1 Encoding boolean logic, basic arithmetic, and data structures

Even though all values in the \(\lambda\)-calculus are functions, it would be nice to somehow have objects which could be worked with like constants, such as integers or boolean values.

1.1 Encoding booleans

We wish to implement functions \(\text{TRUE}\), \(\text{FALSE}\), \(\text{IF}\), \(\text{AND}\), and so forth, such that the expected behavior holds, including statements such as

\[
\begin{align*}
\text{IF} \ \text{TRUE} \ x \ y & \rightarrow x \\
\text{AND} \ \text{TRUE} \ \text{FALSE} & \rightarrow \ \text{FALSE}
\end{align*}
\]

If, for no \(a\) \(priori\) good reason, we define \(\text{TRUE}\) and \(\text{FALSE}\) as:

\[
\begin{align*}
\text{TRUE} & \triangleq \lambda xy. x \\
\text{FALSE} & \triangleq \lambda xy. y
\end{align*}
\]

Then we see we desire to have \(\text{IF}\) be of the form

\[
\text{IF} = \lambda b \ t \ f. \ (if \ b = \text{TRUE} \ then \ t, \ if \ b = \text{FALSE} \ then \ f)
\]

And now the definitions used for the boolean values become useful, because \(\text{TRUE} \ t \ f \rightarrow t\) and \(\text{FALSE} \ t \ f \rightarrow f\), so all we need to do is apply the boolean passed to \(\text{IF}\):

\[
\text{IF} \triangleq \lambda b \ t \ f. \ (b \ t \ f)
\]

With \(\text{IF}\) in hand, defining other boolean operators becomes straightforward (if rather inefficient):

\[
\begin{align*}
\text{AND} & \triangleq \lambda b_1 \ b_2. \ \text{IF} \ (b_1) \ (\text{IF} \ b_2 \ \text{TRUE} \ \text{FALSE}) \ (\text{FALSE}) \\
\text{OR} & \triangleq \lambda b_1 \ b_2. \ \text{IF} \ (b_1) \ (\text{TRUE}) \ (\text{IF} \ b_2 \ \text{TRUE} \ \text{FALSE}) \\
\text{NOT} & \triangleq \lambda b_1. \ \text{IF} \ b_1 \ \text{FALSE} \ \text{TRUE}
\end{align*}
\]

We have no types here, so while the behavior of these operators is clear when they are fed boolean values as we have defined them, they can be applied to any \(\lambda\)-term... though with a good chance of garbage coming out.

1.2 Encoding integers

To encode numbers, we’ll use Church numerals. That is, the number \(n\) represented as a function which, given another function, returns the \(n\)-fold composition of that other function: \(n(f) \rightarrow f^n\). So, for example,

\[
\begin{align*}
0 & \triangleq \lambda f \ x. \ x \quad (\text{since } f^0(x) = x) \\
1 & \triangleq \lambda f \ x. \ f(x) \quad (f^1 = f) \\
2 & \triangleq \lambda f \ x. \ f(f(x)) \quad (f^2(x) = f(f(x))) \\
\text{SUCC} & \triangleq \lambda n. \lambda f \ x. \ f((n)f)x \quad \text{Applying } f \text{ once more}
\end{align*}
\]

With these numbers, we can perform basic arithmetic, such as \(\text{PLUS}\). An obvious approach might be

\[
\text{PLUS} \triangleq \lambda n_1 \ n_2. \ \lambda f \ x. \ (n_2f)((n_1f)x)
\]
Here we are applying $f^{n_2}$ to $f^{n_1}$ to get $f^{n_1+n_2}$. Alternately, recall that numbers (as we have defined them) act on functions to repeatedly apply the function, and addition can be viewed as repeated application of the successor function:

\[ \text{PLUS} \triangleq \lambda n_1 \ n_2. \ (n_1 \ \text{SUCC}) \ n_2 \]

1.3 Data Structures

Logic and arithmetic are good places to start, but we still are lacking any sort of useful data structure. Consider, for example, ordered pairs. It would be nice to have functions CONS, FIRST, and SECOND which obeyed the equational specifications:

\[
\begin{align*}
\text{FIRST}(\text{CONS}(e_1 \ e_2)) &= e_1 \\
\text{SECOND}(\text{CONS}(e_1 \ e_2)) &= e_2 \\
\text{CONS} \ (\text{FIRST} \ p) \ (\text{SECOND} \ p) &= p
\end{align*}
\]

