This note provides the following:

- Boolean and IF
- Arithmetic and integers
- Data structures (lists, trees, arrays, cons cells(pairs))
- Recursive functions

Lambda calculus terms can become long. For compactness we will use certain names, as well as multiple arguments, as abbreviation. We will write \textit{name} \equiv e to indicate that \textit{name} is an abbreviation for e. Here are some definitions for names we will use:

\begin{align*}
\text{APPLY\_TO\_FIVE} & \equiv (\lambda f ( f 5)) \\
\text{COMPOSE} & \equiv \lambda (f g) (\lambda x (f (g x))) \\
\text{TWICE} & \equiv (\lambda f (\lambda x (f (f x))))
\end{align*}

Here, \textit{COMPOSE} composes two functions, and \textit{TWICE} returns a function that calls the given function twice. For example:

\[(\text{TWICE INC}) 2 \mapsto 4\]

On the other hand, we can use \textit{COMPOSE} to define the \textit{TWICE}:

\[\text{TWICE} \equiv (\lambda f (\text{COMPOSE} f f))\]

1 Boolean

Lambda Calculus is universal. This means that no primitive boolean type or 'if' statement is needed. We can form them as follows:

\begin{align*}
\text{TRUE} & \equiv (\lambda x (\lambda y x)) \sim (\lambda (x y) x) \\
\text{FALSE} & \equiv (\lambda x (\lambda y y)) \sim (\lambda (x y) y) \\
\text{IF} & \equiv (\lambda (b t f) (b t f))
\end{align*}

So, \textit{TRUE} is a function which takes two arguments and returns the first one, \textit{FALSE} returns the second one and \text{if} \ e_0 \ \text{then} \ e_1 \ \text{else} \ e_2 \ \Rightarrow \ \text{IF} \ e_0 \ e_1 \ e_2. \ Note that call-by-name is important. \ e_1 \ and \ e_2 \ are not evaluated eagerly by \textit{IF}. So it doesn’t necessarily diverge if \ e_1 \ or \ e_2 \ does.

2 Arithmetic

Another data type which we need is natural numbers. We can model the number \(n\) as a function that composes an arbitrary function \(n\) times, like \(n = f \mapsto f^n\). This representation is called Church numerals. Here is the definition:

\begin{align*}
0 & \equiv (\lambda(f x) x) \quad (= \text{FALSE}) \\
1 & \equiv (\lambda(f x) (f x)) \\
2 & \equiv (\lambda(f x) (f (f x))) \\
3 & \equiv (\lambda(f x) (f (f (f x)))) \\
n & \equiv (\lambda(f x) (f (\cdots (f x) \cdots)))
\end{align*}
We can now define operations on integers. \( \text{INC} \) adds one to a number. It’s a function \( f \mapsto f \cdot 1 \). So we have

\[
\text{INC} \equiv \lambda n (\lambda f (\lambda x (f(n,f)x))) \\
+ \equiv \lambda(n_1,n_2)((n_1\text{ INC} n_2)
\]

3 Data structure

We can construct pairs and lists. The pair/list operations are:

\( \text{CONS } x \ y \): construct a list with head \( x \) and tail \( y \)
\( \text{LEFT } x \ y \): return first item in list (or first item in pair)
\( \text{RIGHT } x \ y \): return remainder of list (or second item in pair)

So we have the following equations that any implementation must satisfy:

\[
\text{LEFT} (\text{CONS } x \ y) = x \\
\text{RIGHT} (\text{CONS } x \ y) = y \\
\text{CONS} ((\text{LEFT } p)(\text{RIGHT } p)) = p
\]

Here is one way to implement these operations:

\[
\text{CONS} \equiv (\lambda(x \ y) (\lambda f (f(x \ y))))_p \\
\text{LEFT} \equiv \lambda p (p \ TRUE) \\
\text{RIGHT} \equiv \lambda p (p \ FALSE)
\]

If we use these operations in ways that the equations above do not handle, we get garbage. Consider \( \text{LEFT } 0 \) and it happens to evaluate to identity. Programming using these encodings is error-prone. This is a defect of this style.

