The invention of λ-calculus

Jens E. Pedersen <`jeped@kth.se`>

February 26, 2022

In this essay, I will be exploring the invention of λ-calculus (lambda-calculus) as a groundbreaking method for modelling *computations* in a form that is both mathematically sound *and* practically viable. The enormous impact of λ-calculus is hard to overstate and its dual mathematical and physical nature have unlocked striking correspondences. I chose lambda calculus partly due to this foundational property, but also because I find it interesting to understand how such a scientifically important insight comes to be, and what lessons can be drawn from that to further progress science.

The essay starts with a brief exposition on λ-calculus and a consideration of the pivotal moments in the journey towards lambda calculus, followed by a brief reflection on the necessary conditions for such a pivotal insight. Finally, I conclude by discussing what drove the invention of λ-calculus and how it may help further novel ideas within computational sciences.

A word of warning before we begin; the topic of lambda calculus is complicated and I have purposefully omitted to discuss the efforts of Gottlob Frege, the Burali-Forti paradox, and the Kleene–Rosser paradox. Additionally, I have sacrificed rigorous definition in favour of comprehension and readability (particularly around Gödel's first incompleteness theorem). The interested reading is referred to the references for completeness. In particular, I recommend the articles from the Stanford Encyclopedia of Philosophy [7].

1 The invention of λ-calculus

```
t ::=  x          variable
       λx.t       abstraction
       t t        application
```

Figure 1: Primitives of λ-calculus *terms* (`t`) [11].

Lambda calculus "embodies ... function definition and application in the purest possible form" [11] by distilling computation to its constituent parts, so to speak. Formally, the calculus defines variables, abstractions, and applications that can, hypothetical, describe *any* computation. That is, it is *universal*.

For context, here are two examples: the lambda expression `λx.(x + 1)` corresponds to the more well-known expression $f(x) = x + 1$ and

```
fact = λn. if n=0 then 1 else n * fact(n-1)
```

is the recursive factorial function. Note that the second example conveniently omits the definition of operations such as *if*. That is easily remidied, because the beauty and originality of λ-calculus is exactly that such definitions can be derived directly from the concepts from Figure 1 (see [11] for a more detailed exposition). In the case of if, we simply provide yet another λ expression that *tests* the logical condition (here, $n = 0$) and chooses to evaluate `then` if `True`, otherwise `else`. This was just one example, but as we shall see, massively complex theories can be derived and proven using these rudimentary principles, by methodically constructing λ terms.

In this context, "calculus" is defined as *a means to analyse or evaluate a given mathematical description.* Notice the correspondence between computation and model —not unlike the "original" calculus—where the abstract description *merges* with the practical. Before touching on the implications of such a duality, let us revisit the historical context within which Alonzo Church created this novel calculus to get a solid grasp on the *motivation* for such an invention.

1.1 Early years (< 1930)

"History of Lambda-calculus and Combinatory Logic" 2006 [3] separates the history of λ-calculus into three periods: intense and fruitful studies in the 1920s and 1930s, relative quiet from 1930-1960, and finally from 1960 until this very day, where the λ-calculus was truly put to use.

Symbolic representations of function abstractions has been around long before 1930. One can even argue the concept has existed since the antique, in the form of numbers loosely combined with mathematical operations, such at the calculation of the area of a circle ($f(\text{radius}) \rightarrow \text{area}$) or similarly simple algebra. The Ukrainian mathematician Moses I. Schönfinkel, however, took issue with this way of operating. Schönfinkel studied in Göttingen from 1914-1924 under the famous mathematician David Hilbert, who is known for his studies within the foundations of algebra, geometry, and rigorous proofs about whether a computer could answer *universally valid* statements (the *entscheidungsproblem*). Schönfinkels wanted to eliminate so-called *bound* variable, that ties itself to a certain value or externally enforced "condition". One example of such a bound variable is in the statement "for every *n*, do ...". *n* is *bound* because it is closely tied to the concept of *n*s (whatever they may be), as opposed to being a *free* variable, which can freely be substituted for something else, not connected to the outside world. Free variables are also known as placeholders, and are of particular important in mathematics, because they can be eliminated one by one to produce simpler and cleaner formulas, unlike bound variables.

