Scribe: Elliot Anshelevich | Lecturer: Andrew Myers

1 Definition of Lambda Calculus

So far we have only looked at IMP, which has no functions. Now, we will look at another language, known as Lambda Calculus, which is all functions. Here is a context-free grammar of this language:

$$e \ ::= x \mid e_0 \ e_1 \mid \lambda \ x \ e_0$$

where

x is an **Identifier**. This refers to a variable defined by surrounding context.

$e_0 \ e_1$ is an **Application**. Here, e_0 is a function and e_1 is the argument given to it, so $e_0 \ e_1$ applies the function e_0 to the argument e_1.

$\lambda \ x \ e_0$ is an **Abstraction/lambda term**. This defines a new function with argument variable x and body e_0 (something like ML's fn $x \Rightarrow e_0$)

The Lambda Calculus is actually a notation for writing down mathematical functions, but we can also treat it as a universal, simple core language. Note that while Lisp and Scheme are based somewhat on Lambda Calculus, there are differences as well.

So, what is a valid program in Lambda Calculus? To answer that, we must first define open and closed terms. A *term* is an expression denoting a value. For example, something like int in C or Java would not be a term. A *closed* term is a term where all identifiers are bound by the closest containing abstraction. For example, in the term ($\lambda \ x$ ... x ($\lambda \ y$ ... y ...) ...), the y is bound to the inner lambda term and x to the outer. An *open* term is a term that is not closed, i.e. where there are some identifiers that are not bound to anything. For example, the term ($\lambda \ x \ (y \ x)$) is open, since y is not bound to anything in this term. Now we can finally define the set of legal Lambda Calculus programs. This is just the set of *all closed terms*.

To fully define Lambda Calculus, we must still define the evaluation rules for it. In Lambda Calculus, we consider functions to be values, which means that a lambda term evaluates to itself, since it is already a final value. Therefore, we only need to define evaluation rules for applications. Applications are evaluated by a rule known as β-reduction:

$$((\lambda \ x \ e_1) \ e_2) \rightarrow e_1\{e_2/x\}$$

where $e_1\{e_2/x\}$ means "e_1 with e_2 substituted for occurrences of x". Note that defining "substituted" can be rather tricky. Here are some examples of β-reduction:

$$((\lambda \ \mathsf{x} \ \mathsf{x}) \ e) \rightarrow \mathsf{x}\{e/\mathsf{x}\}=e$$
$$((\lambda \ \mathsf{x} \ (\lambda \ \mathsf{x} \ \mathsf{x})) \ e) \rightarrow (\lambda \ \mathsf{x} \ \mathsf{x})\{e/\mathsf{x}\} = (\lambda \ \mathsf{x} \ \mathsf{x})$$
$$(((\lambda \ \mathsf{x} \ (\lambda \ \mathsf{y} \ (\mathsf{y} \ \mathsf{x}))) \ 3) \ \mathit{INC}) \rightarrow ((\lambda \ \mathsf{y} \ (\mathsf{y} \ 3)) \ \mathit{INC}) \rightarrow (\mathit{INC} \ 3) \rightarrow 4$$

In the above examples, we have used INC (the functions which takes an integer as an argument and adds 1 to it) and 3. However, we have no numbers appearing in our grammar for Lambda Calculus. In fact, we shall see how to form numbers from closed lambda terms later.

2 Functions With Multiple Arguments

⟩From the constructs we defined in the previous section, we can form many others. We shall start with higher-order functions. With Lambda Calculus, we can express functions which return or accept other functions easily (since all values are *only* functions). For example, here is a function which applies another function to 5 and returns the result: ($\lambda \ f \ (f \ 5)$). And here is a function that returns a function that applies another function to its argument: ($\lambda \ v \ (\lambda \ f \ (f \ v))$). Applied to 5, it gives us the previous function.

