1 The Lambda Calculus

Lambda calculus is a notation for describing mathematical functions and programs. It is a mathematical system for studying the interaction of *functional abstraction* and *functional application*. It captures some of the essential, common features of a wide variety of programming languages. Because it directly supports abstraction, it is a much more natural model of universal computation than a Turing machine is.

1.1 Syntax

A λ calculus term is:

1. a variable $x \in \mathbf{Var}$, where **Var** is a countably infinite set of variables;
2. a function e_0 applied to an argument e_1, usually written $e_0\ e_1$ or $e_0(e_1)$; or
3. a lambda term, an expression $\lambda x.\, e$ representing a function with input parameter x and body e. Where a mathematician might write $x \mapsto x^2$, in the λ-calculus we would write $\lambda x.x^2$.

In BNF notation,

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

Note that we used the word *term* instead of *expression*. A term is an expression that describes a computation to be performed. In general, programs may contain expressions that are not terms; for example, type expressions. However, in the untyped lambda calculus that we are now studying, all expressions are terms. A term represents a value that exists only at run time; a type is a compile-time expression used by the compiler to rule out ill-formed programs. For now there are no types.

Parentheses are used just for grouping; they have no meaning on their own. Like other familiar binding constructs from mathematics (e.g., sums, integrals), lambda terms are greedy, extending as far to the right as they can. Therefore, the term $\lambda x.\, x\ \lambda y.\, y$ is the same as $\lambda x.\, (x\ (\lambda y.\, y))$, not $(\lambda x.\, x)\ (\lambda y.\, y)$.

For simplicity, multiple variables may be placed after the lambda, and this is considered shorthand for having a lambda in front of each variable. For example, we write $\lambda xy.\, e$ as shorthand for $\lambda x.\, \lambda y.\, e$. This shorthand is an example of *syntactic sugar*. The process of removing it in this instance is called *currying*.

We can apply a curried function like $\lambda x.\, \lambda y.\, x$ one argument at a time. Applying it to one argument results in a function that takes in a value for x and returns a constant function, one that returns the value of x no matter what argument it is applied to. As this suggests, functions are just ordinary values, and can be the results of functions or passed as arguments to functions (even to themselves!). Thus, in the lambda calculus, functions are *first-class values*. Lambda terms serve both as functions and data.

1.2 BNF Notation

In the grammar

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

describing the syntax of the pure λ-calculus, the e is not a variable in the language, but a *metavariable* representing a syntactic class (in this case λ-terms) in the language. It is not a variable at the level of the programming language. We use subscripts to differentiate syntactic metavariables of the same syntactic class. For example, e_0, e_1 and e all represent λ-terms.

1.3 Variable Binding

Occurrences of variables in a λ-term can be *bound* or *free*. In the λ-term $\lambda x.\, e$, the lambda abstraction operator λx binds all the free occurrences of x in e. The *scope* of λx in $\lambda x.\, e$ is e. 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 inspecting the program text. A term is *closed* if all variables are bound. A term is *open* if it is not closed.

2 Substitution and β-reduction

Now we get to the question: How do we run a λ-calculus program? The main computational rule is called *β-reduction*. This rule applies whenever there is a subterm of the form $(\lambda x.\, e)\ e'$ representing the application of a function $\lambda x.\, e$ to an argument e'.

To perform a β-reduction, we substitute the argument e' for all free occurrences of the formal parameter x in the body e. This corresponds to our intuition for what the function $\lambda x.\, e$ means.

We have to be a little careful; we cannot just substitute e' blindly for x in e, because bad things could happen which could alter the meaning of expressions in undesirable ways. We only want to replace the free occurrences of x within e, because any other occurrences are bound to a different binding; they are really different variables. There are some additional subtleties to substitution that we'll return to later.

There are many notations for substitution, which can be confusing. Pierce writes $[x \mapsto e']e$. Other notations for the same idea are encountered frequently, including $e[x \mapsto e']$, $e[x \leftarrow e']$, $e[x := e']$. Because we will use square brackets for other purposes, we will use the notation $e\{e'/x\}$.

