CS611 Lecture 7 Lambda Calculus 2001 September 14

Scribe: Eric Breck and Jeff Hartline Lecturer: Andrew Myers

1 Introduction

We have developed a number of tools for describing programming languages, but so far we have only applied the toolbox to IMP. IMP has no functions, (procedures, routines, methods). It has no way to write a functional abstraction. This makes it pretty uninteresting as a programming language. So we will overcompensate and introduce a language with nothing but functional abstraction: the lambda calculus.

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. However, not all functions are 'realizable'—there are functions that you can describe but which aren't computable. Lambda calculus is one attempt to write down functions that you could actually hope to evaluate in the real world.

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:

$$\begin{array}{lcll} e & ::= & x & \text{(identifiers)} \\ & \mid & e_0\ e_1 & \text{(application)} \\ & \mid & \lambda\ x\ e & \text{(abstraction)} \end{array}$$

In an abstraction, x is the *argument*, e is the *body* of the function. Notice that this function doesn't have a name. Even in mathematics, you would say $f(x) =$... and then the function would have a name f. In lambda calculus, functions are anonymous.

Here are some examples of terms (for clarity we will fully parenthesize our expressions). First, the identity function:

$$(\lambda\ x\ x)$$

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

$$(\lambda\ x\ (\lambda\ a\ a))$$

Note: In some of the examples that follow, there will appear symbols that do not appear in the above grammar (+,1,2,INC,...). These can be thought of as identifiers that are defined somewhere else or as shorthand for some term that we have not defined yet (see next class).

2.1 Closed terms

Every term has some identifiers. An identifier is *bound* to the closest enclosing variable of the same name. For example, in $(\lambda\ x\ (\lambda\ x\ x))$ - the innermost x is talking about the inner x variable, not the outer x variable. A *closed term* is one in which all identifiers are bound. An identifier that is not bound is called a *free variable* of the term. An *open term* is not closed. We will consider a *program* in the lambda calculus to be any closed term.

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:

$$\begin{array}{lcll} e & ::= ... & \lambda\ (x_1 ... x_n)\ e & \text{(multi-abstraction)} \\ & | & e_0\ e_1\ e_2\ ...\ e_n & \text{(multi-application)} \end{array}$$

In the multi-application, e_0 is a n-arg function, $e_1 ... e_n$ are arguments.

It turns out this isn't any more expressive than basic lambda calculus. 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?

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

For example, suppose we were going to add one and two. We write $1 + 2$ in ordinary infix notation, or in *lambda*-calculus notation:

$$(+\ 1\ 2)$$

We can desugar this to:

$$((+1)2)$$

So, '(+ 1)' takes a number and adds one to it. This is equivalent to the INC function used below. + is now a higher-order function. This way of taking multi-argument functions and turning them into higher-order functions is called *currying* — in honor of Haskell Curry.

Another useful construct that turns out to be syntactic sugar is local variables. We can desugar

$$\text{let } x = e_1 \text{ in } e_2$$

to

$$((\lambda\ x\ e_2)\ e_1)$$

3 Semantics

3.1 β-reduction

We would like to believe that the expression $(\lambda\ x\ e_0)\ e_1$ is 'equal' in some sense to the body of e_0 where we have replaced x with e_1—we want to 'call' or 'invoke' the function e_0 with e_1 as its argument. The substitution is not as simple as it seems: not all xs should be replaced with e_1, just the xs bound to the outermost x in our expression. The notation for this *syntactic substitution* is as follows: $e_0\{e_1/x\}$. Using this notation, we introduce the β-reduction:

$$(\lambda\ x\ e_0) \mapsto e_0\{e_1/x\}$$

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 have already seen one set of rewrite rules: the small-step semantics of IMP.

Lets see how we might perform computation using β-reduction. In this example, INC is a function which takes a number and returns the next larger number. Start with the following expression:

$$(((\lambda\ x\ (\lambda\ y\ (y\ x\ )))\ 3)\ \text{INC})$$

After we apply one β-reduction, we get:

$$((\lambda\ y\ (y\ 3))\text{INC})$$

Applying another β-reduction yields:

$$(\text{INC}\ 3)$$

Finally, we compute the result of INC applied to 3:

$$4$$

3.2 Infinite loops

We would like to claim that λ-calculus is at least as good as IMP, and therefore we should be able to write an λ-calculus infinite loop. Define the term Ω as follows:

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

If we try β-reducing Ω, we simply get Ω back again. Mathematically: $\Omega \mapsto \Omega$. Using β-reductions, the program Ω will never terminate. When an expression does not terminate, we say the expression *diverges.* Notationally, we write $\Omega \Uparrow$ to mean Ω diverges.

4 λ-Calculus Semantic Systems

λ-calculus has two common semantic systems: call-by-name semantics and call-by-value semantics. Call-by-name semantics adheres to lazy evaluation, where expressions are passed around unevaluated for as long as possible. Call-by-value semantics performs strict evaluation: all expressions are evaluated before being passed as function arguments. Call-by-name thus says do not evaluate function arguments, let the function decide when to perform evaluation. Call-by-value requires that function arguments be reduced to values before the function is processed.

What is a value? A value is defined as a final configuration of the operational semantics: a legal state that can no longer be operated on or 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 CBN and CBV systems of structural operational semantics (SOS):

	β *reduction*	evaluation steps	
Call-by-name SOS:	$\dfrac{}{(\lambda\ x\ e_0)e_1 \mapsto e_0\{e_1/x\}}$	$\dfrac{e_0 \ \mapsto\ e_0'}{e_0\ e_1\ \mapsto\ e_0'\ e_1}$	
Call-by-value SOS:	$\dfrac{}{(\lambda\ x\ e)v \mapsto e\{v/x\}}$	$\dfrac{e_0 \ \mapsto\ e_0'}{e_0\ e_1\ \mapsto\ e_0'\ e_1}$	$\dfrac{e\ \mapsto\ e'}{v\ e\ \mapsto\ v\ e'}$

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

Is there a difference between CBN and CBV? YES! Recalling our expression for Ω, let us try:

$$(\lambda\ x\ (\lambda\ y\ y))\ \Omega$$

Let us see what happens when we apply CBN and CBV semantics to this statement.

$$(\lambda\ x\ (\lambda\ y\ y))\ \Omega \quad \Downarrow^{CBN} \quad (\lambda\ y\ y)$$

$$(\lambda\ x\ (\lambda\ y\ y))\ \Omega \quad \Uparrow^{CBV}$$

In call-by-name, we ignore the fact that Ω diverges and return the identity function. In call-by-value, we are forced to evaluate Ω and thus the program diverges. However, if a program does not diverge using call-by-value semantics then call-by-name and call-by-value semantics will give the same result. Although it appears that call-by-name is a superior semantics, most programming languages (ML, for instance) are call-by-value because laziness is difficult to implement efficiently.