Schönfinkel envisioned a *language* in which formal reasoning can take place such that *propositions* similar to "A is True" or "B is not A" can be stated, but that is also powerful enough to describe *predicates* that *test* certain propositions, such as "Is A larger than B?" or "Is B not A?". By removing bound variables, he stripped down sentences of reasoning to its constituent parts. In other words, by removing bound variables, Schönfinkel could avoid an "ontological commitment" to completely liberate the singleton logical statement [5].

Schönfinkel succeeded in his goal and published his results in 1924, where he introduced so-called *basic combinators* [12]. The combinators served as *primitive* building blocks (functions) that produced more complex building blocks (still functions), hence untangling the bound variables into free variables. To prove this, he defined a constant combinator **K** as the trivial function that always returns a constant given any argument ($f(x) = c$). By using the application combinator **S** that takes a free variable x, a function f, and an environment Γ and *evaluates* $f(x)$ within the context of Γ ($\Gamma \vdash f(x)$), Schönfinkel could show that all other constructions (e. g. the identity combinator **I**) could be reduced away.

This was a foundational step towards the field of *combinatorial logic* as the study of *unbounded* statements that are *completely independent* from underlying assumptions. In other words, combinatorial logic sets out to reduce logical statements (so-called first-order logic) into single, unprejudiced formulas. From a philosophical point, this was an important step towards an isolation of reasoning about the ontological world [5].

Across the Atlantic, John von Neumann studied the axiomatic set theory. It is unclear whether von Neumann was inspired by Schönfinkel, but since he visited Göttingen an undoubtedly met him, it is a likely influence despite their aims being different [3]. Instead of reducing bound variables, von Neumann set out to axiomatize logical *systems* into primitive *objects*, while ensuring strict logically coherence. By reducing an *object* to a set of functions and their corresponding arguments, he demonstrated combinatorial completeness in around 1925 by composing function applications. As opposed to Schönfinkel's efforts, von Neumann did not avoid bound variables because he did not consider it a goal in itself [3]. The initial steps of his theory would evolve over many iterations (later into the Von-Neumann-Bernays-Gödel set theory), but this kind of function-object classification is considered by many to be the initial steps towards modern algebra and category theory.

To recapitulate, by the end of the 20s, the initial steps to combinatorial logic and category theory were made. These exciting developments paved the way for the *axiomatization* of large and blurry concepts, by decomposing them into transparent and logically sound concepts. One addition to this puzzle is the work of Haskell Curry, who believed that any combinator, in its capacity of combining axioms, should be able to combine *any* axiom—even itself [3]. In modern terms this came to be known as *recursion* and it puts severe constraints on the axioms, since it no longer suffices to consider combinatorial permutations between axioms, but *infinitely* looping combinatorial permutations.

1.1.1 Russel's paradox and Gödel's incompleteness theorem

Two critical obstacles are known to lie on the path towards to the continued development of airtight axiomatic reasoning. It goes without saying that such developments require rock solid logic consistency. That is, the systems *cannot* face unforeseen paradoxes when combining axioms in esoteric ways. Two such paradoxes have been shown by Bertrand Russel in 1901 and Kurt Gödel in 1931, and they are important to be aware of before we venture further towards the invention of λ-calculus.

Russel's paradox goes as follows.

1. Let **R** be the set of all sets that are *not* members of themselves: $R = \{x | x \notin x\}$.
2. If **R** is not a member of itself, then it must be a member of itself by definition. But if it is a member of itself, then it cannot be a member of itself by definition. Or, $R \in R \iff R \notin R$.

The paradox was formulated by Russel in 1901, but its implications are clear for the ideas of Curry and his recursive axioms. Without a way to mitigate paradoxical "unrestricted comprehension", the theory looses its correctness.

Yet another hindrance towards axiomatic consistency was discovered in 1931 by the Austrian-Hungarian mathematician Kurt Gödel: the *first incompleteness theorem* (the reader is encouraged to discover the second incompleteness theorem in [7]). Briefly, the theorem goes as follows.