Rewriting $(\lambda x.\, e)\ e'$ to $e\{e'/x\}$ is the basic computational step of the λ-calculus. In this rewrite step, the reduced expression (or *redex*, for short) is $(\lambda x.\, e)\ e'$, and the right-hand side, or *contractum*, is $e\{e'/x\}$.

2.1 α-renaming

In the term $\lambda x.x\ z$, the name of the bound variable x does not really matter. This term is semantically the same as $\lambda y.\, y\ z$. Renamings like this are known as α-reductions. In an α-reduction, the new bound variable must be chosen so as to avoid capture. If a term α-reduces to another term, then the two terms are said to be α-equivalent. This defines an equivalence relation on the set of terms, denoted $e_1 =_\alpha e_2$. An α-reduction doesn't really make computational progress, so it is often referred to as α-renaming. Recall the definition of free variables $FV(e)$ of a term e. In general, we have $\lambda x.\, e =_\alpha \lambda y.\, e\{y/x\}$ when $y \notin FV(e)$. The proviso $y \notin FV(e)$ is to avoid the capture of a free occurrences of y in e as a result of the renaming.

2.2 Ω

Let us define an expression we will call Ω:

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

What happens when we try to evaluate it? Applying a β-reduction, we get the same term back again:

$$\Omega \quad = \quad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \quad \longrightarrow \quad (x\ x)\{(\lambda x.\, x\ x)/x\} \quad = \quad \Omega$$

We have just coded an infinite loop! When an expression e can go through infinite sequence of evaluation steps, we write $e \Uparrow$. When it evaluates to a value v, we write $e \Downarrow v$ or just $e \Downarrow$ if we don't care what the value is.

2.3 η-reduction

Here is another notion of equality. Compare the terms e and $\lambda x.\, e\ x$ If these two terms are both applied to an argument e', they will both reduce to $e\ e'$, provided x has no free occurrence in e. Formally, $(\lambda x.\, e_1\ x)\ e_2 \longrightarrow e_1\ e_2$ if $x \notin FV(e_1)$. Therefore, e and $\lambda x.\, e\ x$ behave the same way when treated as functions and should

be considered equal. Another way of stating this is that e and $\lambda x.\, e\ x$ behave the same way in all contexts of the form $[\,\cdot\,]\ e'$. This gives rise to a reduction rule called η-reduction: $\lambda x.\, e\ x \longrightarrow e$ if $x \notin FV(e)$. The reverse operation, called η-expansion, has practical uses as well. In practice, η-expansion is used to delay divergence by trapping expressions inside λ-terms.

2.4 Confluence

In the classical λ-calculus, no reduction strategy is specified, and no restrictions are placed on the order of reductions. Any redex may be chosen to be reduced next. A λ-term in general may have many redexes, so the process is nondeterministic. We can think of a reduction strategy as a mechanism for resolving the nondeterminism, but in the classical λ-calculus, no such strategy is specified. A value in this case is just a term containing no redexes. Such a term is said to be in normal form.

This makes it more difficult to define extensional equality. One sequence of reductions may terminate, but another may not. It is even conceivable that different terminating reduction sequences result in different values. Luckily, it turns out that the latter cannot happen.

It turns out that the λ-calculus is confluent (also known as the Church–Rosser property) under α- and β-reductions. Confluence says that if e reduces by some sequence of reductions to e_1, and if e also reduces by some other sequence of reductions to e_2, then there exists an e_3 such that both e_1 and e_2 reduce to e_3. It follows that normal forms are unique up to α-equivalence. For if $e \Downarrow v_1$ and $e \Downarrow v_2$, and if v_1 and v_2 are in normal form, then by confluence they must be α-equivalent. Moreover, regardless of the order of previous reductions, it is always possible to get to the unique normal form if it exists.

