Lecture 13: λ Calculus I

built on 2021/02/16 at 09:42:20

we will look at a simple computation model—arguably much simpler than the Turing machine we saw earlier—that turns out to be equivalent to the Turing machine in terms of expressive power and is believed to be the foundation of modern functional languages (e.g., Haskell, Scala, Standard ML, and Swift)

1 What is Lambda Calculus?

The lambda calculus was originally invented by Alonzo Church in the early 1930's as a basis for studying a kind of logic, known as constructive logic, and not as a model of computation. It was later shown that lambda computability is equivalent to Turing computable functions. For context, it helps to discern the differences between imperative programming and functional programming.

1.1 Imperative vs. Functional

At a high level, the Turing machine (TM) is pretty much similar to our idea of a typical programming language, where the program keeps a state and our computation is accomplished by a series of statements that repeatedly modify such a state to produce an answer. This has the essence of *imperative programming*, the programming paradigm most of us are familiar with.

By contrast, there is a completely different programming paradigm that avoids keeping and changing states, but instead rests on the idea of viewing a computation as the evaluation of mathematical functions. Indeed, the whole computation is simply evaluating functions and data are embedded as part of these functions. This has the essence of *functional programming*. At this point, it should be noted that imperative programs do have functions, but for the most part, they are not functions in the mathematical sense that we are familiar with because such functions keep and modify their states.

1.2 Abstraction and Application

To describe what lambda calculus is, we will first consider the function $f(x) = x + 3$. When we see this, we know that the input is x and the output is given by adding 3 to x. Moreover, we have given this function a name: the function f. Indeed, if we were to read this aloud, we would probably say "let f be a function that adds 3 to x." Upon further examination, it is clear that there are 2 steps happening here: first, we express a function that adds 3 to x; then, we say, let's call this function f.

Lambda Abstraction. The lambda notation allows us to talk about what a function does without giving it an explicit name. For example, we write $\lambda x.\,(x+3)$ to indicate a function that takes x as input and whose output is given by adding 3 to x. This way of defining a function is known as *lambda abstraction*. The flip side of it is application.

Application. In conventional mathematics, we write $f(2)$ to mean apply the function f to the value 2. In lambda calculus, we'll simply write $(\lambda x.\,(x+3))\,(2)$ to mean the same thing. What happens next is we can try to simplify the resulting expression by "plugging in" the applicand for the abstract variable. Hence,

$$(\lambda x.\,(x+3))\,(2) \to 2+3 \to 5.$$

Unfortunately, up until now, we have done nothing more than introducing new notation for concepts that already exist in conventional settings. The lambda calculus, however, represents a more radical departure from the conventional settings. In this framework, everything denotes a function, functions can be defined using lambda abstraction, and anything can be applied to anything else.

As odd as this may sound, we have, for example: if F is a term in lambda calculus, $F(F)$ is always valid. In the literature, this is known as untyped lambda calculus—because expressions are untyped, there are no restrictions on what can be applied to what.

Having gathered some intuition now, we will start over completely.

2 Grammar for Untyped Lambda Calculus

Variables are usually denoted by $x, y, z, \ldots$. Constants are usually denoted by symbols $a, b, c, \ldots$. Terms are defined inductively as follows:

- each constant is a term;
- each variable is a term;
- if M and N are terms, $(M\,N)$ is a term; and
- if M is a term and x is a variable, then $(\lambda x.\, M)$ is a term.

If we strictly follow this grammar, the terms will quickly be loaded with parentheses and λ's. To keep ourselves sane, some convention is in order:

- Because application takes place from left to right, instead of writing $((M\,N)\,P)\,Q$, we tend to prefer writing $M\,N\,P\,Q$.
- Because lambda abstraction is understood to take the widest scope, instead of writing $\lambda x.\,(M\,N\,P)$, we tend to prefer writing $\lambda x.\, M\,N\,P$, which dropped that pair of parentheses.
- Instead of writing $\lambda x.\,\lambda y.\,\lambda z.\,\ldots$, we tend to prefer writing $\lambda xyz.\,\ldots$.

