The Lambda Calculus: A Historical and Practical Tour

Richard B. Elrod

Student, Dept. of Mathematics, Youngstown State University, Youngstown, Ohio
E-mail address: rbelrod@student.ysu.edu

2010 *Mathematics Subject Classification.* Primary 03B40, 68N18;
Secondary 68N15, 03B15, 03D10

Key words and phrases. Lambda calculus, logic, functional programming,
programming languages, type theory, Alonzo Church, Alan Turing

Contents

Preface

The present paper, written in partial fulfillment of the requirements for the History of Mathematics course at Youngstown State University, will guide the reader on a tour through an important development in the field of mathematical logic, namely the Lambda Calculus, which has since contributed to the advancement of the fields of *computer science* and *type theory* among others.

After reading this paper in its entirety, the reader should come away with an understanding of why the Lambda Calculus was created and which problems it was meant to solve, how it was developed, where it and variations of it have been applied and the significance of these in these other fields, and knowledge (but not full proofs) of several important related theorems.

As the requirement for this paper states that "[our] goal should be to make the material [being presented] clear to a sophomore mathematics major who has not previously studied this material," we take take a fairly high-level approach throughout the paper, and specifically we avoid several proofs which the interested reader will likely want to explore upon completion of the present paper. References related to these will be mentioned in the text, and provided in the bibliography.

The paper intends to adhere to the style guidelines set forth by the American Mathematical Society (AMS).

We wish the reader enjoyment reading about this important branch of mathematical history.

Richard B. Elrod

Part 1

Pre-History

CHAPTER 1

Frege and "Currying"

1.1. Gottlob Frege

In the spirit of Rosser [**Ros82**] who himself plays an important role in our story, we begin our journey with Gottlob Frege, a German mathematician and logician born on November 8, 1848. Focusing primarily on the intersection of philosophy and mathematics (i.e., the philosophy of mathematics) alongside mathematical logic, Frege "discovered, on his own, the fundamental ideas that have made possible the whole modern development of logic and thereby invented an entire discipline." [**Dum18**]

In 1891, Frege gave a lecture [**Fre91**] to the *Fenaischen Gesellschaft für Medizin und Naturwissenschaften* in which he discussed the notion of functions and their relation to his former notion of *concepts*, and in which he began to assign specific logical rules for how functions operate. He put forth that the concept of a function had been used in areas such as analysis originally, but that the original understanding of them might have been summarized by saying something such as "A function of x was taken to be a mathematical expression containing x, a formula containing the letter x." It was only later that a more strict definition came to be down, and the purpose of Frege's lecture and eventual article was to narrow down laws which provide a more meaningful definition of what a function is.

Two years later in 1893, Frege published [**Fre93**] the first part of a series called *Grundgesetze der Arithmetik* (Basic Laws of Arithmetic), which used his own formulation of functions to state and proove theorems regarding arithmetic. Within this piece is a section which, using his notation[1], effectively shows how functions which take two arguments can be thought of as two functions which each take one argument. This idea, elaborated upon in the subsequent section, was explored further by Moses Schönfinkel [**Sch20**] and later by Haskell Curry [**CF58**], after whom the technique is now named.

1.2. Curry: Food? Mathematician? Or A Useful Technique For Working With Functions?

As we will see later, Lambda Calculus was originally created to aid in the study of functions. In the original lambda calculus, every function took exactly one argument. While this might seem like a limitation at first glance, in this section, we examine why that is not the case.

Although in lower-level mathematics courses, it can be easy to forget that functions themselves are first-class mathematical objects, let us remember this fact. In fact, let us recall what a (total) function is from a formal, set-theoretic foundation.

[1] a functional notation which is vastly different from what we use today, but likely to be found interesting by a reader of the present paper.

DEFINITION 1.1. A **function** $f : X \to Y$ is a set of 2-tuples (x, y), so that $x \in X$ and $y \in Y$, such that every element of X is the first component of exactly one of the tuples in the set.

Let us consider for a moment functions which take two parameters, such as $f : (X, Y) \to Z$ (or to use a slightly more conventional notation, $f : X \times Y \to Z$). Then elements of the set described in the definition above must have the form $((x, y), z)$, where $x \in X$, $y \in Y$, and $z \in Z$. In this case, we can apply the function f to the two arguments x and y, as in $f(x, y) = z \in Z$.

