CSCI.4430/6969 Programming Languages Lecture Notes

September 4, 2003

1 η-Conversion

Consider the expression

$$(\lambda x.(\lambda x.x^2 \quad x) \quad 3).$$

Using β-reduction, we can take $E = (\lambda x.x^2 \quad x)$ and $M = 3$. In the reduction we only replace the one x that is free in E to get

$$\xrightarrow{\beta} (\lambda x.x^2 \quad 3).$$

We use the symbol $\xrightarrow{\beta}$ to show that we are performing β-reduction on the expression (As another example we may write $\lambda x.x^2 \xrightarrow{\alpha} \lambda y.y^2$ since α-renaming is taking place).

Another type of operation possible on lambda calculus expressions is called η-conversion ("eta"-reduction when applied from left to right). We perform η-reduction using the rule

$$\lambda x.(E \quad x) \xrightarrow{\eta} E.$$

η-reduction can only be applied if E is a lambda expression taking a single argument, and x does not appear free in E.

Starting with the same expression as before, $(\lambda x.(\lambda x.x^2 \quad x) \quad 3)$, we can perform η-reduction to obtain

$$(\lambda x.(\lambda x.x^2 \quad x) \quad 3) \xrightarrow{\eta} (\lambda x.x^2 \quad 3),$$

which gives the same result as β-reduction. In the following example, there is no redex, but we can perform η-reduction.

$$\lambda x.(y \quad x) \xrightarrow{\eta} y.$$

η-conversion can affect termination of expressions in applicative order expression evaluation. For example, the Y reduction combinator has a terminating applicative order form that can be derived from the normal order combinator form by using η-conversion.

2 Currying Combinator

The *currying* combinator takes a function and returns a curried version of the function. For example, it would take as input the `plus` function, which has the type

$$\texttt{plus} : (\mathbf{Z} \times \mathbf{Z}) \to \mathbf{Z}.$$

The type of a function defines what kinds of things the function can receive and what kinds of things it produces as output. In this case `plus` takes two integers ($\mathbf{Z} \times \mathbf{Z}$), and returns an integer. The definition of `plus` in Scheme is

```
(define plus
  (lambda (x y)
    (+ x y)))
```

The currying combinator would then return the curried version of `plus`, called `plusc`, which has the type

$$\texttt{plusc} : \mathbf{Z} \to (\mathbf{Z} \to \mathbf{Z}).$$

Here, `plusc` takes one integer as input and returns a function from the integers to the integers ($\mathbf{Z} \to \mathbf{Z}$). The definition of `plusc` in Scheme is

```
(define plusc
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

The Scheme version of the currying combinator, which we will call `curry`, would work as follows:

`(curry plus)` $\Rightarrow$ `plusc`.

Using the input and output types above, the type of the `curry` function is

$$\texttt{curry} : (\mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}) \to (\mathbf{Z} \to (\mathbf{Z} \to \mathbf{Z})).$$

So the `curry` function should take as input an uncurried function and return a curried function. In Scheme, we can write `curry` as follows:

