CSCI.4430/6969 Programming Languages
Lecture Notes

January 24, 2005

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-reduction can be considered a program optimization. For example, consider the following Oz definitions:

```
declare Increment = fun {$ X} X+1 end

declare Inc = fun {$ X} {Increment X} end
```

Using η-reduction, we could reduce `{Inc 6}` to `{Increment 6}` avoiding one extra-function call. This compiler optimization is also called *inlining*.

η-conversion can also 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 (see Section 2.2.)

2 Combinators

Any lambda calculus expression with no free variables is called a *combinator*. Because the meaning of a lambda expression is dependent only on the bindings of its free variables, combinators always have the same meaning independently of the context in which they are used.

There are certain combinators that are very useful in lambda calculus:

The *identity* combinator is defined as

$$I = \lambda x.x.$$

It simply returns whatever is given to it. For example

$$(I \quad 5) \Rightarrow (\lambda x.x \quad 5) \Rightarrow 5.$$

The identity combinator in Oz can be written:

```
declare I = fun {$ X} X end
```

Contrast it to, for example, a `Circumference` function:

```
declare Circumference = fun {$ Radius} 2*PI*Radius end
```

The semantics of the `Circumference` function depends on the definitions of `PI` and `*`. It is, therefore, *not* a combinator.

The *application* combinator is

$$App = \lambda f.\lambda x.(f \quad x),$$

and allows you to evaluate a function with an argument. For example

$$\begin{array}{ll} & ((App \quad \lambda x.x^2) \quad 3) \\ \Rightarrow & ((\lambda f.\lambda x.(f \quad x) \quad \lambda x.x^2) \quad 3) \\ \Rightarrow & (\lambda x.(\lambda x.x^2 \quad x) \quad 3) \\ \Rightarrow & (\lambda x.x^2 \quad 3) \\ \Rightarrow & 9. \end{array}$$

The *sequencing* combinator is

$$Seq = \lambda x.\lambda y.(\lambda z.y \quad x)$$

where z is chosen so that it does not appear free in y.

This combinator guarantees that x is evaluated before y, which is important in programs with side-effects. Assuming we had a "display" function sending output to the console, an example is

$$((Seq \quad (\text{display "hello"})) \quad (\text{display "world"}))$$

2.1 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

$$\texttt{Plus} : (\mathbf{Z} \times \mathbf{Z}) \rightarrow \mathbf{Z}.$$

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 Oz is

```
declare Plus =
fun {$ X Y}
   X+Y
end
```

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 Oz is

```
declare PlusC =
fun {$ X}
   fun {$ Y}
      X+Y
   end
end
```

The Oz 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 Oz, we can write `Curry` as follows:

```
declare Curry =
fun {$ F}
   fun {$ X}
      fun {$ Y}
         {F X Y}
      end
   end
end
```

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 Oz, the `+` operator is considered a procedure that is defined in the `Number` module and can be accessed as `Number.‘+‘`. 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.

2.2 Recursion Combinator

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

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases} .$$

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):

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

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 $\mathbf{Z} \to \mathbf{Z}$. The function f will return the proper recursive function with the type $\mathbf{Z} \to \mathbf{Z}$, 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

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

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 $\mathbf{Z} \to \mathbf{Z}$, but it is taking another f, which has the more complex type $(\mathbf{Z} \to \mathbf{Z}) \to (\mathbf{Z} \to \mathbf{Z})$. A function that has the correct input type is the identity combinator, $\lambda x.x$. Applying the identity function, we get

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

which is equivalent to

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

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,

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

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-expansion (that is, η-conversion in reverse). This is an example where η-conversion can have an impact on the termination of an expression.

Exercises

1. η-reduce the following λ-calculus expressions, if possible:
   (a) $\lambda x.(\lambda y.x\,x)$
   (b) $\lambda x.(\lambda y.y\,x)$
2. Use η-reduction to get from the applicative order Y combinator to the normal order Y combinator.
3. Prove that $(f \quad (Y \quad f)) \Rightarrow (Y \quad f)$.