What about functions which take multiple arguments? The grammar we defined for Lambda Calculus only allows for functions which take one argument. To allow for multiple arguments, we would apparently need something like:

$$e \ ::= \ ... \mid e_0 \ e_1 ... e_n \mid \lambda \ (x_1 ... x_n) \ e_0$$

Here, $\lambda \ (x_1 ... x_n) \ e_0$ is a function which takes n arguments $x_1 ... x_n$, and has the body e_0. $e_0 \ e_1 ... e_n$ denotes an application of a function e_0 which takes $n-1$ arguments to the arguments $e_1 ... e_n$. However, we do not need these additions and can treat multiple-argument application and abstraction as convenient *syntactic sugar*. We can *desugar* (trivially rewrite syntactically) these terms into the single-argument calculus:

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

In this way, multi-argument functions are *curried* (applied one argument at a time):

$$(+ \ 1 \ 5) \ \Rightarrow \ ((+ \ 1) \ 5)$$

Notice that $(+ \ 1 \ 5)$ is really just a shorthand (syntactic sugar) for $((+ \ 1) \ 5)$. Here is another example:

$$(((\lambda \ x \ (\lambda \ y \ (y \ x \ ))) \ 3) \ INC) \rightarrow ((\lambda \ y \ (y \ 3)) \ INC) \rightarrow (INC \ 3) \rightarrow 4$$
Shorthand: $(\lambda \ (x \ y) \ (\ y \ x \ )) = (\lambda \ x \ (\lambda \ y \ (\ y \ x \ )))$
$$((\lambda \ (x \ y) \ (\ y \ x \ )) \ 3 \ INC) \rightarrow (INC \ 3) \rightarrow 4$$

3 Operational Semantics

Now we shall consider an operational semantics for Lambda Calculus. The configuration is just an expression of the language, since we have no store (the state is determined entirely by the expression). In large-step semantics, we have the following inference rule:

$$\frac{e_0 \Downarrow \lambda \ x \ e_2 \quad e_2\{e_1/x\} \Downarrow v}{e_0 \ e_1 \Downarrow v}$$

This rule can actually be applied in several different ways. Using *call-by-name* semantics, we have that e_1 is not evaluated before substitution. However, we could also use *call-by-value* semantics, in which case arguments are fully evaluated before substituting them into the body of the function. In either case, we have that any lambda term is a *value*: $v ::= \lambda \ x \ e$. Therefore, if we wanted to write an evaluation rule for a lambda term, it would just evaluate to itself, since it is a value. The only other possible case other than a lambda term or an application is a single identifier. However, that is not a valid program since it is not closed.

As for small-step semantics, we have the following rules for call-by-name:

$$\frac{}{(\lambda \ x \ e_1) \ e_2 \ \rightarrow \ e_1\{e_2/x\}} \qquad (\beta\text{-reduction})$$

$$\frac{e_1 \rightarrow e_1'}{e_1 \ e_2 \ \rightarrow \ e_1' \ e_2}$$

To model call-by-value semantics, we instead have the following rules:

$$\frac{}{(\lambda \ x \ e_1) \ v \ \rightarrow \ e_1\{v/x\}} \qquad (\beta\text{-reduction})$$

$$\frac{e_2 \rightarrow e_2'}{v \ e_2 \ \rightarrow \ v \ e_2'}$$

$$\frac{e_1 \rightarrow e_1'}{e_1 \ e_2 \ \rightarrow \ e_1' \ e_2}$$

These rules require that an expression must be fully evaluated before it can be substituted in a β reduction.

4 Some More Constructs

Since Lambda Calculus is Turing complete, there must be a way to write an infinite loop in it. Here is an infinite loop:

$$LOOP \triangleq (\lambda\ x\ (x\ x))(\lambda\ x\ (x\ x)) \rightarrow\ ?$$

This expression *diverges* (never stops taking small steps), since, as the reader can easily check, this expression evaluates to itself: $LOOP \rightarrow LOOP$.

