

Comp 411 Notes - 11 Apr 2005

TYPED CHURCH ENCODINGS IN SYSTEM F

Scribe: Dan Smith
(dlsmith@rice.edu)

1 Booleans

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

$$\mathbf{tru} = \lambda t. \lambda f. t$$

$$\mathbf{fls} = \lambda t. \lambda f. f$$

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

$$\mathbf{tru} = \lambda X. \lambda t : X. \lambda f : X. t$$

$$\mathbf{fls} = \lambda X. \lambda t : X. \lambda f : X. f$$

And the type is:

$$\mathbf{CBool} = \forall X. X \rightarrow X \rightarrow X$$

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

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

2 Naturals

In the untyped lambda calculus,

$$0 = \lambda s. \lambda z. z$$

$$1 = \lambda s. \lambda z. s \ z$$

$$2 = \lambda s. \lambda z. s \ (s \ z)$$

...

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

As a first attempt then, we have:

$$0 = \lambda X. \lambda Y. \lambda s : X \rightarrow Y. \lambda z : X. z$$

$$1 = \lambda X. \lambda Y. \lambda s : X \rightarrow Y. \lambda z : X. s \ z$$

$$2 = \lambda X. \lambda Y. \lambda s : X \rightarrow Y. \lambda z : X. s (s z)$$

...

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:

$$0 = \lambda X. \lambda s : X \rightarrow X. \lambda z : X. z$$

$$1 = \lambda X. \lambda s : X \rightarrow X. \lambda z : X. s z$$

$$2 = \lambda X. \lambda s : X \rightarrow X. \lambda z : X. s (s z)$$

...

And the type is:

$$\mathbf{CNat} = \forall X. (X \rightarrow X) \rightarrow X \rightarrow X$$

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

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

For example,

$$\mathbf{IsEven} 2 \mapsto^* 2[\mathbf{Bool}] \text{ not true} \mapsto^* \text{not (not true)}$$

Notice that all the recursion of the problem is handled within the definition **CNat**, so we don't need any recursion in the definition of **IsEven**.

As a second example, let's define **Plus**. It should have type $\mathbf{CNat} \rightarrow \mathbf{CNat} \rightarrow \mathbf{CNat}$, so immediately we have:

$$\mathbf{Plus} = \lambda x : \mathbf{CNat}. \lambda y : \mathbf{CNat}. \dots$$

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

$$\mathbf{Plus} = \lambda x : \mathbf{CNat}. \lambda y : \mathbf{CNat}. \lambda X. \lambda s : X \rightarrow X. \lambda z : X. \dots$$

Now, what we want is to apply **s** ($x + y$) times.

$$\mathbf{Plus} = \lambda x : \mathbf{CNat}. \lambda y : \mathbf{CNat}. \lambda X. \lambda s : X \rightarrow X. \lambda z : X. y[X] s (x[X] s z)$$

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

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

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