However, note that it is still possible for a reduction sequence not to terminate, even if the term has a normal form. For example, $(\lambda x.\lambda y.y)\ \Omega$ has a nonterminating reduction sequence $(\lambda xy.y)\ \Omega \longrightarrow (\lambda xy.y)\ \Omega \longrightarrow \ldots$ but also has a terminating reduction sequence, namely $(\lambda x.\lambda y.y)\ \Omega \longrightarrow \lambda y.y$. It may be difficult to determine the most efficient way to expedite termination. But even if we get stuck in a loop, the confluence property guarantees that it is always possible to get unstuck, provided the normal form exists.

3 Encoding language features

Even though all values in the λ-calculus are functions, it would be nice to somehow have objects which could be worked with like integers and boolean values, and that let us build data structures.

3.1 Encoding booleans

We wish to implement functions *TRUE*, *FALSE*, *IF*, *AND*, and so forth, such that the expected behavior holds, including statements such as

$$\begin{aligned} \mathit{IF\ TRUE}\ x\ y &\rightarrow x \\ \mathit{AND\ TRUE\ FALSE} &\rightarrow \mathit{FALSE} \end{aligned}$$

If, for no *a priori* good reason, we define *TRUE* and *FALSE* as:

$$\begin{aligned} \mathit{TRUE} &\triangleq \lambda xy.\, x \\ \mathit{FALSE} &\triangleq \lambda xy.\, y \end{aligned}$$

Then we see we desire to have *IF* be of the form

$$\mathit{IF} = \lambda b\ t\ f.\, (\mathit{if}\ b = \mathit{TRUE}\ \mathit{then}\ t,\ \mathit{if}\ b = \mathit{FALSE}\ \mathit{then}\ f)$$

And now the definitions used for the boolean values become useful, because $\mathit{TRUE}\ t\ f \rightarrow t$ and $\mathit{FALSE}\ t\ f \rightarrow f$, so all we need to do is apply the boolean passed to *IF*:

$$\mathit{IF} \triangleq \lambda b\ t\ f.\, (b\ t\ f)$$

With *IF* in hand, defining other boolean operators becomes straightforward (if rather inefficient):

$$
\begin{array}{rcl}
AND & \triangleq & \lambda b_1\ b_2.\, IF\ (b_1)\ (IF\ b_2\ TRUE\ FALSE)\ (FALSE) \\
OR & \triangleq & \lambda b_1\ b_2.\, IF\ (b_1)\ (TRUE)\ (IF\ b_2\ TRUE\ FALSE) \\
NOT & \triangleq & \lambda b_1.\, IF\ b_1\ FALSE\ TRUE
\end{array}
$$

We have no types here, so while the behavior of these operators is clear when they are fed boolean values as we have defined them, they can be applied to any λ-term... though with a good chance of garbage coming out.

3.2 Encoding integers

To encode numbers, we'll use Church numerals. That is, the number n represented as a function which, given another function, returns the n-fold composition of that other function: $n(f) \mapsto f^n$. So, for example,

$$
\begin{array}{rcll}
0 & \triangleq & \lambda f\ x.\, x & \textit{(since } f^0(x) = x\textit{)} \\
1 & \triangleq & \lambda f\ x.\, fx & (f^1 = f) \\
2 & \triangleq & \lambda f\ x.\, f(fx) & (f^2(x) = f(f(x))) \\
SUCC & \triangleq & \lambda n.\, \lambda f x.\, f((nf)x) & \textit{Applying } f \textit{ once more}
\end{array}
$$

With these numbers, we can perform simple arithmetic, such as *PLUS*. An obvious approach might be

$$PLUS \triangleq \lambda n_1\ n_2.\, \lambda f\ x.\, (n_2 f)((n_1 f)x)$$

Here we are applying f^{n_2} to f^{n_1} to get $f^{n_1+n_2}$. Alternately, recall that numbers (as we have defined them) act on functions to repeatedly apply the function, and addition can be viewed as repeated application of the successor function:

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

We can also define a function that computes the predecessor of a number, and therefore subtraction. And since multiplication is simply repeated addition, we can build a multiplication operator using *PLUS*.