3 Equivalence and Free vs. Bound

Most people will agree that $f(x) = x + 3$ and $f(z) = z + 3$ encode essentially the same function because in both cases, the function adds 3 to the input regardless of what variable is used to denote the input. The notion of α-equivalence captures the same idea in lambda calculus. To describe this precisely, we need to categorize variables as free vs bound.

A variable is *bound* if it is captured—that is, it is part of the argument of a function. Otherwise, the variable is free. This is best illustrated with an example: in $(\lambda z.\, yz)\,x$, the variables x and y are free but z is bound.

Equivalence. Two terms M and N are α-equivalent if they differ only in the names of the bound variables. That is to say, we can obtain N by suitably renaming the bound variables of M, For example, $\lambda x.\, x$ and $\lambda y.\, y$ are equivalent.

Throughout this chapter, it is convenient to think of α-equivalent terms as being the same term. Therefore, when we say M and N are the same, this should be taken to mean M is equivalent to N, written $M \equiv_\alpha N$.

4 Beta Reductions

The main activity we do with lambda terms is we simplify them. If M and N are lambda terms and x is a variable, then we write $[x \leftarrow N]\,M$ to denote the result of substituting N for x in M after first renaming any bound variables of M that would interfere with the free variables of N after performing the substitution. As a basic example,

$$[x \leftarrow yyz]\,\lambda w.\, xxw = \lambda w.\,(yyz)\,(yyz)\,w$$

This is important because $(\lambda x.\, M)\,N$ means the same as $[x \leftarrow N]\,M$—and the act of replacing the former by the latter is known as β-contraction.

With this in mind, we say that Q *β-reduces to Q' in one step*, written $Q \to_1 Q'$ if Q can be turned into Q' by performing a single β-contraction. We say that Q *β-reduces to Q'*, written $Q \to_* Q'$ if Q can be turned into Q'

through a series of zero or more one-step β-reductions. If a term cannot be β-reduced anymore, we say that it is β-irreducible.

Because β-reduction is the only kind of reduction discussed in this chapter, we will simply say "reduce" instead of "β-reduce".

Example 4.1 *Here are some examples, including some rather odd happenings:*

(i) As a basic example, we have

$$\begin{aligned}(\lambda x.\, xxy)\,\lambda z.\, z &\to_1 (\lambda z.\, z)\,(\lambda z.\, z)\,y \\ &\to_1 (\lambda z.\, z)\,y \\ &\to_1 y\end{aligned}$$

(ii) "Simplifying" a term can make it more complex:

$$\begin{aligned}(\lambda x.\, xxy)\,(\lambda x.\, xxy) &\to_1 (\lambda x.\, xxy)\,(\lambda x.\, xxy)\,y \\ &\to_1 (\lambda x.\, xxy)\,(\lambda x.\, xxy)\,yy \\ &\to_1 (\lambda x.\, xxy)\,(\lambda x.\, xxy)\,yyy \\ &\to_1 \ldots\end{aligned}$$

(iii) "Simplfying" doesn't always lead to an irreducible form:

$$\begin{aligned}(\lambda x.\, xx)\,(\lambda y.\, yyy) &\to_1 (\lambda y.\, yyy)\,(\lambda y.\, yyy) \\ &\to_1 (\lambda y.\, yyy)\,(\lambda y.\, yyy)\,(\lambda y.\, yyy) \\ &\to_1 \ldots\end{aligned}$$

(iv) "Simplifying" a term can leave a term unchanged:

$$(\lambda x.\, xx)\,(\lambda x.\, xx) \to_1 (\lambda x.\, xx)\,(\lambda x.\, xx)$$

(v) Some term can be reduced in more than one way, yet leading to the same final outcome.

I: *Start with the outermost application:*

$$\begin{aligned}(\lambda x.\,(\lambda y.\, yx)\,z)\,v &\to_1 (\lambda y.\, yv)\,z \\ &\to_1 zv\end{aligned}$$

II: *Start with the innermost application:*

$$\begin{aligned}(\lambda x.\,(\lambda y.\, yx)\,z)\,v &\to_1 (\lambda x.\, zx)\,v \\ &\to_1 zv\end{aligned}$$

Rather surprisingly, the final outcome is zv in both cases.

