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 $NAME \equiv$ e to indicate that $NAME$ is an abbreviation for e. Here are some definitions for names we will use:

$$\begin{aligned}
APPLY_TO_FIVE &\equiv (\lambda\ f\ (\ f\ 5\ )) \\
COMPOSE &\equiv \lambda\ (f\ g)\ (\lambda\ x\ (f\ (g\ x))) \\
TWICE &\equiv (\lambda\ f\ (\lambda\ x\ (f\ (f\ x))))
\end{aligned}$$

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

$$(TWICE\ INC)\ 2 \mapsto^* \ 4$$

On the other hand, we can use $COMPOSE$ to define the $TWICE$:

$$TWICE \equiv (\lambda f (COMPOSE\ f\ f))$$

1 Boolean

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

$$\begin{aligned}
TRUE &\equiv (\lambda x (\lambda y\ x)) \sim (\lambda (x\ y)\ x) \\
FALSE &\equiv (\lambda x (\lambda y\ y)) \sim (\lambda (x\ y)\ y) \\
IF &\equiv \lambda (btf)\ (b\ t\ f)
\end{aligned}$$

So, $TRUE$ is a function which takes two arguments and returns the first one , $FALSE$ returns the second one and $if\ e_0\ then\ e_1\ else\ e_2 \Rightarrow IF\ e_0\ e_1\ e_2$. Note that call-by-name is important. e_1 and e_2 are not evaluated eagerly by 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:

$$\begin{aligned}
0 &\equiv (\lambda (f\ x)\ x) \qquad (= FALSE) \\
1 &\equiv (\lambda (f\ x)\ (f\ x)) \\
2 &\equiv (\lambda (f\ x)\ (f\ (f\ x))) \\
3 &\equiv (\lambda (f\ x) (f (f (f\ x)))) \\
n &\equiv (\lambda (f\ x)\ (f (\cdots (f\ x) \cdots)))
\end{aligned}$$

We can now define operations on integers. *INC* adds one to a number. It's a function $f^n \mapsto f^{n+1}$. So we have

$$\begin{aligned} INC \equiv\ & \lambda n\ (\lambda f\ (\lambda x\ (f(n\ f)\ x))) \\ + \equiv\ & \lambda(n_1\ n_2)\ ((n_1\ INC)\ n_2) \end{aligned}$$

3 Data structure

We can construct pairs and lists. The pair/list operations are:
$(CONS\ x\ y)$: construct a list with head x and tail y
$(LEFT\ x\ y)$: return first item in list (or first item in pair)
$(RIGHT\ x\ y)$: return remainder of list (or second item in pair)

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

$$\begin{aligned} LEFT(CONS\ \ x\ \ y) =\ & x \\ RIGHT(CONS\ \ x\ \ y) =\ & y \\ CONS((LEFT\ \ p)(RIGHT\ \ p)) =\ & p \end{aligned}$$

Here is one way to implement these operations:

$$\begin{aligned} CONS \equiv\ & (\lambda(x\ y)\ (\underbrace{\lambda f\ \ (f(x\ y))}_{p}) \\ LEFT \equiv\ & \lambda p(p\ TRUE) \\ RIGHT \equiv\ & \lambda p(p\ FALSE) \end{aligned}$$

If we use these operations in ways that the equations above do not handle, we get garbage. Consider *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 *FACT* as:

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

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 *FACT'*, which will be passed a function f such that $((f\ \ f)\ n)$ to compute the factorial of n.

$$FACT' \equiv (\lambda f\ \ (\lambda n\ \ IF\ \ (ISZERO\ \ n)\ \ 1\ \ (\times\ \ n\ \ (f\ f\ (-\ n\ \ 1))))$$

And the actual factorial function we are to define is $FACT'$ applied to itself.

$$FACT \equiv (FACT'\ \ FACT')$$

Now the function *FACT* actually works! As an example, let's see what happens when we evaluate $(FACT\ \ n)$:

$$\begin{aligned} FACT\ \ n\ \ =\ \ & (FACT'\ \ FACT'\ \ n) \\ =\ \ & \lambda n\ \ IF\ (\ ISZERO\ \ n)\ \ 1\ \ (\times n\ \ (\underbrace{FACT'\ \ FACT'\ (n-1)}_{FACT(n-1)}))) \end{aligned}$$

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:

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

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

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

and as a result:

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

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

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

So we can define FIX as:

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

Now we can apply the removal technique we used above to *FIX*,

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

The traditional form of FIX, which requires call-by-name, is the Y combinator:

$$Y \equiv (\lambda\ f\ ((\lambda\ x\ (f\ (x\ x))\ (\lambda\ x\ (f\ (x\ x))))))$$

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 ($\mathit{FIX}\ \ f$) to be extensionally equal to $f(\mathit{FIX}\ \ f)$:

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

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

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

$$\mathit{FIX} \equiv \mathit{FIX'}\ \ \mathit{FIX'}$$

The Y combinator can be similarly repaired:

$$Y_{\mathrm{CBV}} \equiv \lambda\ f\ \ ((\lambda\ x\ \ (\lambda\ y\ \ (f\ \ (x\ x)\ \ y)))\ \ (\lambda\ x\ \ (\lambda\ y\ \ (f\ \ (x\ x)\ \ y))))$$