4 Define a Recursive Functions

Consider a recursive function which computes the factorial of an integer. By intuition, we will describe \( \text{FACT} \) as:

\[
\text{FACT} \equiv (\lambda n \ \text{IF } (\text{ISZERO } n) \ 1 \ (\times n \ (\text{FACT } (n-1))))
\]

But this is just a description, not a definition. We need to somehow remove the recursion within the definition. We will do this by defining a new function of \( \text{FACT}' \), which will be passed a function \( f \) such that \((f f) n\) to compute the factorial of \( n \).

\[
\text{FACT}' \equiv (\lambda f (\lambda n \ \text{IF } (\text{ISZERO } n) \ 1 \ (\times n \ (f f (n-1)))))
\]

And the actual factorial function we are to define is \( \text{FACT}' \) applied to itself.

\[
\text{FACT} \equiv (\text{FACT}' \ \text{FACT}')
\]

Now the function \( \text{FACT} \) actually works! As an example, let’s see what happens when we evaluate \( \text{FACT } n \):

\[
\text{FACT } n = (\text{FACT}' \ \text{FACT}' \ n) \\
= \lambda n \ \text{IF } (\text{ISZERO } n) \ 1 \ (\times_n (\text{FACT}' \ \text{FACT}' (n-1)))) \\
\]

\[
\text{FACT}(n-1)
\]
5 Recursion Removal Tricks

Now, let’s see what we just did to the *FACT* function to remove recursion. In general, suppose $F = e$, where $e$ mentions $F$, we use a 3-step process to remove the recursion in $F$:

1. Define a new term $F'$ with a parameter $f$;
2. Substitute $(f f)$ for all $F$ to get $F'$:
   $$F' \equiv (\lambda f \ e \{(f f)/F\})$$
3. Replace any external reference to the recursive function $F$ with an application of our new function applied to itself, i.e. $F \equiv F' F'$

6 Abstracting with the Fixed Point Operator

Recall our original recursive description of the factorial function:

$$\text{FACT} = (\lambda n \ \text{IF} \ (\text{ISZERO} \ n) \ 1 \ (\times \ n \ (\text{FACT}(-n\ 1)))$$

This description’s solution is the factorial function. Note that we can simplify this equation by introducing a new function, say *FACTEQN*:

$$\text{FACTEQN} \equiv \lambda f \ (\lambda n \ \text{IF} \ (\text{ISZERO} \ n) \ 1 \ (\times \ n \ (f \ (-n\ 1)))$$

and as a result:

$$\text{FACT} \equiv (\text{FACTEQN} \ \text{FACT})$$

Thus, $\text{FACT}$ is a fixed point of $\text{FACTEQN}$. Suppose we have an operator $\text{FIX}$ that found the fixed point of functions. In other words, for any function $f$,

$$(\text{FIX} \ f) = f(\text{FIX} \ f)$$

So we can define $\text{FIX}$ as:

$$\text{FIX} = (\lambda f \ (f \ (\text{FIX} \ f)))$$

Now we can apply the removal technique we used above to $\text{FIX}$,

$$\text{FIX}' \equiv (\lambda y \ (\lambda f \ (f \ (y \ f))))$$

$\text{FIX} \equiv (\text{FIX}' \ \text{FIX}')$

The traditional form of $\text{FIX}$, which requires call-by-name, is the *Y* combinator:

$$\text{Y} \equiv (\lambda f \ ((\lambda x \ (f \ (x \ x))) \ (\lambda x \ (f \ (x \ x))))$$

Both of these definitions have the defect that they diverge when used in a CBV language. We can address this by noting that we only expect $(\text{FIX} \ f)$ to be extensionally equal to $f(\text{FIX} \ f)$:

$$(\text{FIX} \ f) \ x = f (\text{FIX} \ f) \ x$$

$\text{FIX} = \lambda f \ (\lambda x \ (f \ (\text{FIX} \ f) \ x))$

$\text{FIX}' \equiv \lambda y \ \lambda f \ (\lambda x \ (f \ (y \ f) \ x))$

$\text{FIX} \equiv \text{FIX}' \ \text{FIX}'$

The *Y* combinator can be similarly repaired:

$$\text{Y}_{\text{CBV}} \equiv \lambda f \ ((\lambda x \ (\lambda y \ (f \ (x \ y))) \ (\lambda x \ (\lambda y \ (f \ (x \ y))))$$