1. Define a mapping between any primitive A into its "code number" $\texttt{code}(A) \in \mathbb{N}$. Relations on axioms can now operate directly on the domain of numbers, instead of axioms (details are omitted for the sake of brevity). Recall that in combinatorial logic, proofs (truthful relations) are also primitives. Importantly, the property of being provable reduces to whether or not the number corresponding to the predicate is provable.
2. Define F as the function that checks whether "x is the code number of a proof of the formula with code number x". That is, whether $\texttt{code}(A) = \texttt{proof}(\texttt{code}(A))$.
3. Given an arbitrary function A, we can now establish an equivalence relation between the object D and the function A applied to the *code* of D, since A exists in the mapping above. Note that we are not claiming that D and $A(\texttt{code}(D))$ are equivalent, just that they must have the same truth-value.
4. Apply step 3) to the *inverted* version of $\texttt{proof}(\texttt{code}(A))$, such that we check whether $\texttt{code}(A) \neq \texttt{proof}(\texttt{code}(A))$. This can only be shown if $\texttt{code}(A)$ is *not* provable. Which is a paradox, because we saw in 3) that $\texttt{code}(A)$ can only be true if $\texttt{proof}(\texttt{code}(A))$ is true.

This compact enactment serves to illustrate the seriousness with which logic consistency should be taken. Logic soundness is an obviously nice property to have, however blindly following the axiomatic nature of proofs leads to situations where statements can no longer be proved! Put simply, axiomatic theories risk running into severe problem when they strive for logic completeness.

1.2 The invention of λ-calculus (1930s)

Alonzo Church caught wind of the developments by Schönfinkel, von Neumann, and Curry, and set out to develop a formal system "with the aim of providing a foundation for logic which would be more natural than Russel's type theory ... and would not contain any free variables" [3]. He published his ideas in a 1932 paper, choosing functions as the basic object (similar to von Neumann) and abstraction as the means of application [4]. For seemingly inconsistent reasons, he chose the letter "λ" to represent abstractions and letters (x) to denote objects [3, 9]. To avoid Russel's paradox, he allowed a propositional function **F** to assume a non-binary and undefined value. In other words, he ignores the *law of excluded middle*, that states that every proposition is *either* **True** or **False**, to *postpone* the evaluation of terms that may be impossible to define without encountering a paradox. Optimistically, this can be seen as the first steps towards *lazy* evaluation, a famous property of some modern functional languages (like "Haskell" named after Haskell Curry).

As an example of the clever axiomatization, Church defines the natural numbers as follows:

$$\mathbf{1} \equiv \lambda fx.fx, \quad \mathbf{Succ} \equiv \lambda nfx.f(nfx), \quad \mathbf{n} = \lambda fx.f(\ldots(fx)\ldots)$$

This provides a good example on how bound variables can be completely avoided, so let us walk carefully through it: the λ construction for **1**, takes a function f and a numeral x as input and the returned value—f applied to the numeral *once*—provides the very first natural number, 1. To get the next natural number in line (2), we simply take the same function f and apply x to it twice, now parametrized over which number n we want the **Succ**essor of. Generally, it takes n times to arrive at the **n**th natural number, as shown in the right-most part where we have written out the successive application of f. That is, admittedly, quite tedious to do, but it proves that we can perform this calculation without any bounded variables. In other words, we have shown that natural numbers can be entirely self-contained. Also notice how closely the above syntax relates to the combinatorial logic of Schönfinkel and the functional composition of von Neumann.

After the publication of his 1932 paper, Church continued to extend and test his invention together with his students Stephen Kleene and Barkley Rosser. The initial untyped version of the calculus was abandoned in favour of a typed version, where axioms were associated with type information that restricted the use of abstractions. Such a type system could, for instance, discourage the application of **Succ** to a real number (what is the successor of π?).

In the coming decades, the fundamentals of λ-calculus was improved and extended to improve completeness, cut-elimination (for sequent calculus to deduct a sequence of statements, and to refine the raw theorems. Several derivations of the λ-calculus also were developed, even by Church himself [11, 3], but even after years of development, the basic typed λ-calculus proved surprisingly rich.

