Lambda Calculus and its Impact on Computer Science

Joris Klaasse Bos
Zaanlands Lyceum

12th October 2020

Contents

1 Preface

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

With the decline of OOP, many other paradigms are gaining in popularity. One increasingly popular paradigm is functional programming. Functional programming is fundamentally based on lambda calculus and it has been seeping into other paradigms and into mainstream languages. Most of the popular languages now implement lambda functions and have ways to write in a more declarative style of programming. In this paper I will look at all the ways lambda calculus is being used in computer science and how it may be used in the future.

3 Introduction to lambda calculus

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.

3.1 A short history

3.2 The syntax

Lambda calculus is all about first-class higher-order pure (anonymous[1]) unary functions. Such a function takes a single input, and returns a single expression that is only dependent on the input, so it doesn't have any outside state. Such a function can take and return any expression, which in lambda calculus is always a function. 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.

[1] The core lambda calculus has no way of naming functions.

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. It is a simple process of substitution.

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 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 functions to create more complex functions[2]. 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 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 (as with 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, 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, how it is stateless, etc. But we'll get to all of that eventually. If you get this, everything else will follow naturally (mostly).

3.3 Combinatory logic

Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic (Wikipedia 2020a). 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 (1924), and was later rediscovered by Curry (1930). Combinators are just symbols, in this case letters, that perform operations on symbols that succeed it. We've actually looked at one of these combinators already.

3.3.1 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 ($I := \lambda a.a$). In fact, all combinators can be defined in lambda calculus. This combinator may seem quite useless, but it is actually quite useful when composing combinators, which we'll come to soon.

3.3.2 The omega combinator

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:

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

[2]That's what makes them higher-order (and first-class).

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 (1936, 1937) proved[3]: there is no single algorithm to decide wether a lambda expression has a β-normal form. This is called the *halting problem*[4].

MM is sometimes called the Ω combinator. Omega, because it is the end of the Greek alphabet. The M combinator is sometimes called the ω combinator because of this. These combinators often have many different names. Sometimes because scientists discovered them seperatly, unaware of eachother, sometimes because scientists like to give them pet names.

3.3.3 The constant combinator

The next combinator we'll cover is K. It is a combinator that takes two arguments and returns the first. We can easily define it in lambda calculus as follows:

$$K := \lambda ab.a$$

Remember that we defined this as a curry'd function. This means we can give it just one argument and get a new function out of it. Let's say we apply K to 5:

$$K5 = (\lambda ab.a)5 = \lambda b.5$$

Our new function, $K5$, is a function that takes an argument and returns 5. This means that whatever we apply this function to, we always get 5. K gets its name from the German word *Konstant*, meaning constant. You can probably see why.

Just like with the previous combinators, it'll prove very useful, much more so than you'd expect.

3.3.4 The kite

Here is where things get a little spicier. Our next combinator is KI. It takes two arguments and returns the latter. We can define it in lambda calculus as follows:

$$KI := \lambda ab.b$$

You may already be thinking about its name. Why does it have two letters? And why are they two letters we've talked about already? Well, the answer is very simple. If you apply K to I, you get KI. Don't believe me? Let's try!

If we use our definition of KI and apply it to xy we get:

$$KIxy = (\lambda ab.b)xy = y$$

But if we use the K and I combinators seperately, we get the following:

$$KIxy = (\lambda ab.a)Ixy = (\lambda b.I)xy = Iy = y$$

[3]This is not what he proved specifically.

[4]To be precise: In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever (Wikipedia 2020b).

If you think about it, it is very logical. If K takes two arguments and returns the first, then, in this case, it uses up both I and x and returns I, which will just return the next argument, in this case y. KI will always return the second symbol after the I, because—again—the first gets used up by K.

We can also just see what function we get when we apply K to I:

$$KI = (\lambda ab.a)I = \lambda b.I = \lambda b.\lambda a.a = \lambda ba.a$$

We get our definition of KI (except the names of the arguments are switched).

We're starting to define combinators as combinations of other combinators. Every combinator, in fact, can be defined as a combination of other combinators. That's why they are called combinators.

3.3.5 The starling

3.3.6 The cardinal

3.3.7 The bluebird

3.3.8 S and K

3.3.9 To Mock a Mockingbird

3.3.10 Equality

4 Using lambda calculus for computation

4.1 Binary logic

4.2 Church numerals

5 Functional programming (lambda calculus applied)

5.1 Laziness

5.2 Haskell

6 Functional programming in other paradigms

7 Possibilities for the future

8 Afterword

References

Curry, Haskell Brooks (1930). "Grundlagen der kombinatorischen Logik". In: *American journal of mathematics* 52.4, pp. 789–834.

Schönfinkel, Moses (1924). "Über die Bausteine der mathematischen Logik". In: *Mathematische annalen* 92.3-4, pp. 305–316.

Turing, Alan Mathison (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363, p. 5.

— (1937). *On computable numbers, with an application to the Entscheidungsproblem; a correction*. Royal Society.

Wikipedia (2020a). *Combinatory logic—Wikipedia, The Free Encyclopedia*. URL: `https://web.archive.org/web/20200920164914if_/https://en.wikipedia.org/wiki/Combinatory_logic` (visited on 20/09/2020).

— (2020b). *Halting problem — Wikipedia, The Free Encyclopedia*. URL: `https://web.archive.org/web/20201005163636/https://en.wikipedia.org/wiki/Halting_problem` (visited on 05/10/2020).