Introduction to Theoretical Computer Science

Tutorial Sheet 6

The first three questions are exercises in working with simple types and recursion. The next two ask you to think about various possible extensions to types. (Everything, and far more, is of course in Pierce.)

(1) What types make the expression $\lambda f.\lambda g.\lambda x.fx(gx)$ well typed? (This expression is known as the S combinator.)

(2) The term $\mathbf{fix}(\lambda x{:}\mathsf{nat}.x)$ is well-typable. What is its type? What is the value of the term?

(3) Here is a recursive function that takes a nat and returns its Church numeral version:

$$\mathbf{letrec}\ \mathsf{church} = \lambda n{:}\mathsf{nat}.\mathbf{if}(=\ n\ 0)(\lambda f.\lambda x.x)(\lambda f.\lambda x.f(\mathsf{church}(-\ n\ 1)fx))\ \mathbf{in}\ \mathsf{church}(2)$$

What types should we give to f and x?

Unsugar the declarations into λ and $\mathbf{fix}$, and evaluate it using our usual call-by-name strategy. Does it evaluate all the way to the expected answer $\lambda f.\lambda x.f(fx)$?

(4) Languages like Haskell and ML allow the creation of 'tagged union' or 'variant' types, such as

```
data IntOrBool = MyInt int | MyBool bool
```

Such types can equally well be added to the simply-typed λ-calculus, using the syntax of your choice. How would you actually do this? What would you need to add?

Now throw type variables into the mix. Suppose we allow ourselves to write *type equations* such as

$$\alpha = \mathsf{Empty}(\mathsf{unit}) \mid \mathsf{Cons}(\mathsf{nat}, \alpha)$$

What is the? a? solution to this equation?

What about

$$\alpha = \mathsf{Cons}(\mathsf{nat}, \alpha) \quad ?$$

(5) Our inability to well-type the fixpoint combinator

$$\mathsf{Y} \stackrel{\mathsf{def}}{=} \lambda F.(\lambda X.F(XX))(\lambda X.F(XX))$$

arose because of the application of X to itself, meaning that the type of X must be the same as the type of its argument.

Now that we've thought about *recursive* types in the last question, we can deal with it. Suppose τ is some type, and we want the F to have type $\tau \to \tau$ – for example, in the definition of the factorial function on slide 78, we want $F{:}\mathsf{nat} \to \mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat})$, so τ is $\mathsf{nat} \to \mathsf{nat}$.

Then we need X to be a function returning a τ; then if the argument type of X is α, we need

$$\alpha = \alpha \to \tau$$

Imagine we have such a type, call it α_τ. Show that we can now well-type the specific version of the Y where $F{:}\tau \to \tau$.

The general way of extending the type system to allow such thing is to add a 'fixpoint' operator at the type level: our α_τ would be written $\mu\alpha.\alpha \to \tau$.

With some more work, we can embed the entire pure untyped λ-calculus into the typed calculus with recursive types: see the end of section 20.1 of Pierce for the details.