Lambda Calculus and its Impact on Computer Science

Joris Klaasse Bos
Zaanlands Lyceum

October 10, 2020

1 Prologue

The reason I chose this subject is that it combines two things I enjoy: abstract maths and computer science. I have been programming for about five to six years now. I mainly enjoy low-level programming, so, naturally, C is my most used language and I am most familiar with a simple procedural paradigm. Such a paradigm is, however, not always very easy to use when working with very large and complex systems. I, as many, started out with Object-Oriented Programming, but I did not like that very much. Therefore, I have been exploring alternative paradigms, including Data-Oriented Programming and Functional Programming. I am quite familiar with Data-Oriented Programming and the Rust programming language by now, but Functional Programming isn't something I've ever really got into yet. I did find out about lambda calculus and combinatory logic, which intrigued me, but I haven't got into it beyond a basic level of understanding. That is why I decided to research it for this paper.

2 Introduction

Lambda calculus is, as its name suggests, a calculus. A calculus is a system of manipulating symbols, which by themselves don't have any semantic meaning, in a way that is somehow meaningful. We all know algebra. Algebra itself doesn't have an innate meaning, but we can use it to represent and solve real world problems. Algebra, however, is limited. Not every problem can be represented in algebra. There are many branches of mathematics that use different systems. One example would be formal logic, which is used for logical operations on booleans. Another such example is lambda calculus.

2.1 A short history

2.2 The syntax

Lambda calculus is all about unary anonymous functions. Such a function has no name to identify it, takes only one input, and returns a single expression that is only dependent on the input, so it doesn't have any outside state. A simple function definition in lambda calculus looks as follows:

$$\lambda a.a$$

The lambda signifies a function. Everything following it will be part of that function's definition. The a before the . is the name of the argument. There is only one, because, as I said before, all functions in lambda calculus are unary. Everything following the . is part of the function body, which is the return expression. The funcion above is the identity function in lambda calculus; it just returns the input. This is the equivalent of multiplying by one, or defining a function like $f(x) = x$, or multiplying a vector with the identity matrix; it does nothing.

But how do we use this function? Well, just like defining a function, it is quite simple. If you want to apply this function to a symbol, you just put it in front of the symbol. Something like this:

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

Which evaluates to x, because you remove the x and then replace all the a's in the function body with x and then remove the function signifier and argument list.

In this case you need parentheses around the function, otherwise x would be considered part of the function's body, which it isn't. It's also important to note that lambda calculus is left-associative, that is, it evaluates an expression from left to right. This means that the function on the far left of an expression gets invoked first.

I have now basically explained the entire lambda calculus, it is really that simple. I have explained abstraction (functions), application (applying functions), and grouping (parentheses), which is basically all we need. You can also give names to expressions. We could name our identity function I as follows:

$$I := \lambda a.a$$

But this isn't really part of the core lambda calculus anymore, just some syntactic sugar. This way, instead of constantly having to write $\lambda a.a$, we can just write I. So instead of writing:

$$(\lambda a.a)x = x$$

We could use our previous definition of I and write:

$$Ix = x$$

We have now covered identifiers too.

But if this is all there is, how can this possibly be Turing complete? How do we do boolean logic, or algebra? How can we do things with only unary anonymous functions? What are a and x supposed to represent? If there is no concept of value, how do we even use this meaningfully?

Well, the key is this: a function can return any expression, so even other functions, not just a single symbol. We can start composing these simple functions into more complex functions. Let's say that we wanted to have a function that takes two arguments, and then applies the first argument to the second one. You are probably asking yourself a few questions. For example, what does it mean for one argument to be applied to another? Well, as I said, these arguments are expressions and can thus be functions themselves. But the biggest question you are probably asking yourself is: how can you have a function that takes two arguments?

Well, we actually can't, but what we can do is to have a function that takes one argument and returns another function that takes one argument. We can define the function as follows:

$$\lambda a.\lambda b.ab$$

We currently have a function definition inside the body of another function. If we now apply this function to a symbol like x, we get this:

$$(\lambda a.\lambda b.ab)x = \lambda b.xb$$

We get a new function that takes an argument and applies x to it. If we now apply this function to a symbol like y, we get this:

$$(\lambda b.xb)y = xy$$

Alternatively, we could write it all on one line:

$$(\lambda a.\lambda b.ab)xy = (\lambda b.xb)y = xy$$

xy in this case is what we would call the *β-normal form* of the preceding expressions. That just means that it is in the simpelest form and isn't able to be evaluated any further. Reducing an lambda expression to the β-normal form is called *β-reduction.*

You can start to see how we can compose unary anonymous functions to create more complex functions. In this example we used two nested unary functions to get the same result you would with a binary function. Such a nested function is often called a *Curry'd function.*

You might think that to yourself that having this many nested functions can be quite convoluted and not very readable, and you're quite right. That's why people often use a shorthand notation. They would basically write it as if it is a single binary function (or any n-ary function). They would write the example function above as:

$$\lambda ab.ab$$

Do keep in mind that even though this looks and, for the most part, acts as if it is a single binary function, but it really isn't. It still is a Curry'd function that feeds in the arguments one-by-one, but this way the expression becomes more readable and easier to think about conceptually.

Congratulations, you now know the very basics of lambda calculus. You may still not see how this is Turing complete or how this can be useful and meaningful. You might also already see some of the intrigues of lambda calculus; how simple it is, how it doesn't have a concept of value or data, how everything is an expression, etc. But we'll get to all of that eventually. If you get this, everything else will follow naturally (mostly).

2.3 Combinatory logic

Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic. That basically means a form of logic without values, just like in lambda calculus, but just pure logical expressions, using so called *combinators*. The idea of combinators first came from Schönfinkel, and was later rediscovered by Curry. Combinators are just symbols, in this case letters, that perform operations on symbols that succeed it. We have actually looked at one of these combinators already.

2.3.1 Identity

The first combinator we'll cover is I. It does exactly the same thing as our I function we defined in lambda calculus in the previous subsection. In fact, all combinators can be defined in lambda calculus. This combinator may seem quite useless, but it is actually quite usefull when composing combinators, which we'll come to soon.

2.3.2 The mockingbird

The next combinator we'll cover is M. All it does is repeat its one argument twice. It can be defined in lambda calculus as follows:

$$M := \lambda f.ff$$

We could, for example, look at what happens when you apply M to I. We get:

$$MI = II = I$$

Or, written out in lambda calculus:

$$(\lambda f.ff)\lambda a.a = (\lambda a.a)\lambda a.a = \lambda a.a$$

What happens if you apply M to M? You get:

$$MM = MM = MM = ...$$

and so on to infinity. Or in lambda calculus:

$$(\lambda f.ff)\lambda f.ff = (\lambda f.ff)\lambda f.ff = (\lambda f.ff)\lambda f.ff = ...$$

This expression cannot be evaluated. We say that it doesn't have a β-normal form. In lambda calculus and combinatory logic not every expression is reducable. As Turing found out, there is no single algorithm to decide wether a lambda expression has a β-normal form. This is called the *halt-problem*.