\ &\rightarrow^* \lambda f x. \bar{m} f (\bar{n} f x) \\ &\rightarrow^* \lambda f x. f^m (f^n x) \\ &= \lambda f x. f^{m+n} x \\ &= \overline{m + n}. \end{aligned}$$

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:

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

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

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

Other useful arithmetic operations and tests are easy to encode and are left as exercises.

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:

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

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 :

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

Thus, $\text{pair } e_1 e_2 \longrightarrow^* \lambda f. f e_1 e_2$. 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.

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

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

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

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

4 Lists

One can define lists $[x_1; \dots; x_n]$ and λ -terms corresponding to the standard list operators cons, head, and tail (or car and cdr, for the Lisps). We leave these constructions as an exercise.

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:

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

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 $\text{let } x = e_1 \text{ in } e_2$ as syntactic sugar for $(\lambda x. e_2) e_1$. This is a very common encoding.