1 Booleans

Recall the untyped encodings of boolean values (chapter 5):

\[
\text{tru} = \lambda t. \lambda f. t \\
\text{fls} = \lambda t. \lambda f. f
\]

Our first task is to assign types to these terms. We’d like the types of \(t\) and \(f\) to be polymorphic; but they have to be the same because \(\text{tru}\) and \(\text{fls}\) must have the same type. Thus, we have:

\[
\text{tru} = \lambda X. \lambda t : X. \lambda f : X. t \\
\text{fls} = \lambda X. \lambda t : X. \lambda f : X. f
\]

And the type is:

\[
C\text{Bool} = \forall X. X \rightarrow X \rightarrow X
\]

As an example, we could use \(\text{tru}\) like this (if we had \(\text{Nat}\) in our type system):

\[
\text{tru}[\text{Nat}] 1 3 \mapsto^* 1
\]

2 Naturals

In the untyped lambda calculus,

\[
0 = \lambda s. \lambda z. z \\
1 = \lambda s. \lambda z. s \ z \\
2 = \lambda s. \lambda z. s \ (s \ z) \\
\ldots
\]

To add types, we need to decide on types for \(s\) and \(z\). \(z\) can have an arbitrary type, so we’ll introduce a type variable \(X\). Since \(s\) will be applied to \(z\), it must be a function that accepts an \(X\): \(X \rightarrow Y\).

As a first attempt then, we have:

\[
0 = \lambda X. \lambda Y. \lambda s : X \rightarrow Y. \lambda z : X. z \\
1 = \lambda X. \lambda Y. \lambda s : X \rightarrow Y. \lambda z : X. s \ z
\]
\[ 2 = \lambda X. \lambda Y. \lambda s : X \to Y. \lambda z : X. s (s z) \]

But this is clearly wrong, because 0 has return type X, while 1 has return type Y, and 2 isn’t typeable at all. We will have to restrict X and Y to be equal:

\[
0 = \lambda X. \lambda s : X \to X. \lambda z : X. z \\
1 = \lambda X. \lambda s : X \to X. \lambda z : X. s z \\
2 = \lambda X. \lambda s : X \to X. \lambda z : X. s (s z)
\]

And the type is:

\[ \text{CNat} = \forall X. (X \to X) \to X \to X \]

As a first example, we can now easily define an IsEven function. (Assuming we have \text{Bools}, \text{not}, and \text{true}; we could alternately use \text{CBool}, \text{tru}, and \text{λb : CBool. λx : X. λf : X. b[X] f t}).

\[ \text{IsEven} = \lambda n : \text{CNat}. n[\text{Bool}] \text{not true} \]

For example,

\[ \text{IsEven} \ 2 \mapsto 2[\text{Bool}] \text{not true} \mapsto \text{not (not true)} \]

Notice that all the recursion of the problem is handled within the definition \text{CNat}, so we don’t need any recursion in the definition of \text{IsEven}.

As a second example, let’s define \text{Plus}. It should have type \text{CNat} \to \text{CNat} \to \text{CNat}, so immediately we have:

\[ \text{Plus} = \lambda x : \text{CNat}. \lambda y : \text{CNat}. \ldots \]

Now we must construct a \text{CNat} as the result value. The type of \text{CNat} (that is, \forall X. (X \to X) \to X \to X) makes coming up with the first part trivial:

\[ \text{Plus} = \lambda X : \text{CNat}. \lambda Y : \text{CNat}. \lambda X. \lambda s : X \to X. \lambda z : X. \ldots \]

Now, what we want is to apply \text{s} \ (x + y) times.

\[ \text{Plus} = \lambda x : \text{CNat}. \lambda y : \text{CNat}. \lambda s : X \to X. \lambda z : X. y[X] s (x[X] s z) \]

Other operations, such as multiplication and exponentiation, can be easily defined in terms of \text{Plus} (we could have started at an even more basic level, and defined addition in terms of \text{Succ}). For example,

\[ \text{Times} = \lambda x : \text{CNat}. \lambda y : \text{CNat}. y[\text{CNat}] (\text{Plus} x) 0 \]

Fun for a rainy day: define multiplication and exponentiation (and subtraction, if it’s especially rainy) in the style of \text{Plus} above—that is, without relying on some other function defined for \text{CNats}.