What the technique of *currying* says is that for every function f, we can think of $f : X \times Y \to Z$ as being equivalent to a function $f_{curried} : X \to (Y \to Z)$, and vice versa.

Let us look at an example.

EXAMPLE 1.2. Let $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ be defined by the usual addition operation, $f(n, k) = n + k$.

By currying, this is the same as a function $f_{curried} : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$, defined as follows:

$f_{curried}(x)$ = a function $g : \mathbb{Z} \to \mathbb{Z}$, where g is then defined as $g(y) = x + y$, where x is the original argument to which $f_{curried}$ was applied.

So we have: $f(2, 3) = 2 + 3 = 5$, and:

$f_{curried}(2)$ = a function which when applied to some number adds 2 to it. So we can do: $f_{curried}(2)(3) = 2 + 3 = 5$.

In modern notation, we might express this by: $Z^{X \times Y} \cong (Z^Y)^X$, where the left hand side represents the set of all functions $X \times Y$ to Z and the right hand side represents the set of all functions from X to the set of functions from Y to Z. In modern terminology, we might fall back on advancements in category theory beyond the scope of the present paper, where the concept can be generalized further to categories other than sets and functions[2]. The correspondence can also be defined by other branches of mathematics such as function spaces, topology, homological algebra, domain theory, logic, and type theory.

For the sake of completeness, let us prove it true for set functions.

LEMMA 1.3. *There exists a bijection between* $Z^{X \times Y}$ *and* $(Z^Y)^X$.

PROOF. Let $g : X \times Y \to Z$ be given. Also let $x \in X$ be given, and let the function f_x be given by $f_x(y) = g(x, y)$. Let the function f be given by $f(x) = f_x$. We must show this is one-to-one and onto.

Suppose g_1 and g_2 are given with $f_1(x)(y) = g_1(x, y)$ and $f_2(x)(y) = g_2(x, y)$. If $f_1 = f_2$, then $\forall x. f_1(x) = f_2(x)$. Then $\forall x, y. f_1(x)(y) = f_2(x)(y)$. So $\forall x, y. g_1(x, y) = g_2(x, y)$. So $g_1 = g_2$ which shows one-to-one.

Now let f^* be given. Define g by $g(x, y) = f^*(x)(y)$. Then we get f_x from g by $f_x(y) = g(x, y))$ and f by $f(x) = f_x$. So $\forall x, y. f(x)(y) = g(x, y) = f^*(x)(y)$ and we can claim $f = f^*$. This shows onto, completing the proof.[3] □

[2]In category-theoretic terms, this relationship is a universal property of exponential objects and forms an adjunction in a cartesian closed category. This generalization has many uses, but category theory is a complex topic and is probably not something which should be explained to a sophomore who has yet to discover it.

[3]The notation of the proof actually becomes much nicer if we assume the existence of Lambda Calculus. However since we have not gotten to that yet, we fall back to slightly clumsy set-theoretic notation.

CHAPTER 2

Combinatory Logic

2.1. Lambda Calculus Before Its Time

We now turn our attention to the advent of a logical system known as *combinatory logic* or *combinator calculus*, a logic system and notation developed by Moses Schönfinkel [**Sch20**] in 1920 (published in 1924) and rediscovered by Haskell Curry [**Cur30**] in late 1927, less than a decade before Lambda Calculus entered the scene.

The system was developed for the purpose of eliminating concerns of variable bindings (i.e. variables introduced by quantifiers such as "$\forall$" and "$\exists$") when working with statements and expressions in higher-order logic (HOL).

In combinatory logic, there is a set of primitive *combinators*. Let us define what this means.

DEFINITION 2.1. A **higher-order function** is a function which only uses function application and previously defined combinators to generate its result.

DEFINITION 2.2. Similarly, a **combinator** is a function whose argument is also a function.

In the parlance of Lambda Calculus, we can ascribe to *combinator* a slightly different definition, as we will see in the subsequent chapter.

As combinators each take one argument, let us assume in this chapter that all combinators are curried as discussed in the foregoing chapter. Thus, if we speak of a function taking "multiple parameters", we imply shorthand notation for curried functions which take one parameter each.

