Turing Completeness

We have already seen how surprisingly expressive the Lambda Calculus is. This lab sheet will more rigorously explore this claim. Specifically, we shall show that it is *Turing Complete*, i.e., able to simulate a Turing machine, and consequently, any computable function. Although the following exercises do not constitute a formal proof, it is relatively easy to extend them to one.

– Eddie Jones

1. First, we shall recap the Church encoding of algebraic datatypes through the inductively defined natural numbers:

   $$\begin{aligned}&\textbf{data}\ \mathit{Nat}\\ &\quad = \mathit{Z}\\ &\quad \mid \mathit{S}\ \mathit{Nat}\end{aligned}$$

   The recipe specifies that each *Nat* abstracts over its would-be constructors. So the number two is encoded as the lambda term $\lambda z\ s.\ s\ (s\ z)$. These datatypes are eliminated by supplying "alternative" constructors, the function *isZero* for example, is encoded as $\lambda n.\ n\ \mathit{True}\ (\mathit{const}\ \mathit{False})$.
   a) Define the branching combinator *ifZero*.
   b) Lists may also be inductively defined:

   $$\begin{aligned}&\textbf{data}\ [a]\\ &\quad = [\,]\\ &\quad \mid a : [a]\end{aligned}$$

   i – Show that the *length* function, which collapses a list by applying specific functions as constructors, just like Haskell's *foldr* function, is lambda definable.
   ii – Haskell also allows for infinite lists:

   $$\begin{aligned}&\mathit{repeat} :: a \to [a]\\ &\mathit{repeat}\ x = x : \mathit{repeat}\ x\end{aligned}$$

   Using the Y combinator, show that the *repeat* function is lambda definable.
   iii – Show that the term *length* (*repeat* x) has at least one diverging reduction sequence for all x.

* 2. A pair of infinite lists will serve as the *tape* of our Turing machine, with one cell under the head, and the rest extending to the left and right. This is sometimes referred to as a zipper:

data *Zipper a*
  = *Zipper* [*a*] *a* [*a*]

Note, the portion of the tape to the left of the head is right-to-left.

a) Define the initial tape filled with the lambda term $\overline{0}$. Form this question onwards you may use any previously defined terms such as *repeat*, and Church constants, i.e. $\overline{n}$.

b) As with other inductively defined datatypes, we can *pattern match* on the tape:

```
λt. t          -- The tape
  (λl h r.     -- It's components
    □          -- The continuation
  )
```

i – Our turing machine must be able to navigate it's tape. Define the left and right operations that move the head if it hasn't reached the edge, and does nothing otherwise.

ii – Informally justify why we do not need to consider these edge cases.

** 3. No machine as powerful as a turing machine can be composed of just memory; we also need control. In a Turing Machine this takes the form of a *finite state machine*. We shall use the natural numbers as our states.

a) The natural numbers are not finite, informally justify why this is not a problem when constructing a simulation.

b) We also need to store the current state. Continuing to use the Church encoding, our machine constructor will now take four arguments, the tape to the left, the symbol at the head, the state, and the tape to the right. Define the lambda term that moves left if the state and the current symbol is a zero, and moves right otherwise.

c) Again using the Y combinator, define the lambda term that performs this actions repeatedly.

d) Of course not every machine should loop forever. While the the Y combinator is constructs a simple fixed-point, the recursive call can used conditionally as in the factorial function:

$$fac = Y\ (\lambda f\ n.\ n\ \overline{1}\ (\lambda m.\ \underline{f}\ n * m))$$

Define the machine that moves left until it finds the symbol $\overline{0}$, and then halts, i.e. returns the current state.

*** 4. We have now seen all the components of a turing machine and are now ready to implement an example: the (3-state) busy beaver as

described in the following table. The busy beavers were constructed to be the longest running, but terminating, turing machines with a given number of states. Evaluate at your own risk.

Read	State								
	0			1			2		
	Write	Move	Next	Write	Move	Next	Write	Move	Next
0	1	R	1	1	L	0	1	L	1
1	1	L	2	1	R	1	1	R	HALT