CSCI.4430/6969 Programming Languages
Lecture Notes

September 1st, 2005

2.2 Order of Evaluation

There are different ways to evaluate lambda expressions. The first method is to always fully evaluate the arguments of a function before evaluating the function itself. This order is called *applicative order*. In the expression

$$(\lambda x.x^2 \quad (\lambda x.x + 1 \quad 2)),$$

the argument $(\lambda x.x + 1 \quad 2)$ should be simplified first. The result is

$$\Rightarrow (\lambda x.x^2 \quad 2 + 1) \Rightarrow (\lambda x.x^2 \quad 3) \Rightarrow 3^2 \Rightarrow 9.$$

Another method is to evaluate the left-most redex first. A redex is an expression of the form $(\lambda x.E \quad M)$, on which β -reduction can be performed. This order is called *normal order*. The same expression would be reduced from the outside in, with $E = x^2$ and $M = (\lambda x.x + 1 \quad 2)$. In this case the result is

$$\Rightarrow (\lambda x.x + 1 \quad 2)^2 \Rightarrow (2 + 1)^2 \Rightarrow 9.$$

As you can see, both orders produced the same result. But is this always the case? It turns out that the answer is "no" for certain expressions whose simplification does not terminate. Consider the expression

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

It is easy to see that reducing this expression gives the same expression back, creating an infinite loop. If we consider the expanded expression

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

we find that the two evaluation orders are not equivalent. Using applicative order, the $(\lambda x.(x \quad x) \quad \lambda x.(x \quad x))$ expression must be evaluated first, but this never terminates. If we use normal order, however, we evaluate the entire expression first, with $E = y$ and $M = (\lambda x.(x \quad x) \quad \lambda x.(x \quad x))$. Since there are no x's in E to replace, the result is simply y. It turns out that it is only in these particular non-terminating cases that the two orders may give different results. The *Church-Rosser theorem* (also called the *confluence property* or the *diamond property*) states that if a lambda calculus expression can be evaluated in two different ways and both ways terminate, both ways will yield the same result.

Also, if there is a way for an expression to terminate, using normal order will cause the termination. In other words, normal order is the best if you want to avoid infinite loops. Take as another example the C program

```
int loop() {
  return loop();
}
```

```
int f(int x, int y) {
  return x;
}

int main() {
  return f(3, loop());
}
```

In this case, using applicative order will cause the program to hang, because the second argument `loop()` will be evaluated. Using normal order will terminate because the unneeded `y` variable will never be evaluated.

Though normal order is better in this respect, applicative order is the one used by most programming languages. Why? Consider the function $f(x) = x + x$. To find $f(4/2)$ using normal order, we hold off on evaluating the argument until after placing the argument in the function, so it yields

$$f(4/2) = 4/2 + 4/2 = 2 + 2 = 4,$$

and the division needs to be done twice. If we use applicative order, we get

$$f(4/2) = f(2) = 2 + 2 = 4,$$

which only requires one division. Since applicative order avoids repetitive computations, it is the preferred method of evaluation in most programming languages, where short execution time is critical.

2.3 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

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

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

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

2.3.1 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

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

Here, `PlusC` takes one integer as input and returns a function from the integers to the integers ($\mathbf{Z} \rightarrow \mathbf{Z}$). 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

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

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.

Exercises

3. Write a function composition combinator in the λ-calculus.

4. Define a curried version of `Compose` in Oz, `ComposeC`, without using the `Curry` combinator. (Hint: It should look very similar to the λ-calculus expression from the previous exercise.)