Introduction to Lambda Calculus

CSCI 3434: Theory of Computation, Fall 2023

Luke Sellmayer

12/20/2023

1 Abstract

The heart of lambda calculations is functions and evaluating them. In 1936, Alonzo Church was the first to show that there is no solution to the *Entscheidungsproblem* using his lambda calculus. While lambda calculus has mostly been overtaken by the Turing Machine in terms of modeling computation, some of its key ideas have persisted even to this day.

2 Introduction

The history of lambda calculus begins with the *Entscheidungsproblem*, the "decision problem". Originally posed by mathematicians David Hilbert and Wilhelm Ackermann in 1928, the problem asks whether, given a set of axioms and some proposition, if said proposition is provable or not from those axioms; equivalently, does there exist a universal algorithm that can prove whether propositions are True or False using a given set of mathematical axioms?[3]

Before work could be done on solving the *Entscheidungsproblem*, it was clear that the meaning of "algorithm" would have to be given a formal definition. Algorithms are finite sequences of instructions used to perform computation; perhaps the most recognized model of computation is the *Turing Machine*, invented in 1936 by Alan Turing. Since Turing machines are able to recreate any computer algorithm [4], Turing was able to use his machines to prove the *Entscheidungsproblem* was undecidable [5].

However, before Turing's famous proof, another computation model had already been invented and used to prove the undecidability of the *Entscheidungsproblem*: lambda calculus, invented by Alonzo Church in 1932 and used on the *Entscheidungsproblem* in 1935 [1].

This paper seeks to define lambda calculus, its notation and how to encode mathematical statements with it. Church's Theorem on the *Entscheidungsproblem* will also be covered.

3 Definition and Basic Notation

The heart of lambda calculations is functions and evaluating them. All of lambda calculus is written using *lambda expressions*, notated here as $\langle \lambda\ expr \rangle$. Lambda calculus can be expressed using only three rules in a grammar:

$$\begin{aligned}\langle \lambda\ expr \rangle \rightarrow\ & \langle var \rangle \\ \mid\ & \lambda\, \langle var \rangle \,.\, \langle \lambda\ expr \rangle \\ \mid\ & \langle \lambda\ expr \rangle\, \langle \lambda\ expr \rangle\end{aligned}$$

The first production creates a *variable*, which will be some input to a function.

The second production is *function abstraction*, or function definition. In regular mathematical notation, to describe a function f taking input x and mapping it to x^2, we would write $f : x \mapsto x^2$. In lambda calculus, we would instead write $\lambda x.\ x^2$. All functions in lambda calculus are *anonymous*: they don't have names.

The third production is *function application*, or function evaluation. To evaluate our function f (as defined previously) on some input x, we would write $f(x)$ in our regular mathematical notation. In lambda calculus, we would instead write $\left(\lambda x.\ x^2\right) x$. The left lambda expression is the function, while the right lambda expression is its argument. Parentheses have been included to clarify which is the function and which is its argument.

4 Reductions

Lambda calculus would probably not be very useful if we did not have a way to calculate lambda expressions. Luckily, there are three main *reductions* that we can perform to evaluate lambda expressions: $\alpha - conversions$, substitutions, $\beta - reductions$.

$\alpha - conversions$ allow *bound* variables to be renamed: for examples we can perform an $\alpha - conversion$ on $\lambda x.\ x$ to obtain $\lambda y.\ y$, or $\lambda x.\ x \to_\alpha \lambda y.\ y$, and we say that the first lambda expression is $\alpha - equivalent$ to the second one. We can only perform this conversion on *bound* variables. For example, we **cannot** do $\lambda x.\ \lambda y.\ x \to_\alpha \lambda y.\ \lambda y.\ y$, as these two lambda expressions have different meanings.

Substitutions are similar to $\alpha - conversions$, but instead replace all occurrences of *free* variables in a lambda expression with another lambda expression. Notationally, for some lambda expression M containing free variable x, we can replace x with another lambda expression N, which we express as $M\,[x := N]$.

$\beta - reductions$ are the heart of evaluating lambda expressions. We define the $\beta - reduction$ of the lambda expression $(\lambda x.\ M)\, N$ in terms of the substitution $M\,[x := N]$. For example, for we can perform a $\beta - reduction$ on the lambda expression $(\lambda n.\ 2 \times n)\, 7$ to get 2×7: $(\lambda n.\ 2 \times n)\, 7 \to_\beta 2 \times 7$.

