CS611 Lecture 2 — Lambda calculus encodings — 1 September, 2004

Scribe: Lucja Kot, Hitesh Ballani — Lecturer: Andrew Myers

1 Introduction

The lambda calculus was originally developed in the 1930s before computers. It was developed by mathematicians, trying to come up with a way of writing down functions. One way of describing functions mathematically is via their *extension*. This can be a list of pairs of (input,output) values, or as a graph mapping one domain to the other.

Lambda calculus gives an *intensional* representation—a program, or what you have to do in order to evaluate the function. As an aside, this class is quite a bit about how to get from an intensional representation, an algorithm, to the extension, meaning, or effect of a function. Real programming languages such as Lisp, Scheme, Haskell and ML are very much based on lambda calculus, although there are differences as well.

2 Syntax

The following is the syntax of the λ-calculus. A term is defined as follows:

$$\begin{array}{lcll} e & ::= & x & \textit{(variable)} \\ & | & e_0\ e_1 & \textit{(application)} \\ & | & \lambda x.\, e & \textit{(abstraction)} \end{array}$$

In an abstraction, x is the *argument*, e is the *body* of the function. Notice that this function doesn't have a name. In mathematics it is common to write $f(x) = e$ to define a function named f. A corresponding anonymous notation that is frequently used is $x \mapsto x^2$; this is really the same thing as $\lambda x.\, x^2$.

Here are some examples of terms. First, we present (and define) the identity function:

$$ID = \lambda x.\, x$$

The next example is a function that will ignore its argument and return the identity function.

$$\lambda x.\, \lambda a.\, a$$

This is the same as

$$\lambda x.\, ID$$

2.1 Closed terms

An occurrence of a variable x in a term is said to be *bound* if there is an enclosing $\lambda x.\, e$; otherwise, it is *free*. A *closed term* is one in which all identifiers are bound; we call such terms *combinators*. We will consider a *program* in the lambda calculus to be any closed term.

For example, consider the following term:

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

Both occurrences of x are bound, the first occurence of y is bound, the a is free, and the last y is also free, since it is outside the scope of the λy.

If a program has some variables that are free, then you do not have a complete program as you do not know what to do with the free variables. Hence, a well formed program in lambda calculus is going to be closed.

2.2 Higher-order functions

In lambda calculus, we can define *higher-order functions*. These are functions that can take functions as arguments and/or return functions as results. In fact, every argument is a function and every result is a function. That is, functions are first-class values.

This example takes a function as an argument and applies it to 5:

$$\lambda f.\,(f\ 5)$$

We can further generalize. This function takes an argument v and returns a function that calls its argument on v:

$$\lambda v.\,\lambda f.\,(f\ v)$$

2.3 Multi-argument functions and currying

In the syntax given above, we only define functions as taking a single argument. Why don't we need to have multiple arguments? We could imagine extending the syntax as follows:

$$\begin{array}{rlll} e & ::= \ldots & \lambda x_1 \ldots x_n.\, e & \text{(multi-argument abstraction)} \\ & \mid & e_0\ e_1\ e_2\ \ldots\ e_n & \text{(multi-argument application)} \end{array}$$

In the multi-argument application, e_0 is a n-argument function and $e_1 \ldots e_n$ are the arguments.

It turns out this isn't any more expressive than basic lambda calculus. We may actually write down terms that look like the above in this course on occasion, but it's just *syntactic sugar*. It is a little easier for the programmer to read but can be transformed into something more basic that does not require the extended syntax. A transformation that removes syntactic sugar is called *desugaring*.

How do we desugar multi-argument functions?

$$\lambda x_1 \ldots x_n.\ e \Rightarrow \lambda x_1.\,\lambda x_2.\,\lambda x_n.\ e$$
$$e_0\ e_1\ \ldots\ e_n \Rightarrow (\ldots((e_0\ e_1)\ e_2)\ \ldots e_n)$$

Note that application is *left-associative* - that is, $e_1\ e_2\ e_3$ is parsed as $(e_1\ e_2)\ e_3$.

3 Semantics

3.1 β-reductions

We would like to believe that the expression $(\lambda x.\, e)\ v$ is 'equal' in some sense to the body of e where we have replaced x with v—we want to call/invoke/apply the function e with v as its argument. The substitution is not as simple as it seems: not all xs should be replaced with v, just the xs bound to the outermost x in our expression. The notation for this *syntactic substitution* is as follows: $e\{v/x\}$. Using this notation, we introduce the β-reduction:

$$(\lambda x.\, e)\ v \longrightarrow e\{v/x\}$$

We say that the two terms above are β-equivalent.

The β-reduction is an example of a *rewrite rule*, a rule that says you can take one piece of syntax and replace it with another piece of syntax.

We will see shortly how to perform computations using β-reduction. However, we will deal first with the important issue of choosing an order in which to apply the reductions. There are several options for this.

3.2 CBV (call-by-value) semantics

In call-by-value semantics, all expressions are evaluated before being passed as function arguments. Formally, we will require that function arguments be reduced to values before the function is processed.

What is a value? A value is a closed term that can no longer be reduced in any way. Syntactically, all values are of the form $\lambda x.\, e$. In the semantic descriptions that follow, we use v to denote any possible value.