We can begin with CONS, trying to wrap two given values for later use as the arguments to a yet-unused third value, similar to the manner IF wrapped its two branch options for extraction by the appropriate boolean:

\[ \text{CONS} \triangleq \lambda a \ b. \ \lambda f. \ fab \]

To get the first element given to a CONS, we need some function which will take two arguments and return the first. Conveniently, that is exactly what TRUE does, as we defined it (note: other encodings of booleans are not guaranteed to work!). Similarly, CONS $a \ b$ FALSE $= b$. 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 $p$ isn’t a term of the form CONS $a \ b$, expect the unexpected.

2 Recursion and the Y-combinator

With an encoding for IF, we have some control over the flow of a program, but we do not yet have the ability to write a loop, such as a factorial FACT function. In ML, we can write

\[
\text{fun fact}(n) = \text{if} \ n < 2 \ \text{then} \ 1 \ \text{else} \ n \times \text{fact}(n-1)
\]

But how can this be pulled off in the $\lambda$-calculus, where all the functions are anonymous? As a first guess, we might start with the equational specification

\[
\text{FACT} = \lambda n. \text{IF}(< n \ 2) \ (1) \ (* \ n \ ((\text{FACT})(-n \ 1)))
\]

Alas, FACT is just shorthand for the stuff on the RHS, and until we know what needs to be written down for FACT, we can’t write down FACT. Now if only there were a way to remove the recursive call...

2.1 Recursion Removal Trick

Suppose we break up the recursion into two steps. First, make a function FACT’ which says “If I was given my own name, I could do what FACT should”. Thus, if $f = \text{FACT'}$, then we can treat FACT’($f$) as if it were just FACT:

\[
\text{FACT}' \triangleq \lambda f. \ \lambda n. \text{IF}(< n \ 2) \ (1) \ (* \ n \ ((f \ f)(-n \ 1)))
\]
And, since $\text{FACT}'$ $\text{FACT}'$ should behave as we want $\text{FACT}$ to,

$$\text{FACT} \triangleq \text{FACT}' \text{FACT}'$$

We can now see the recursion working:

$$\begin{align*}
\text{FACT}(4) &= (\text{FACT}' \text{FACT}') 4 \\
&= \lambda n. \text{IF}(< n 2) (1) (*) n (\left(\text{FACT}' \text{FACT}'\right)(-n 1)) (4) \\
&= \text{IF}(< 4 2) (1) (*) 4 (\left(\text{FACT}\right)(-4 1)) \\
4 * \text{FACT}(3) &= (* 4 (\text{FACT}(-4 1)))
\end{align*}$$

2.2 Fixed points and the Y-Combinator

For reasons which will soon become apparent, it might be useful to look at the problem of finding $\text{FACT}$ as a problem of finding a fixed point. Note that for the following function, $\text{FACT} = F(\text{FACT}) = (n F)(\text{FACT})$:

$$F = \lambda f. \lambda n. \text{IF}(< n 2) (1) (*) n (\left(f\right)(-n 1))$$

This comes essentially from wrapping the RHS of the equational specification for $\text{FACT}$ in $\lambda \text{FACT}. \text{RHS}$, and could be applied to other recursive functions: that is, every recursive function $R$ can be viewed as the fixed point of an associated function $R'$. So, if we had a function $Y$ which returns a fixed point of a given input, we could find recursive functions $R$ by applying $Y$ to their associated $R'$.

Equationally, we can specify such a $Y$ by

$$Y F = F(Y F)$$

This would give us

$$Y = \lambda F. \lambda x. F(Y F)x$$

While this is, itself, recursive, we can apply the recursion removal trick to get

$$Y \triangleq Y' Y'$$

$$Y' \triangleq \lambda y. \lambda F. \lambda x. (F((y y)F))x$$

This $Y$ is the infamous “$Y$-combinator”: A closed $\lambda$-term (combinator) which finds solutions to recursion situations. It is more commonly written

$$Y \triangleq \lambda f. ((\lambda x. f xx)(\lambda x. f xx))$$

Regardless of how it is written, it can be used to define recursive functions without stepping through particular cases of recursion removal, such as

$$\text{FACT} \triangleq Y \lambda n. \text{IF}(< n 2) (1) (*) n (f(-n 1)))$$