One important part of this richness was the discovered correspondence between *calculability* and λ-calculus . The british mathematician Alan Turing showed that his definition of "computable numbers" was equivalent to λ-defined numerical functions. In Church's own thesis, he suggests that any definition in a λ-calculus "exactly captured the informal concept of effective calculability" [3]. As we will discuss below, we are still discovering the implications of that fundamental insight.

Despite this sound success, Church failed to properly address for the incompleteness theorem, which still holds for formal axiomatic theories and remains unaddressed to this day. Debate is ongoing as to what that means for combinatorial logic.

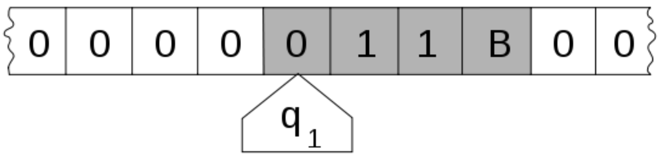

Figure 2: A read/write-head q_1 moving along an infinite tape in The Turing Machine.

1.3 Widespread adaption (> 1960)

The implication of Turing and Church's claims around *universal calculability* was greatly extended in the decades to come. After the initial ideas was standardized and the rough edges of the proofs polished, it became clear that λ-calculus would become pivotal to modern science.

Most significantly is probably the notion of the "computing machine" that lead to a *physically* realizable λ-calculus . Turing was able to realize the ideas from combinatorial logic in the physical domain by analogy of the famous *Turing machine*. A Turing machine can read and write from an infinitely long tape on which a write head can either read or write symbols (objects in λ-calculus), as shown in Figure 2. It turns out that to one particular instance of a λ calculator. A "compute head" then determines what to put on the tape, given what has been read. The ability to only read or write to a tape seems staggeringly simple, but this is exactly what happens in λ-calculus : there are no bound variables, only direct, standalone operations that compose almost indefinitely to produce immensely complicated calculations. The "compute head" is simply a λ-function! This, by the way, also led to the developments of *state machines* that have been heavily applied in other scientific domains such as biology, physics, and even social sciences. Computing machines, and all their derivatives, remain highly active areas of research.

Other innovations based on combinatorial logic and λ-calculus include **polymorphic type systems, linear type systems, higher-order logic, and category theory**. **Polymorphic type systems** allow for the seamless transition between similarly typed objects, again with logical soundness that guarantees the correctness of the program execution. **Linear type systems** goes even further to exploit the type system to provide guarantees about resource utilization within a system. As an example, it is sometimes desirable to only access a resource once, which can be encoded as a typed constraint.

Higher-order logic is, in a sense, a derivation of the computational logic that adds additional quantifiers as well as semantics. This is heavily exploited in modern computational languages to simplify and generalize complicated operations while resting on the logical foundations to, again, ensure correctness. And it even spills over to domains such as linguistics, where compositional logic plays a large part when constructing syntax and semantics.

Finally, many of the ideas from the relatively new but impactful field of **category theory** are rooted in the contributions of von Neumann, Church, Schönfinkel, and Turing. In particular, the strong correspondence between λ-calculus and cartesian closed categories are interesting, because cartesian closed categories have widespread applications ranging from programming, to topology, to quantum computing [1]. The correspondence relies exactly on the fact that functions of two variables can be expressed as functions of a single variable, as we saw with the *reduction* in λ-calculus. This reduction has become more widely known as "currying", named after Haskell Curry).

2 Lessons from the discovery of λ-calculus

We started by discussing the intriguing duality between the *model description* and *practical application* of a mathematical theory. From my perspective, this is exactly why λ-calculus is so important and ubiquitous. Previous to the invention of λ-calculus , there was no way to formally define a model *and* prove that it works in practice. So far, any efforts towards rigorous *computability* were hindered by bound variables and incoherent logic.

The lesson here, I believe, is twofold. First of all, it pays to *get the foundation right*. The sheer impact of λ-calculus as a cognitive and practical tool, with which we can derive new intuitions and build new machines that would have otherwise been completely out of reach, is hard to understate. From that perspective Schönfinkel's visionary insistence of proof simplification by eliminating bounded variables, stands as a pivotal moment of inspiration. At the time, it seems, it was entirely unclear exactly what this could imply. It took Schönfinkel and the group in Göttingen years of hard work to arrive at these insights, without any seemingly practical relevance in terms of physicality or capital. One can only admire the tenacity and foresight the developments of these early pioneers, and the profound foresight of Schönfinkel has garnered some interest in his (illusive) person [14].