Here are the CBV structural operational semantics (SOS):

$$\begin{array}{lccc} & \beta - reduction & \text{evaluation steps} & \\ \text{Call-by-value SOS:} & \dfrac{}{(\lambda x.\, e)v \longrightarrow e\{v/x\}} & \dfrac{e_0 \longrightarrow e_0'}{e_0\ e_1 \longrightarrow e_0'\ e_1} & \dfrac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \end{array}$$

$$v ::= (\lambda x.\, e)$$

The fact that there are nothing at the top in the first rule implies that it has no preconditions for application and hence, can always be applied assuming that the expression looks like left-hand side of the rewrite (this is called the *redex*). In contrast, the second and third rules specify in what order beta reductions should be done. For example, the second rule says that a lambda program of the form $e_0\ e_1$ steps to $e_0'\ e_1$ if e_0 steps to e_0'. In other words, evaluation can proceed by rewriting the left side of an application expression.

Here is an example of a computation using these rules. Let *SUCC* be a function which takes a number and returns the next larger number (we will see how to define such a function later). Start with the following expression:

$$(\lambda x.\, (\lambda\ y\ (y\ x\ ))\ 3)\ SUCC$$

After we apply one β-reduction (using the second rule and the β reduction rule), we get:

$$(\lambda\ y\ (y\ 3))SUCC$$

Applying another β-reduction directly yields:

$$SUCC\ 3$$

Finally, we compute the result of *SUCC* applied to 3:

$$4$$

3.3 Infinite loops and motivation for CBN (call-by-name) semantics

Consider the term Ω, defined as follows:

$$\Omega = (\lambda x.\, (x\ x))\ (\lambda x.\, (x\ x))$$

If we try β-reducing Ω, we simply get Ω back again. Mathematically: $\Omega \longrightarrow \Omega$. Using β-reductions, the program Ω will never terminate; we have an infinite loop.

Now consider what happens if we define a term F that discards its argument:

$$F = \lambda x.\, \lambda y.\, y$$

What happens if we try to evaluate $F\Omega$ using CBV semantics? Since CBV always requires us to evaluate the argument to a function first, we will have to begin with Ω. What happens is another infinite loop: $F\Omega \longrightarrow F\Omega$. Intuitively, this corresponds to a programming situation where a method p calls a method q and uses the output of q in computation. If q goes into an infinite loop, so will p.

The problem with the $F\Omega$ example, however, is that the infinite-loop behavior seems counterintuitive; after all, F is a term that should always return the identity function *ID*, no matter what argument x it receives. The problem arises because we are forced by the CBV semantics to evaluate Ω first. But this is a constraint imposed by the CBV semantics; in another model, such as CBN (call-by-name), things can look different.

3.4 CBN (call-by-name) semantics

Unlike CBV, CBN semantics performs *lazy evaluation*, in which unevaluated expressions (i.e., computations) are passed around to functions. Evaluation is delayed as long as possible. The rewrite rules for CBN are:

$$\begin{array}{lcc} & \beta - reduction & \text{evaluation step} \\ \text{Call-by-name SOS:} & \dfrac{}{(\lambda x.\, e_0)e_1 \longrightarrow e_0\{e_1/x\}} & \dfrac{e_0 \;\longrightarrow\; e_0'}{e_0\; e_1 \;\longrightarrow\; e_0'\; e_1} \end{array}$$

Note that applying CBN semantics to $F\Omega$ as defined above yields *ID*, as expected.

Most programming languages (ML, for instance) are call-by-value because laziness is difficult to implement efficiently.

4 Defining common programming language functions in lambda calculus

We now show how to encode some familiar programming-language consructs in lambda calculus.

4.1 Booleans

We define the boolean functions *TRUE* and *FALSE* as follows (note that we use syntactic sugaring to write the functions directly as two-argument functions).

$$TRUE = \lambda xy.\, x = \lambda x.\, \lambda y.\, x \text{ (desugared version)}$$
$$FALSE = \lambda xy.\, y = \lambda x.\, \lambda y.\, y \text{ (desugared version)}$$

We also define the additional *IF* expression as follows:

$$IF = \lambda xyz.\, xyz = \lambda x.\, \lambda y.\, \lambda z.\, (xy)z$$

We can check that this function behaves as expected by checking, for instance, that

$$IF\ TRUE\ e_1\ e_2 \longrightarrow\ e_1$$

We can define boolean functions, for example AND:

$$AND = \lambda b_1 b_2. IF\ b_1 (IF\ b_2\ TRUE\ FALSE)\ FALSE$$

4.2 Natural numbers

We define natural numbers in lambda calculus using *Church numerals*, named for Alonzo Church. The intuition behind Church numerals is to represent a number n as a function that transforms any function f into the function f^n. We proceed as follows:

$$\begin{array}{ll} 0 \triangleq \lambda f.\, \lambda x.\, x & (\textit{ID} \text{ is } f^0!) \\ 1 \triangleq \lambda f.\, \lambda x.\, fx & \\ 2 \triangleq \lambda f.\, \lambda x.\, f(fx) & \\ 3 \triangleq \lambda f.\, \lambda x.\, f(f(fx)) & \\ n \triangleq \lambda f.\, \lambda x.\, f(\ldots(f(fx))) & (\text{with } n \text{ applications of } f \text{ in a row}) \end{array}$$

And so on. We can also now define the successor function, *SUCC*. This implementation uses the argument number n to construct the function f^n, then applies f one more time to get f^{n+1}.

$$SUCC = \lambda n.\, \lambda f.\, \lambda x.\, f((n\ f)\ x)$$

Addition can be defined using $SUCC$; adding n_1 to n_2 is simply taking the successor of n_2, n_1 times in a row.

$$PLUS = \lambda n_1 n_2.\, (n_1\ SUCC)\ n_2$$

Functions such multiplication, exponentiation, subtraction can also be implemented.

We can implement additional useful features such as data structures using lambda calculus terms, as discussed in Pierce.