2.2. Combinatory Terms

In combinatory logic[1], terms can take one of two forms: P (which is one of the primitive combinators), or $(E_1\ E_2)$ which is the application of the combinatory term E_1 to E_2.

Though various formulations of cominatory logic exist (by modifying which primitive combinators are included in the system), the most common formulation includes three primitive combinators, known as S, K, and I. It is worth noting that Schönfinkel's original system had several other primitive combinators as well, but he proved that all of the others (and in fact I) can be derived from only S and K[**Sch20**].

[1]The information contained in this section is well-known to anyone who has studied this branch of math and computer science to any extent. As such, we don't cite a specific source throughout this section (any introductory combinatory logic book suffices), but the reader interested in the history of the subject should consult [**Sch20**].

Looking at S, K, and I, we define the I combinator to be simply the identity function. Whatever is passed to it is simply returned unchanged, as in $(I\ x) = x$.[2]

The K creates constant functions. That is, given an argument x, $(K\ x)$ is a combinator which takes a parameter y and always returns x, ignoring the newly passed y. Thus, $(K\ x\ y) = x$. Here it is important to note that by convention terms are grouped to the left, so $(K\ x\ y) = ((K\ x)\ y)$.

The S combinator is a generalized version of the substitution function, and is defined as follows: $(S\ x\ y\ z) = (x\ z\ (y\ z))$. That is, it applies the argument z to the arguments x and y, and then applies the result of the latter application to the result of the former.

The astute reader might recognize that a function equivalent to I can be constructed purely out of S and K.

$$\begin{aligned}(S\ K\ K\ x) &= (K\ x\ (K\ x)) \\ &= x\end{aligned}$$

2.3. Connection to Lambda Calculus

"So," the reader might ask, "what has this got to do with Lambda Calculus?" It turns out, as we will see, that it has quite a lot to do with Lambda Calculus. In fact, this combinator calculus is *computationally equivalent* to the Lambda Calculus (which itself is computationally equivalent to Turing machines and to Gödel's notation of generalized recursive functions). But let us not get ahead of ourselves.

The combinator calculus described above (and others derived from it) can be seen as a particular kind of Lambda Calculus, but it is of interest to our story at this point because it plays a critical role in the development of Lambda Calculus.

While Schönfinkel founded the theory of combinatory logic, he only published [**Sch20**] and no other paper on the topic. Only one other paper bears his name [**CH06**]. In fact, [**CH06**] tells use that "By 1927, he was said to be mentally ill and in a sanatorium," due to his later years being spent in "hardship and poverty." He died in 1942. Sadly, the last known of his manuscripts were burned for warmth due to the wartime conditions of the era [**CH06**].

2.3.1. Enter: Haskell Curry. In 1900, Haskell Curry was born and would come to rediscover combinatory calculus independently of Schönfinkel (and of von Neumann who made use of combinatory logic in his work in 1925, though it is not known whether or not his work was derived from Schönfinkel's work, or discovered independently) [**CH06**]. The statement of the substitution rule for propositional logic was missing from Russell and Whitehead's *Principia Mathematica*, which was admitted by Russell later. We let [**CH06**] talk more about Curry:

> Around 1926-27, Curry began to look for a way to break the substitution process down into simpler steps, and to do this, he introduced a concept of combinatoressentially the same as Schönfinkel's.

[2]Note that depending on who is asked, the study of combinatory logic doesn't itself include variables, and it is purely the combinators themselves which are studied. Here and in the following examples, we use variables x, y, ..., for demonstrating how the combinators work when applied to something.

Later in 1927, Curry would discover [**Sch20**]. He would go on to create a formal system based on combinators along with a completeness proof, which used an abstraction algorithm that was the first of its kind. Later, Church [**Chu35**], Rosser [**Ros35**] and Rosenbloom [**Ros50**], would develop competing "simpiler" algorithms, but these would often produce more esoteric (though equivalent) results, and were more inefficient [**CH06**]. When the 1970s came around and interest rose in the use of such systems for practical programming purposes, algorithms similar to Curry's original were once again of interest [**CH06**].

We finish this section by once again defering to [**CH06**]:

> [... By] the end of the 1920s the combinator concept had provided two useful formal techniques: a computationally efficient way of avoiding bound variables, and a finite axiomatization of set theory.

