CSCI.4430/6969 Programming Languages
Lecture Notes

January 20, 2005

1 Brief History of Programming Languages

Ada Augusta, the Countess of Lovelace, the daughter of the poet Lord Byron, is attributed as being the first "programmer" ever, using Babbage's Analytical Engine, circa 1843.

Imperative programming languages include: Fortran (Backus 1954) as one of the first "high-level" (compiled) programming languages created at IBM and used for numerical computing, Cobol (Hopper 1960) for business applications, Algol (Naur 1960), BASIC (Kemeny and Kurtz 1963), Pascal (Wirth 1970), C (Kernighan and Ritchie 1972) used to program the UNIX operating system at Bell Labs, and Ada (Whitaker 1979).

Object-oriented programming languages include: Simula (Dahl and Nygaard 1967), Smalltalk (Kay 1980), C++ (Stroustrop 1980), and Java (Gosling 1994) initially created to program appliances and later used and introduced to program dynamic web content.

Actor-oriented programming languages include PLASMA (Hewitt 1975), Scheme (Sussman and Steele 1975), Act (Lieberman 1981), ABCL (Yonezawa 1988), Actalk (Briot 1989), Erlang (Armstrong 1990), E (Miller et al 1998) and SALSA (Varela and Agha 1999).

Functional programming languages include: Lisp (McCarthy 1962), ML (Milner 1980), and Haskell (1990). The most well-known logic programming language is Prolog (Colmerauer and Roussel 1972) created to process natural language (French). Oz (Smolka 1995) is a declarative programming language that can be used to combine multiple paradigms, e.g., functional, object-oriented, and logical programming. Scripting languages include Python (van Rossum 1985), Perl (Wall 1987), and Tcl (Ousterhout 1988).

2 Lambda Calculus

The lambda calculus created by Church and Kleene in the 1930's is at the heart of functional programming languages. It is Turing-complete, that is, any computable function can be expressed and evaluated using the calculus. It is useful to study programming language concepts because of its high level of abstraction. We will briefly motivate the calculus and introduce its syntax and semantics.

The mathematical notation for defining a *function* is with a statement such as

$$f(x) = x^2, \quad f : \mathbf{Z} \rightarrow \mathbf{Z},$$

where $\mathbf{Z}$ is the set of all integers. The first $\mathbf{Z}$ is called the *domain* of the function, or the set of values x can take. The second $\mathbf{Z}$ is called the *range* of the function, or the set containing all possible values of $f(x)$.

Suppose $f(x) = x^2$ and $g(x) = x + 1$. Traditional function *composition* is defined as

$$f \circ g = f\left(g(x)\right).$$

With our functions f and g,

$$f \circ g = f\left(g(x)\right) = f(x+1) = x^2 + 2x + 1.$$

Similarly

$$g \circ f = g\,(f(x)) = g(x^2) = x^2 + 1.$$

Therefore, function composition is not commutative.

In lambda (λ) calculus, we can use a different notation to define these same concepts. To define a function $f(x) = x^2$, we instead write

$$\lambda x.x^2.$$

Similarly for $g(x) = x + 1$ we instead write

$$\lambda x.x + 1.$$

To describe a function *application* such as $f(2) = 4$, we write

$$(\lambda x.x^2 \quad 2) \Rightarrow 2^2 \Rightarrow 4.$$

The *syntax* for lambda calculus expressions is

$$\begin{array}{rll} e ::= & v & \text{– variable} \\ \mid & \lambda v.e & \text{– lambda expression} \\ \mid & (e \quad e) & \text{– procedure call} \end{array}$$

The *semantics* of the lambda calculus, or the way of evaluating or simplifying expressions, is defined by the rule

$$(\lambda x.E \quad M) \Rightarrow E\{M/x\}.$$

The new expression $E\{M/x\}$ can be read as "replace 'fresh' x's in E with M". Informally, a "fresh" x is an x that is not nested inside another lambda expression. We will cover free and bound variable occurrences in detail later on.

For example, in the expression

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

$E = x^2$ and $M = 2$. To evaluate the expression, we replace x's in E with M, to obtain

$$(\lambda x.x^2 \quad 2) \Rightarrow 2^2 \Rightarrow 4.$$

In lambda calculus, all functions may only have one variable. Functions with more than one variable may be expressed as a function of one variable through *currying.* Suppose we have a function of two variables expressed in the normal way

$$h(x, y) = x + y, \quad h : (\mathbf{Z} \times \mathbf{Z}) \rightarrow \mathbf{Z}.$$

With currying, we can input one variable at a time into separate functions. The first function will take the first argument, x, and return a function that will take the second variable, y, and will in turn provide the desired output. To create the same function with currying, let

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

and

$$g : \mathbf{Z} \rightarrow \mathbf{Z}.$$

That is, f maps integers to a function, and g maps integers to integers. The function $f(x)$ returns a function g_x that provides the appropriate result when supplied with y. For example,

$$f(2) = g_2, \quad \text{where} \quad g_2(y) = 2 + y.$$

So

$$f(2)(3) = g_2(3) = 2 + 3 = 5.$$

In lambda calculus this function would be described with currying by

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

For function application, we nest two application expressions

$$((\lambda x.\lambda y.x + y \quad 2) \quad 3).$$

We may then simplify this expression using the semantic rule (also called beta (β) reduction)

$$((\lambda x.\lambda y.x + y \quad 2) \quad 3) \Rightarrow (\lambda y.2 + y \quad 3) \Rightarrow 2 + 3 \Rightarrow 5.$$

The composition operation $\circ$ can itself be considered a function (also called *higher-order* function) that takes two other functions as its input and returns a function as its output; that is if the first function is $\mathbf{Z} \to \mathbf{Z}$ and the second function is also $\mathbf{Z} \to \mathbf{Z}$, then

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

We can also define function composition in lambda calculus. Suppose we want to compose the square function and the increment function, defined as

$$\lambda x.x^2 \quad \text{and} \quad \lambda x.x + 1.$$

We can define function composition as a function itself with currying by

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

Applying two variables to the composition function with currying works the same way as before, except now our variables are functions.

$$\begin{aligned} & ((\lambda f.\lambda g.\lambda x.(f \quad (g \quad x)) \quad \lambda x.x^2) \quad \lambda x.x + 1) \\ \Rightarrow \quad & (\lambda g.\lambda x.(\lambda x.x^2 \quad (g \quad x)) \quad \lambda x.x + 1) \\ \Rightarrow \quad & \lambda x.(\lambda x.x^2 \quad (\lambda x.x + 1 \quad x)). \end{aligned}$$

The resulting function gives the same results as $f(g(x)) = (x + 1)^2$.

2.1 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.

However, things get more complicated. 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.

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

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

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.

Exercises

1. α-convert the outer-most x to y in the following λ-calculus expressions, if possible:
   (a) $\lambda x.(\lambda x.x\,x)$
   (b) $\lambda x.(\lambda x.x\,y)$
2. β-reduce the following λ-calculus expressions, if possible:
   (a) $(\lambda x.\lambda y.(x\,y)\,(y\,w))$
   (b) $(\lambda x.(x\,x)\,\lambda x.(x\,x))$