1 Lambda Calculus

Lambda calculus is a notation for describing mathematical functions and even programs. We will first assume that there is a countably infinite set of variable names we can use. An element in the language is:

1. A variable x
2. A function e_0 applied to an argument e_1, written $e_0(e_1)$ or just $e_0\ e_1$.
3. Or a body of code with every instance of a variable replaced by a formal argument. $\lambda x.e$

About parentheses: Parentheses are used just for grouping; they have no meaning on their own. Expressions are parsed left to right, and lambdas are greedy, extending as far to the right as they can. For simplicity, multiple variables may be placed after the lambda. This is really just a shorthand for having a lambda in between each variable. For example, we write $\lambda x, y.\, e$ as shorthand for $\lambda x.\, \lambda y.\, e$. This shorthand is an example of *syntactic sugar*, and the process of removing it is called *currying*.

What a mathematician might express as:

$$x \mapsto x^2$$

Lambda Calculus expresses as:

$$\lambda x.x^2$$

This suggests that functions are just ordinary variables, and can be passed as values to a different function or even themselves.

Lambda calculus has two separate uses here. First, it's a mathematical way of expressing programs. It allows us to mathematically manipulate them as data. Second, it's a fully functional programming language. It will be used as such.

The syntax of the lambda calculus is defined by a context-free grammar:

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

There are some minor differences between how we write this grammar from the way that grammars are written in a compilers course. We refer to the e as a *syntactic metavariable*. That is, it is a variable representing language syntax; it is not a variable at the level of the programming language. We use subscripts to differentiate syntactic metavariables. For example, e_0 and e_1 are both metavariables of the same syntactic class as e.

1.1 Variable Binding

All variables must be bound. A lambda expression introduces a *binding occurrence* before the dot. Each variable must be bound to a binding occurrence. We define the scope of a variable by the closest binding occurrence. This is called *lexical scoping*; the variable's scope is defined by the text of the program. It is "lexical", because it is possible to determine its scope before the program runs, by reading the program text in a straightforward way.

There are different kinds of expressions in a typical programming language: *terms* and *types*. A term represents a value or computation that exists at run time; a type is largely a compile-time expression used by the compiler to rule out ill-formed programs. For now there are no types; all expressions are terms. A closed term is a term in which all terms are bound; An open term is the negation of a closed term. A *well-formed program* in the lambda calculus is any closed term.

1.2 β reduction

Now we get to the question of how do we run lambda calculus programs. For example, $(\lambda x.\, e_1)\ e_2$ should be equivalent to e_1 with e_2 replacing all the x's in e_1. We can express this as $e_1\{e_2/x\}$, although there are many notations for substitution. Pierce writes $[x \mapsto e_2]e_1$. Because we will be using similar notation for something else, we will use the notation $e_1\{e_2/x\}$.

For lambda calculus terms to make sense as functions, we expect that the following two terms are equivalent and should mean the same thing: $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$. This equivalence is known as a β-equivalence. Rewriting $(\lambda x.\, e_1)\ e_2$ into $e_1\{e_2/x\}$ is called a β-reduction. If we start with a lambda calculus expression, in general we may be able to perform beta reductions. This corresponds to executing the lambda calculus expression as a program.

1.3 CBV vs. CBN

Now we have another question. What order do we perform these beta reductions in? Most languages use what's known as Call By Value semantics (CBV). They only call functions on values. In other words, $(\lambda x.e_1)e_2$ only reduces if e_2 is a value. But what is a value? We define them as ordinary lambda expressions which are not applied. In other words $\lambda x.x$ is a value, while $(\lambda x.x)1$ is not. Here is an example of CBV evaluation through beta reductions, assuming 3, 4, and $SUCC$ are appropriately defined.

$$\begin{aligned}(((\lambda x.(\lambda y.(y\ x)))\ 3)\ SUCC) &\longrightarrow ((\lambda x.(\lambda y.(y\ x)))\ 3)\\ &\longrightarrow \lambda y.(y\ 3)\\ &\longrightarrow (\lambda y.(y\ 3))\ SUCC\\ &\longrightarrow SUCC\ 3\\ &\longrightarrow 4\end{aligned}$$

Another approach is to use Call by Name semantics (CBN). We defer evaluation of arguments until as late as possible, applying reductions from left to right within the expression.

1.4 Formal SOS

Let's try to formalize CBV with a few inference rules.

$$\frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}\ [\beta \text{ reduction}]$$

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

$$\frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}$$

This is an operational semantics for a programming language based on the lambda calculus. An operational semantics is a language semantics that describes how to run the program. This can be done through informal human-language text, as in the Java Language Specification, or through more formal rule. Rules of this form are known as a Structural Operational Semantics (SOS). It defines evaluation in terms of single steps that can be performed during evaluation, and these single steps are defined in term of the structure of the expression being evaluated. This kind of semantics is known as a *small-step* semantics because it only describes one step at a time; an alternative is a *big-step* (or *large-step*) semantics that describes the entire evaluation of the program to a final value.

We will see other kinds of semantics later in the course, such as axiomatic semantics which tells you what you can prove about a program. Also, there is denotational semantics, which translates a program into an underlying mathmatical representation.

Expressed as SOS, CBN has slightly simpler rules:

$$\frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}\ [\beta \text{ reduction}]$$

$$\frac{e_0 \longrightarrow e_0'}{e_0\ e_1 \longrightarrow e_0'\ e_1}$$

We don't need the rule for evaluating the right-hand side of an application because beta reductions are done immediately once the left-hand side is a value.

1.5 Ω

Let us define an expression we will call Ω:

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

So... what happens when we try to evaluate Ω?

$$\Omega \longrightarrow \Omega$$

We've just coded an infinite loop!

Now what happens if we try using Ω as a parameter? Consider:

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

If we use CBV evaluation on the above program, then we have to first reduce Ω. This puts the evaluator into an infinite loop. CBN on the other hand reduces the above to $\lambda y.\, y$. CBN has an important property: CBN will not go into an infinite loop unless every other semantics would also go into an infinite loop, yet it agrees with CBV whenever CBV would terminate successfully.