CHAPTER 3

Lambda Calculus

3.1. Enter: Alonzo Church

We move ahead in our journey to Alonzo Church, born in 1903 in Washington D.C.[**CH06**]. Church became a student of Princeton, studying there until 1967 [**CH06**].

Circa 1928, Church began to construct a formal system with the purpose of studying logical foundations while being "more natural" than both the type theory put forth by Russell in an appendix of *Principia Mathematica* and Zermelo's set theory [**CH06**]. He also insisted that the system not contain free variables.

Church decided to base the system on functions instead of sets, and included primitively a notion of abstraction and application. Today the notation used for these are, for example, $\lambda x.\ M$ (for abstraction), and $(F\ X)$ (for application) [**CH06**]. It is worth noting (again per [**CH06**]) that Church was not the first to use an explicit notation for function abstraction, but he was the first to do so alongside a set of conversion rules for the notation, and to study the resulting theory in detail, which would become known as Lambda Calculus.

It was later discovered that Russell had "anticipated" the Lambda Calculus several decades earlier between 1903 and 1905, while trying frantically to develop a solution to Russell's Paradox [**Kle03**]. His notation was different, but many of the concepts were very similar. However beacuse his efforts were so focused on solving the paradox, and this work not leading to a satisfactory result, Russell never published his pre-Lambda-Calculus. In fact, it's believed that Church, though influenced by Russell's work in general, could not have known about Russell's anticipatory calculus.

Church's system given in [**Chu32**] was a type-free one, "with unrestricted quantification but without the law of excluded middle." [**CH06**]

The system included rules for substution (allowing for the replacement of all free variables with some term), α-equivalence (stating equivalence of terms up to a change of bound variables – for example, $\lambda x.\ x$ is the same expression (the identity function) as $\lambda y.\ y$), and β-reduction, which is the primary rule for term-rewriting [**ST18**]. There is also a notion of η-equivalence, which allows for the dropping of lambda terms in cases where they do nothing. For example, $\lambda x.\ Mx$ can be rewritten by η-reduction to simply M.

In a 1933 version of his paper with the same name, Church gave a representation of positive integers using Lambda Calculus terms, which became known as *church numerals*[**Chu33**]. The numbers were represented not primitively, but purely in terms of lambda abstractions as follows[**CH06**]:

- $1 = \lambda xy.\ xy$
- succ $= \lambda xyz.\ y(xyz)$

We note, however, that there is no 0 this original system. In [**Kle36**], Stephen Kleene worked out a slightly different way to define integers, which paved the way for more general recursive functions [**Ros82**]. In [**CR36**] it was known that every function from positive integers to positive integers that is definable in terms of the Lambda calculus is effectively calculable [**Ros82**]. Church then presented a result now known as *Church's Thesis* which showed the converse: Effectively calculable functions from positive integers to positive integers are exactly those definable in the Lambda Calculus. This result provided a "strong, and quite unexpected, version of completeness." [**Ros82**]

Also in [**CR36**], a fairly strong form of consistency is shown for the Lambda Calculus, as part of a proof of a well-known theorem called the Church-Rosser Theorem, which we now state but do not prove[1].

THEOREM 3.1 (Church-Rosser). *For every expression x and y so that x reduces to y, there exists a z so that both x reduces to z and y reduces to z.*

There is, admittedly, much more to say about the development itself of Lambda Calculus and the variations that resulted in the 1930s. For example, the 1933 version of Church's Calculus was found to be plagued with the Richard paradox in [**KR35**]. However, at this point in our journey, we refer the interested reader to [**CH06**] and the references therein, and move on to the application of the Lambda Calculus to the problem of computability and the well-known *Entscheidungsproblem.*

3.2. Entscheidungsproblem

The famous *Entscheidungsproblem* dates back to Hilbert in 1928. The idea is to find an algorithm which takes a first-order logic statement as input and produces a yes-or-no answer as to whether or not the statement is universally valid (valid in every structure satisfying the axioms) [**HA28**]. Put another way, the idea is to find an algorithm which decides whether or not a given statement is provable from the axioms, using only the rules of first-order logic.