When looking at a language like this, you might start missing some constructs that you are used to. However, some of them are not really needed, since they can be simulated using our current constructs. For example, Lambda Calculus has no "let" statement like ML does. However, we can simulate (desugar) a let statement in the following way:

$$(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \Longrightarrow (\lambda\ x\ e_2)\ e_1$$

Lambda calculus terms can become long. For compactness we will use certain names, as well as multiple arguments, as shorthand. These are not actually part of the language. Here are some definitions for names we will use:

$$IDENTITY \triangleq (\lambda\ x\ x)$$
$$INC \triangleq (+\ 1)$$
$$APPLY\text{-}TO\text{-}FIVE \triangleq (\lambda\ f(f\ 5))$$
$$COMPOSE \triangleq (\lambda\ (f\ g)(\lambda\ x\ (f\ (g\ x))))$$
$$TWICE \triangleq (\lambda\ f\ (COMPOSE\ f\ f))$$

Here, *COMPOSE* composes two functions, and *TWICE* returns a function that calls the given function twice. For example:

$$((COMPOSE\ INC\ INC)\ 2) \rightarrow 4$$
$$((TWICE\ (TWICE\ INC))\ 0) \rightarrow 3$$

Lambda Calculus is universal. This means that no primitive boolean type or "if" statement is needed. We can form them as follows:

$$TRUE \triangleq (\lambda\ x\ (\lambda\ y\ x)) \ \sim\ (\lambda\ (x\ y)\ x)$$
$$FALSE \triangleq (\lambda\ x\ (\lambda\ y\ y)) \ \sim\ (\lambda\ (x\ y)\ y)$$
$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \ \Rightarrow (IF\ e_1\ e_2\ e_3)$$
$$IF \triangleq (\lambda\ (x\ y\ z)(x\ y\ z))$$

So, *TRUE* is a function which takes two arguments and returns the first one, and *FALSE* returns the second one. Here is why *IF* works:

$$(IF\ TRUE\ e_2\ e_3) \rightarrow (((\lambda\ x\ (\lambda\ y\ x))\ e_2)\ e_3) \rightarrow ((\lambda\ y\ e_2)\ e_3) \rightarrow e_2$$

IF works similarly if the first argument to it evaluates to *FALSE*. Note that call-by-name here is important! e_2 and e_3 are not evaluated eagerly by *IF*.

We can also represent pairs and lists. The pair/list operations are:

$(CONS\ x\ y)$: construct a list with head x and tail y

$(FIRST\ p)$: return first item in list (or first item in pair)

$(REST\ p)$: return remainder of list (or second item in pair)

Here is one way to implement these operations:

$$CONS \triangleq (\lambda\ (x\ y)(\lambda\ f\ (f\ x\ y)))$$
$$FIRST \triangleq (\lambda\ p\ (p\ (\lambda\ (x\ y)\ x))) \quad = (\lambda\ p\ (p\ TRUE))$$
$$REST \triangleq (\lambda\ p\ (p\ (\lambda\ (x\ y)\ y))) \quad = (\lambda\ p\ (p\ FALSE))$$

Another structure which we definitely need is the natural numbers. We can model the number n as a function that composes an arbitrary function n times. These numbers are called *Church numerals*. Here is what they look like:

$0 \triangleq (\lambda\ (f\ a)\ a) \qquad (=FALSE)$

$1 \triangleq (\lambda\ (f\ a)\ (f\ a))$

$2 \triangleq (\lambda\ (f\ a)\ (f\ (f\ a)))$

$3 \triangleq (\lambda\ (f\ a)\ (f\ (f\ (f\ a))))$

$n \triangleq (\lambda\ (f\ a)\ (f\ (...(f\ a)...)))$

We can now define the *INC* function, that adds one to a number, by writing a function that interposes an extra call to the function as follows:

$$\begin{gathered} n \triangleq (\lambda\ (f\ a)(f^n\ a)) \quad \text{, so} \\ nf = (\lambda\ a\ (f^n\ a)) \quad \text{, and} \\ f((n\ f)\ a) = f^{n+1}\ a \quad \text{, therefore} \\ INC \triangleq (\lambda\ n\ (\lambda\ (f\ a)(f((n\ f)\ a)))) \end{gathered}$$

We can now define + and other arithmetic operators, by using the same trick:

$$\begin{gathered} + \triangleq (\lambda\ (n_1\ n_2)\ (\lambda\ (f\ a)((n_1\ f)((n_2\ f)\ a)))) \qquad \text{or} \\ + \triangleq (\lambda\ (n_1\ n_2)\ ((n_1\ INC)\ n_2)) \end{gathered}$$