CSCI.4430/6969 Programming Languages
Lecture Notes

August 28, 2003

1 Scheme Basics

In lambda calculus and Scheme, the lines are blurred between functions (or code) and data. For example, we may redefine the multiplication operation:

```
(define mult *)
```

We may then use it just as you would the times operation:

```
> (* 2 3)
6
> (mult 2 3)
6
```

Let's define the increment function in Scheme:

```
(define increment
  (lambda (x)
    (+ x 1)))
```

Note that in Scheme, parentheses are critical. The command

```
> (increment (increment 1))
3
```

is valid, but adding an extra set of parentheses creates an error:

```
> ((increment (increment 1)))
Error! 3 is not a function.
```

This is because the Scheme interpreter always expects the first item after an open parenthesis to be a function. In this case, however, the statement evaluates to `(3)`, then Scheme tries to find the function `3` and fails.

Since Scheme can treat functions as data to other functions, we can create a function that composes two other functions:

```
> (define compose
    (lambda (f g)
      (lambda (x)
        (f (g x)))))
```

Applying this to the increment function, we can create a function that increments by two:

```
> (define inc2 (compose increment increment))
> (inc2 3)
5
```

2 Currying in Scheme

Unlike in lambda calculus, functions in Scheme are allowed to take more than one variable. The process of currying can then be avoided. However, it is possible to use currying in Scheme if you want to. Take for example the definition of a simple addition function that takes two variables:

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

This function is used simply by placing the function name and the two arguments into a set of parentheses

```
> (plus 4 2)
6
```

If we instead want to use currying (that is, limit the number of arguments to one per function), we need to create a function that takes the first argument and returns a function which generates the proper answer when provided with the second argument. In Scheme this is coded by

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

This code more closely resembles the lambda calculus statement

$$\lambda x.\lambda y.x + y$$

that uses currying. To use the Scheme function, we must supply the arguments one at a time. If we just provide the first argument, we obtain a function:

```
> (define plus2 (plusc 2))
```

This function will produce the correct output when supplied with the other argument; that is the function will perform addition by two:

```
> (plus2 3)
5
> (plus2 5)
7
```

You may also supply two arguments to the `plusc` function, one at a time, to produce output:

```
> ((plusc 4) 2)
6
```

3 Free and Bound Variables in Lambda Calculus

The process of simplifying (or β-reducing) in lambda calculus requires clarification. The general rule is to find an expression of the form

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

called a *redex*, and replace the "free" x's in E with M's. A free variable is one that is not bound to a function definition. For example, in the expression

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

the second x is bound to λx, because it is part of the expression defining that function, which is the function $f(x) = x^2$. The final x, however, is not bound to any function definition, so is considered free. Do not be confused by the fact that the variables have the same name. They are in different scopes, so they are totally independent of each other. An equivalent C program could look like this:

```
int f(int x) {
  return x*x;
}

int main() {
  int x;
  ...
  x = x + 1;
  return f(x);
}
```

In this example, the `x` in `f` could have been substituted for `y` or any other variable name without changing the output of the program. In the same way, the lambda expression

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

is identical to the expression

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

since the final x is unbound, or free. To simplify the expression

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

You could let $E = (\lambda x.x^2 \quad x+1)$ and $M = 2$. The only free x in E is the final one so the correct reduction is

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

The x in x^2 is bound, so it is not replaced.

To complicate things further, it is possible when performing β -reduction to inadvertently change a free variable into a bound variable, which changes the meaning of the expression. In the statement

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

the second y is bound to λy and the final y is free. Taking $E = \lambda y.(x \quad y)$ and $M = (y \quad w)$, we could mistakenly arrive at the simplified expression

$$\lambda y.((y \quad w) \quad y).$$

But now both the second and third y's are bound, because they are both a part of of the λy function definition. This is wrong because one of the y's should remain free as it was in the original expression. To get around this, we can change the λy expression to a λz expression

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

which again does not change the meaning of the expression. This process is called α -renaming. Now when we perform the β -reduction, the original two y variables are not confused. The result is

$$\lambda z.((y \quad w) \quad z).$$

Here, the free y remains free.

4 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 expand the expression to

$$(\lambda x.3 \quad (\lambda x.(x \quad x) \quad \lambda x.(x \quad x))),$$

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 $E = 3$ 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 3. 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.

5 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 *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 (in normal-order evaluation) is

$$Seq = \lambda x.\lambda y.\text{ if } (x \quad y \quad y).$$

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

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

The "if" function evaluates the second or third argument only after the first argument has been evaluated. Notice that this combinator only works in normal-order evaluation. Why?