In [**Kle36**], a definition by Kleene (attributed to Gödel) is given for that of a *general recursive function* [**Ros82**]. Gödel thought that this concept should be "taken as the criterion of effectively calculable." In the same article, however, Kleene showed that general recursiveness is exactly the same as being definable in the Lambda Calculus, lending strong support to Church's Thesis [**Ros82**].

As this research was happening, Alan Turing had been focusing on developing an abstraction notion of a computer, now known as a Turing machine [**Ros82**] [**Tur36**]. He thought that the notion of effectively calculable should be the same as those functions computable on a Turing machine. However a year later, Turing proved that this concept is the same as being definable in the Lambda Calculus [**Ros82**]. This is why the Lambda Calculus and combinator calculus play such a large role within various fields of computer science [**Ros82**]. This result is known as the *Church-Turing thesis.* As a result, today, Lambda Calculus is used as the foundation for many different functional programming languages and is of interest to those doing work in proof theory and type theory, among other branches that

[1]The paper cited provides the first known proof of the Church-Rosser Theorem, but many proofs have been created and studied since. The result, also known as *confluence* of the Lambda Calculus, has proven extremely important to its study.

lie in the intersection of mathematics and computer science. It is worth noting here that the Lambda Calculus provided the first proof of the unsolvability of the Entscheidungsproblem [**CH06**]. However, Turing's proof via his machines was much more accessible and transparent [**CH06**].

As time went on, other later-found-equivalent definitions of "effectively calculable" were proposed. For information on these, we refer the reader to page 16 of [**Ros82**] and the sources cited therein.

3.3. Type Theory

The present paper would not be complete without a mention of the field of type theory. Originally, as noted above, Church's Lambda Calculus was purely untyped.

DEFINITION 3.2. **Type theory** is a branch of mathematics and computer science originally created by Bertrand Russell as an appendix (appendix B) to *Principia Mathematica* as a means of solving Russell's Paradox. It has since become a field of study in its own right and drives much research in the computer science subfield of programming language theory.

What type theory allowed Russell to do was effectively reject $x \in x$ as a formula in his formulation of set theory. In brief[2], Russell's type theory indexed every inhabitant of the given universe with a type (in his case, a natural number). He then required that for $x \in y$ to be a valid statement, the type of y must be one greater than the type of x.

We are told by [**ST18**] that:

> In 1940, Church gave a "formulation of the simple theory of types which incorporates certain features of λ-conversion" [**Chu40**]. Though the history of simple types extends beyond the scope of this paper and Church's original formulation extended beyond the syntax of the untyped λ-calculus (in a way somewhat similar to the original, inconsistent formulation), one can intuitively think of types as syntactic decorations that restrict the formation of terms so that functions may only be applied to appropriate arguments.

In this sense, types allow us to place restrictions on which terms (functions or programs, in the vernacular of computer science) can be inhabited and under which conditions. It is for this reason that the Lambda Calculus plays such a critical role in programming systems such as automated theorem provers like Coq, Agda, and Isabelle/HOL. Church's original formulation of the concept as applied to Lambda Calculus is known as the "Simply Typed Lambda Calculus."

Type theory has been extended many times over the years, and is still an active area of research in modern branches of computer science and mathematics, such as Homotopy Type Theory, which lives in the intersection of topology, homotopy

[2]The original appendix where type theory first appears describes it better than we can in this brief paper. The interested reader is suggested to refer to that appendix for a historic view of type theory. For modern approaches and uses for it, particularly in computer science, Benjamin Pierce's "Types and Programming Languages" or CMU's Robert Harper's "Practical Foundations for Programming Languages" are excellent resources.

theory, type theory, and mathematical foundations, particularly in an intuitionistic setting, with an axiom known as "Univalence."[3]

[3]This topic, while extremely interesting, is extremely far beyond the scope of this paper. The interested reader should reference the Homotopy Type Theory textbook, which is available as a free download on https://homotopytypetheory.org/.

CHAPTER 4

Closing Remarks

4.1. Parting is such sweet sorrow...

We hope that this paper serves not only to fulfill the requirement for the History of Mathematics class as discussed in the preface, but also to whet the appetite of the reader and spark an interest in not only Lambda Calculus and its applications, but also modern branches of mathematics and computer science which make use of it and its variants.