The second lesson I draw, is that science is incremental. Before I studied Church's work, I had the impression that his insights was more isolated and, in a way, the product of a lone genius. Everywhere I have encountered λ-calculus, the historical details have been omitted. Upon closer inspection, however, a clear trail of breadcrumbs appears from Church's inventions to von Neumann's functional calculus, Schönfinkel's combinatorial logic, and even further back to calculus, logic, as well as older ideas of "functions" and "algebra". Church was clearly not the sole inventor of the idea, and while that does not make it less profound, it is evidently a distributed efforts between great minds.

This fits the old trope "standing on the shoulders of giants" well. It is vital that we, as scientists, insist on communicating with our community and carefully study the theories and insights of others. Our work is never done in vacuum, as is made evident by the exchange of ideas between the publications of Schönfinkel, von Neumann, and Church (and this was *long* before telecommunications), and it is unlikely that the individual actors would have arrived at equally impactful contributions without these exchanges—particularly between Princeton and Göttingen.

3 Perspectives and future work

I took up the topic of λ-calculus for three reasons. As a computer scientist that daily thinks about what a *computation* is, I wanted to better understand the origins of computational theory. Second, I wanted to understand what drives research into these fundamental questions. And, finally, I am curious about what the future of computational theories will bring us.

3.1 The philosophical nature of computational theories

The essay has provided some insight into the most significant events that lead to Church's invention, but it also elucidated several rabbit holes, left for the curious reader to explore. Particularly, there is much more to be said about the logical inconsistencies, philosophical implications, and the broader historical perspectives before 1900. The logical inconsistencies are important because they reveal weaknesses in the foundation of the combinatorial logic, which undermine the entire goal rigorous, formal systems. The idea of an "ontological commitment" is deeply rooted in the philosophical idea to approach an understanding of what Plato dubbed the "*true*" reality. However, the openendedness of problems similar to the incompleteness theorem spreads doubt that such system will ever be able to adequately combat *all* paradoxes. And then what? Should we abandon all hopes of ever arriving at a perfect computational theory? Or is it simply a matter of shifting perspectives? Such questions are profoundly thought provoking and worthwhile to ponder. Computation, it would seem, is a window into the reality of our nature.

Language	Year introduced	Syntax
Lisp	1958	`(lambda (x) x + 1)`
Haskell	1990	`\x -> x + 1`
Python	1994	`lambda x:  x + 1`
C++	2011	`[x] { x + 1 }`
Java	2014	`(x) -> x + 1`

Figure 3: λ-expressions are abound in some of the most widespread programming languages like C++, Java, and Python. Interestingly, the functional language `Haskell` (that also features lazy evaluation) is named after Haskell Curry.

3.2 What drives research into fundamental questions?

Another reason for starting this journey was to better understand what drove the invention of λ-calculus. It is safe to say that λ-calculus is a corner stone of modern computing. By understanding how it came to be and what motivated the initial work, we may be able to better understand what will fuel the next big scientific advances.

The motivation behind λ-calculus can be distilled into two components: a curiosity for improved abstractions (lack of bounded variables) and a love for consistent logic (rigorous axiomatization) personified by Schönfinkel and von Neumann, respectively. None of the sources mention a desire for a *complete* theory of computation in the early days, and it is doubtful whether Hilbert, Schönfinkel, Gödel, von Neumann, or Church fully understood the far-reaching consequences of their efforts. In publication, Turing was the first to fully express the notion of *computing machines* [13] and discussed how it could determine the *universal validity* of statements—almost a decade after the initial steps towards combinatorial logic. There *was* a striving for completeness, but in a *mathematical* and *geometric* sense driven by Russel and Hilbert [8]. Nothing in the included sources indicates there was a clear striving towards *computability* early on. On the contrary, the advances within computation seemed to "fall out" of the exploration within mathematics, geometry, and logic.

