Even though the pure $\lambda$-calculus consists only of $\lambda$-terms, we can represent and manipulate common data objects like integers, Boolean values, lists, and trees. All these things can be encoded as $\lambda$-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 $\lambda$-terms to represent the Boolean constants $true$ and $false$ and the usual Boolean operators $\Rightarrow$ (if-then), $\land$ (and), $\lor$ (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:

\[
false \triangleq \lambda xy. y.
\]

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 $\lambda$-terms. The function should return $t$ if $b = true$ and $f$ if $b = false$.

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

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

\[
if = \lambda btf. btf
\]

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

\[
\begin{align*}
and & \triangleq \lambda b_1 b_2. if b_1 b_2 false & \text{or} & \triangleq \lambda b_1 b_2. if b_1 true b_2 & \text{not} & \triangleq \lambda b_1. if b_1 false true
\end{align*}
\]

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 $\lambda$-term. There is no guarantee of any kind of reasonable behavior. Basically, with the untyped $\lambda$-calculus, it is garbage in, garbage out.

1.2 Natural Numbers

We can 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

\[
\]
\[ \pi. \text{ It is the } \lambda\text{-term } \lambda fx. f^n x, \text{ where } f^n = f \circ \cdots \circ f \text{ denotes the } n\text{-fold composition of } f \text{ with itself:} \]

\[
\begin{align*}
\bar{0} & \triangleq \lambda fx. x \\
\bar{1} & \triangleq \lambda fx. fx \\
\bar{2} & \triangleq \lambda fx. f(fx) \\
\bar{3} & \triangleq \lambda fx. f(f(fx)) \\
& \quad \vdots \\
\bar{n} & \triangleq \lambda fx. f(\cdots(f(x)))_n = \lambda fx. f^n x
\end{align*}
\]

We can define the successor function inc as

\[ \text{inc } \triangleq \lambda fnx. f(nfx). \]

That is, inc 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

\[
\begin{align*}
\text{inc} \bar{n} &= (\lambda fnx. f(nfx)) \bar{n} \\
& \quad \xrightarrow{1} \lambda fx. f(\bar{n}fx) \\
& \quad \xrightarrow{1} \lambda fx. f(f^n x) \\
& \quad = \lambda fx. f^{n+1} x \\
& \quad = \bar{n} + 1.
\end{align*}
\]

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

\[ \text{add } \triangleq \lambda mfnx. mf(nfx). \]

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

\[
\begin{align*}
(\lambda mfnx. mf(nfx)) \bar{m} \bar{n} & \xrightarrow{1} \lambda fx. \bar{m}f(\bar{n}fx) \\
& \quad \xrightarrow{1} \lambda fx. f^{m}(f^n x) \\
& \quad = \lambda fx. f^{m+n} x \\
& \quad = \bar{m} + \bar{n}.
\end{align*}
\]

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

\[ \text{add } \triangleq \lambda mn. m \text{ inc } n. \]

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

\[ \text{mul } \triangleq \lambda mn. m (\text{add } n) \bar{0} \quad \text{exp } \triangleq \lambda mn. m (\text{mul } n) \bar{1}. \]

Other useful arithmetic operations and tests are easily encoded and are left as exercises.
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:

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

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. We can have pair do something similar, wrapping its two arguments for later extraction by some function \( f \):

\[
\text{pair } \triangleq \lambda abf. \ fab.
\]

Thus \( \text{pair } e_1 e_2 \to \lambda f. fe_1e_2 \). To get \( e_1 \) back out, we can just apply this to true: \( (\lambda f. fe_1e_2)\text{true} \to \text{true}e_1e_2 \to e_1 \), and similarly applying it to false extracts \( e_2 \). Thus we can define

\[
\begin{align*}
\text{first } & \triangleq \lambda p. p \text{true} \\
\text{second } & \triangleq \lambda p. p \text{false}.
\end{align*}
\]

Again, if \( p \) is not a term of the form \( \text{pair } a b \), expect the unexpected.

1.4 Lists

One can define lists \([x_1; \ldots; x_n]\) and \(\lambda\)-terms corresponding to the OCaml list operators \texttt{List.cons (:)}, \texttt{List.hd}, and \texttt{List.tl}. 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 \texttt{let} expression:

\[
\text{let } x = e_1 \text{ in } e_2
\]

and the scope of \( x \) is \( e_2 \). 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 there is a \(\lambda\)-term that behaves the same way:

\[
(\lambda x. e_2) e_1 \to (\lambda x. e_2) v \xrightarrow{1} e_2\{v/x\}.
\]

We can thus view the \texttt{let} expression \texttt{let } \( x = e_1 \) \text{ in } \( e_2 \) as syntactic sugar for \( (\lambda x. e_2) e_1 \).

References