We note that there is *much* more to be said about every concept and section listed in this paper, and that an extensive survey of the topic would likely be at least double or triple the length of this paper.

Topics that the interested reader might consider researching more about include more information about Simply Typed Lambda Calculus, the "Lambda Cube" which discusses other variants of the Lambda Calculus and their relation to type thoery, different versions of type theory (including Martin-Löf's theory of dependent types, as well as linear type systems), and more about how the connection between Turing's machines, Gödel's general recursive functions, and Church's Lambda Calculus were found to be equivalent in computability power. Those interested particulary in the mathematic applications of Lambda Calculus will be keen to learn about its use as an internal language for Cartesian-closed categories in category theory; and those interested in type theory will note that we did not discuss the Curry-Howard Correspondence, an important correspondence which intimately links types with propositions and programs (functions) with proofs of those propositions.

These topics are necessary for a *complete* overview of the Lambda Calculus, but in the interest of time and conservation of paper, we shall have to say good night, till it be morrow.

Bibliography

[CF58] Haskell B. Curry and Robert Feys, *Combinatory logic*, North-Holland Publishing Company, 1958.

[CH06] Felice Cardone and J. Roger Hindley, *History of lambda-calculus and combinatory logic*, Handbook of the History of Logic, Vol. 5 (2006).

[Chu32] Alonzo Church, *A set of postulates for the foundation of logic*, Annals of Mathematics, Second Series, Vol. 33, No. 2 (Apr., 1932), pp. 346-366 (1932).

[Chu33] ———, *A set of postulates for the foundation of logic*, Annals of Mathematics, Series 2, 34:839–864 (1933).

[Chu35] ———, *A proof of freedom from contradiction*, Proceedings of the National Academy of Sciences of the U.S.A., 21:275–281 (1935).

[Chu40] ———, *A formulation of the simple theory of types*, Journal of Symbolic Logic, 5(2):56–68 (1940).

[CR36] Alonzo Church and J. Barkley Rosser, *Some properties of conversion*, Trans. Amer. Math. Soc., Vol. 39 pp. 472-482 (1936).

[Cur30] Haskell B. Curry, *Grundlagen der kombinatorischen logik (foundations of combinatorial logic)*, American Journal of Mathematics (in German) (1930).

[Dum18] Michael A. E. Dummett, *Gottlob frege — german mathematician and philosopher — britannica.com*, Encyclopaedia Britannica (2018).

[Fre91] Gottlob Frege, *Über funktion und begriff (function and concept)*, Proceedings of the Jena Society for Medicine and Science (1891), Translated by Peter Geach.

[Fre93] ———, *Basic laws of arithmetic*, Oxford University Press, 2016 (originally 1893).

[HA28] David Hilbert and Wilhelm Ackermann, *Grundzüge der theoretischen logik (principles of mathematical logic)*, Springer-Verlag, 1928.

[Kle36] Stephen Cole Kleene, *λ-definability and recursiveness*, Duke Mathematics Journal, Vol. 2 pp. 340-353 (1936).

[Kle03] Kevin C. Klement, *Russell's 1903 - 1905 anticipation of the lambda calculus*, History and Philosophy of Logic, Vol. 24 (2003).

[KR35] Stephen Cole Kleene and J. Barkley Rosser, *The inconsistency of certain formal logics*, Annals of Mathematics, Series 2, 36:630–636 (1935).

[Ros35] J. Barkley Rosser, *A mathematical logic without variables*, Annals of Mathematics, Series 2, 36:127–150 (1935).

[Ros50] Paul Rosenbloom, *The elements of mathematical logic*, Dover, Inc., 1950.

[Ros82] J. Barkley Rosser, *Highlights of the history of the lambda calculus*, Mathematics Research Center, University of Wisconsin-Madison (1982).

[Sch20] Moses Schönfinkel, *On the building blocks of mathematical logic*, Math. Ann. 92 (1924) pp. 305-316 (1920), Written for publication by Heinrich Behmann, Mar. 1924.

[ST18] Shane Steinert-Threlkeld, *Lambda calculi*, Internet Encyclopedia of Philosophy, Stanford University (2018).

[Tur36] Alan M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London Mathematical Society, Series 2, 42:230–265 (1936).