5 Boolean Logic

Defining True and False in lambda calculus is a little unusual: True is a function that returns it's first argument, while False is a function that returns it's second argument.

$$T \equiv \lambda xy.\ x \tag{1}$$

$$F \equiv \lambda xy.\ y \tag{2}$$

Using these definitions, we can define the boolean operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$) as follows:

$$\neg \equiv \lambda xy.\ xyF \tag{3}$$

$$\wedge \equiv \lambda xy.\ xTy \tag{4}$$

$$\vee \equiv \lambda x.\ xFT. \tag{5}$$

6 Representing Numbers

In order to represent numbers, we can use an encoding known as *Church Numerals*:

$$\begin{aligned}
0 &\equiv \lambda f.\ \lambda x.\ x \\
1 &\equiv \lambda f.\ \lambda x.\ fx \\
2 &\equiv \lambda f.\ \lambda x.\ f(fx) \\
3 &\equiv \lambda f.\ \lambda x.\ f(f(fx)) \\
&\vdots
\end{aligned}$$

7 Basic Arithmetic

The successor function takes as input some number n and returns $n+1$:

$$S \equiv \lambda xyz.\ x(xyz). \tag{6}$$

This can be used to define multiplication and addition:

$$n + m \equiv nSm \tag{7}$$

$$n \times m \equiv n(mS)0 \tag{8}$$

8 Recursion

The *Y-Combinator* function can be used to write recursive functions:

$$Y \equiv \lambda f.\ (\lambda x.\ f(xx))(\lambda x.\ f(xx)) \tag{9}$$

The Y-combinator usually will not terminate when applied to single variable functions. By using multivariable functions with one of the variables used as an index, the Y-combinator can work like a *while* or *for* loop used in imperative programming languages.

9 Church's Theorem

In 1936, Alonzo Church was the first to show that there is no solution to the *Entscheidungsproblem* using his lambda calculus [2]. Church was able to do so by demonstrating that there is no algorithm that, taking as input any two lambda expressions, is able to decide whether one of those expressions reduces to another.

The proof uses contradiction as well as the fact that, given any set A of lambda expressions which is non-trivial and closed, that set A does not have a decider, i.e. there is no lambda expression d that can decide whether or not some other lambda expression a is in the set A.

The proof outline is as follows: suppose there is a lambda expression q that takes as inputs the two encodings of lambda expressions a and b, represented as $enc(a)$ and $enc(b)$ respectively. A set A is created containing a and all of its possible $\beta - equivalent$ lambda expressions (all possible lambda expressions that can be $\beta - reduced$ from a), and we note that A is closed as well as non-empty, since it contains the term a and all of its possible $\beta - reductions$.

Now, A cannot contain all possible lambda expressions. We can demonstrate this by considering two cases: either a can be reduced to a normal form or it cannot. If a can be reduced to a normal form, then any terms there cannot be in any lambda expressions in A that do not reduce to a normal. On the other hand, if a cannot be reduced to a normal form, then any lambda expressions already in a normal form cannot be in A.

In either case, A must be non-trivial, as it is neither empty nor contains all possible lambda expressions. Since A is closed and non-trivial, it must not have a decider, which contradicts our assumption.

10 Conclusion

Lambda calculus is a model of computation that was successfully created and used by Alonzo Church to provide a negative answer to the *Entscheidungsproblem*, accomplished by demonstrating that there is no algorithm that can decide whether two lambda expressions can be $\beta - reduced$ to one another.

While lambda calculus has mostly been overtaken by the Turing Machine in terms of modeling computation, some of its key ideas have persisted even to this day. Functional programming languages, such as LISP, Haskell, and Scala all make use of lambda calculus in order to provide imperative programming language features such as loops through the use of the Y-Combinator. These functional programming languages are still used extensively in industry, such as at Google, Microsoft, Discord, and Intel. Furthermore, even modern imperative programming languages, such as Python and MATLAB, have anonymous lambda functions as features. While lambda calculus may be described as the smallest universal programming language, it clearly has made a very large impact on the field of computer science.

References

[1] Alonzo Church. “A set of postulates for the foundation of logic”. In: *Annals of Mathematics* 33 (1932), pp. 346–366. DOI: 10.2307/1968337.

[2] Alonzo Church. “An unsolvable problem of elementary number theory”. In: *American Journal of Mathematics* 58 (1936), pp. 345–363. DOI: 10.2307/2371045.

[3] Wilhelm Ackermann David Hilbert. *Principles of mathematical logic*. Springer-Verlag, 1928. ISBN: 0-8218-2024-99.

[4] Michael Sipse. *Introduction to the Theory of Computation*. McGraw-Hill Book Company, 1997. ISBN: 0-07-061726-0.

[5] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 58 (1936), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.