The fact that the final outcome in the last example is the same regardless of the intermediate derivation steps is not a coincidence. This stems from a deep and important property of lambda calculus.

Theorem 4.2 (Church-Rosser Property) *If $M \to_* N_1$ and $M \to_* N_2$, then there is a term P such that $N_1 \to_* P$ and $N_2 \to_* P$.*

Before we move on, let's try our hand at the following quick exercise:

$$\begin{aligned}(\lambda x.\, y)\,((\lambda z.\, zz)\,(\lambda w.\, w)) &\to_1 (\lambda x.\, y)\,((\lambda w.\, w)\,(\lambda w.\, w)) \\ &\to_1 (\lambda x.\, y)\,(\lambda w.\, w) \\ &\to_1 y\end{aligned}$$

Is there a shorter derivation that yields the same outcome of y? Hint: There is!

5 Programming With Lambda Calculus

Despite its simplicity, lambda calculus as discussed so far can amazingly encode data (e.g., Booleans and the natural numbers) and programs that work on the data, without any extra machinery—this all can be accomplished within the existing setup of lambda calculus.

In the following, we will develop pieces so that we can begin conveniently programming with lambda calculus. Let's start off with Booleans.

5.1 Boolean

We begin by encoding arguably the most fundamental unit of data: a bit. In a sense, this is storing a truth value: true or false. Let us define

$$\mathbf{T} = \lambda xy.\, x$$
$$\mathbf{F} = \lambda xy.\, y$$

Upon closer examination, we see that **T** can be seen as a function that takes two arguments—left and right—and returns the left argument. Symmetrically, **F** can be seen as a function that takes two argument, like before, and returns the right argument.

At first glance, it may not be clear why this is a useful thing at all. To gain a better understanding of how this works, we will implement an **and** function. Remember that the logical **and** operator joins two Boolean values and yields the value true when both the arguments are true. Hence, the goal is for **and** to behave as follows:

x	y	**and** $x\,y$
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

To implement **and**, we will start by writing down the shape that **and** should take. Because **and** takes two arguments, we expect it to look like

$$\mathbf{and} = \lambda xy.\, \_\_\_\_\_$$

Now if we take a close look at the table, it is apparent that when the first argument (x) is false (**F**), the output is always false. Knowing this and the fact that **F** is itself a function that returns its right argument, we can refine **and** to be

$$\mathbf{and} = \lambda xy.\, x \underbrace{(\quad\quad)}_{\text{when } x \text{ is true}} \mathbf{F}$$

By doing so, when x is false, the **and** function returns **F**. *What should happen when x is true?* Further thought reveals that when x is true, the outcome of **and** is the value of y. Hence, we can further refine **and** to be as follows:

$$\mathbf{and} = \lambda xy.\, x\, y\, \mathbf{F}$$

It is worth pointing out a few things:

- We don't claim that **and** M N evaluates to anything of meaning unless both M and N are true or false (as defined above).
- There is nothing unique about the **and** term above. It is just one of the many ways the and function can be implemented.

The **or** function can be implemented similarly and is left as an exercise to the reader (See the exercise section).

If-Then-Else. Since most sane programming languages have a way of implementing the if-then-else construct, we will attempt to define that here now. The specification will be

$$\textbf{if\_then\_else}\,\mathbf{T}\ M\ N \to_* M$$
$$\textbf{if\_then\_else}\,\mathbf{F}\ M\ N \to_* N$$

A moment's thought tells us that this is pretty easy to accomplish. Let us define

$$\textbf{if\_then\_else} = \lambda xyz.\ x\ y\ z,$$

which is equivalent to simply $\textbf{if\_then\_else} = \lambda x.\ x$ if we further think about it.

Following this discussion, we have a way to encode true (**T**) and false (**F**) and functions for **and**, **or**, and **not**. Next time, we will tackle the next challenge: encoding numbers and putting in support for recursion.

6 Exercises

1. Reduce this as much as possible
$$((\lambda x.\ x\ x)\ (\lambda y.\ y))\ (\lambda y.\ y)$$
(Hint: There are "variable reuse," which should be renamed.)
2. Come up with a lambda term for **or**.