```
(define curry
  (lambda (f)
    (lambda (x)
      (lambda (y)
        (f x y)))))
```

This may seem very much like the definition of the `plusc` function. But is the `plusc` function a combinator? No, because the function `+` is a free variable. Remember that the definition of a combinator is that it has no free variables. In Scheme, the `+` operator is considered a variable. This operator has a specific behavior, making this code specific, not universal. So it is crucial in the definition of the currying combinator that the `+` function be changed to a generic function `f`, which can be set to any function you like. The `curry` function has no free variables, and therefore is a combinator.

3 Recursion Combinator

The *recursion* combinator allows recursion in lambda expressions. For example, suppose we want to implement a recursive version of the factorial operation,

$$n! = \left\{ \begin{array}{rl} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{array} \right. .$$

We could start by attempting to write a the recursive function f in the lambda calculus (assuming it has been extended with conditionals, and numbers):

$$\begin{array}{rl} f : \lambda g \lambda n (\text{if} & (= \ n \ 0) \\ & 1 \\ & (* \ n \ (g \ (- \ n \ 1)))). \end{array}$$

This function does not work yet because it does not receive a single argument. Before we can input an integer to the function, we must input a function to satisfy g so that the returning function is the desired factorial. Let's call this function X. Looking within the function, we see that the function X must take an integer and return an integer, that is, its type is $\mathbf{Z} \rightarrow \mathbf{Z}$. The function f will return the proper recursive function with the type $\mathbf{Z} \rightarrow \mathbf{Z}$, but only when supplied with the correct function X. Knowing the input and output types of f, we can write the type of f as

$$f : (\mathbf{Z} \rightarrow \mathbf{Z}) \rightarrow (\mathbf{Z} \rightarrow \mathbf{Z}).$$

What we need is a function X that, when applied to f, returns the correct recursive function.

We could try applying f to itself, i.e.

$$(f \quad f).$$

This does not work, because f expects something of type $\mathbf{Z} \rightarrow \mathbf{Z}$, but it is taking another f, which has the complicated type $(\mathbf{Z} \rightarrow \mathbf{Z}) \rightarrow (\mathbf{Z} \rightarrow \mathbf{Z})$. A function that has the correct input type is the identity combinator, $\lambda x.x$. Applying the identity function, we get

$$\begin{array}{rll} (f \quad I) \Rightarrow & \lambda n.(\text{if} & (= \ n \ 0) \\ & & 1 \\ & & (* \ n \ (I \ (- \ n \ 1)))) \\ \Rightarrow & \lambda n.(\text{if} & (= \ n \ 0) \\ & & 1 \\ & & (* \ n \ (- \ n \ 1))), \end{array}$$

which is equivalent to

$$f(n) = \left\{ \begin{array}{rl} 1 & \text{if } n = 0 \\ n * (n-1) & \text{if } n > 0 \end{array} \right. .$$

We need to find the correct expression X such that when X is applied to f, we get the recursive factorial function. It turns out that the X that works is

$$\begin{aligned} X : (\quad & \lambda x.(\lambda g.\lambda n.(\text{if}\ \ (= \ \ n\ \ 0)\ \ 1\ \ (n\ \ (g\ \ (-\ \ n\ \ 1))))\ \ \lambda y.((x\ \ x)\ \ y)) \\ & \lambda x.(\lambda g.\lambda n.(\text{if}\ \ (= \ \ n\ \ 0)\ \ 1\ \ (n\ \ (g\ \ (-\ \ n\ \ 1))))\ \ \lambda y.((x\ \ x)\ \ y))). \end{aligned}$$

Note that this has a structure similar to the non-terminating expression

$$(\lambda x.(x \quad x) \quad \lambda x.(x \quad x)),$$

and explains why the recursive function can keep going.

X can be defined as $(Y \quad f)$ where Y is the recursion combinator,

$$(f \quad X) \Rightarrow (f \quad (Y \quad f)) \Rightarrow (Y \quad f).$$

As an exercise, prove that $(f \quad (Y \quad f)) \Rightarrow (Y \quad f)$.

The recursion combinator that works for an applicative evaluation order is defined as

$$\begin{aligned} Y = \lambda f.(\quad & \lambda x.(f\ \ \lambda y.((x\ \ x)\ \ y)) \\ & \lambda x.(f\ \ \lambda y.((x\ \ x)\ \ y))). \end{aligned}$$

The same combinator that is valid for normal order is

$$\begin{aligned} Y = \lambda f.(\quad & \lambda x.(f\ \ (x\ \ x)) \\ & \lambda x.(f\ \ (x\ \ x))). \end{aligned}$$

How do we get from normal ordering to applicative order? Use η-conversion (that is, η-reduction in reverse). This is an example where η-conversion can have an impact on the termination of an expression. Use η-reduction to get from applicative order back to normal order.

We can define the different combinators in Scheme. See the Scheme script file "1.ss". This file also shows how Scheme allows a much easier way to perform recursion. As an exercise verify the previous results of lambda calculus in Scheme using the script file. The script file also shows how to perform reverse-currying, or currying the arguments in reverse order.