At least for the case of λ-calculus, the answer to the question "what drives research into fundamental question" would be *curiosity* more than anything else. Loosely interpreted from the descriptions of the main characters in our story, it is likely that a stint of stubbornness in the willful pursuit to eliminate asymmetry and "ugliness", such as bound variables, also played its part.

3.3 Future of computational theory

To conclude this essay on computation, I will briefly consider the continuation of computational theory, with an emphasis on programming languages, linear type theory, and finally category theory.

Figure 3 shows the introduction of λ constructs in select programming languages. Lisp implemented λ constructs as early as 1958, but it took 40-60 years before it permeated into mainstream languages like Python, C++, and Java. Point being, the *abstract* computational theory has been far removed from practical (mainstream) programming. It has taken a long while to catch up, but by now all major languages are heavily reliant on λ constructions. Unfortunately, there are limitations to these modern λ constructs that appear *a posteriori*. As an example, C++ was never designed to carry the inherent rigor of combinatorial logic. Hence, any attempt at axiomatizing expressions in C++ will quickly fail. *Pure* languages like Coq[1] (from 1989), Idris[2] (from 2007), and Lean[3] (from 2013) are designed bottom-up to ensure logical rigor. This is an enticing approach because it combines the decades of research into practical and easily under-

[1]`https://coq.inria.fr/`
[2]`https://www.idris-lang.org/`
[3]`https://leanprover.github.io/`

standable programming language syntax with proof assistants. In a way, this continues the developments of Russel, Hilbert, and Schönfinkel towards a simpler and cleaner mathematics—but *computable*.

One reason that languages like C++ are not logically consistent is the use of resources and input/output patterns. It is challenging to formalize external operations like a network outage or disk failure. *Dependent linear type theory* (DLTT) proposes a solution by building on linear logic [6] and dependent type theory [10] to define rules for objects or resources, *depending* on their type or state [2]. In-depth treatment of this topic is outside the scope of this essay, but one fact deserves highlighting: efforts are being made to *formalize* computations *on external resources*. This is immediately applicable to critical systems (aviation, infrastructure, etc.), but I see no reason why we cannot extend the notion of *computation* even further.

The authors of "Physics, Topology, Logic and Computation: A Rosetta Stone" argue that the way that λ-calculus "falls out" of logic and mathematical concepts is no coincidence [1]. Rather, they claim, there are fundamental correspondences between fields, which can be described with some universal language. If there is such a structure—if there is a deeper truth hiding behind these individual fields—then any advances within singular topics must also lead to advances within the others. To me, the causality of the correspondence is slightly unclear; are the topics linked *because* they share the same underlying structure, or are the *structure itself* so common to our way of reasoning that it prevails across domains? The former case brings hope that we can uncover "true ontological structure". Finally, there is of course the case that the authors of "Physics, Topology, Logic and Computation: A Rosetta Stone" is wrong: perhaps there is no "true nature". Oh well. At least we get to solve cool puzzles.

References

[1] John C. Baez and Mike Stay. "Physics, Topology, Logic and Computation: A Rosetta Stone". In: *arXiv:0903.0340 [quant-ph]* (June 2009). arXiv: 0903.0340. URL: http://arxiv.org/abs/0903.0340.

[2] Jean-Philippe Bernardy et al. "Linear Haskell: practical linearity in a higher-order polymorphic language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Jan. 2018). arXiv: 1710.09756, pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3158093.

[3] Felice Cardone and J Roger Hindley. "History of Lambda-calculus and Combinatory Logic". In: *Handbook of the History of Logic* 5 (2006), p. 95.

[4] A. Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932).

[5] *Combinatorial Logic*. Stanford Encyclopedia of Philosophy, Nov. 2020. URL: https://plato.stanford.edu/entries/logic-combinatory/.

[6] Jean-Yves Girard. *Linear logic*. 1987. URL: https://www.sciencedirect.com/science/article/pii/0304397587900454?via%3Dihub.

[7] *Gödel's Incompleteness Theorems*. Stanford Encyclopedia of Philosophy, Apr. 2020. URL: https://plato.stanford.edu/entries/goedel-incompleteness/.

[8] *Hilbert's Program*. Stanford Encyclopedia of Philosophy, May 2019. URL: https://plato.stanford.edu/entries/hilbert-program/.

[9] *Lambda Calculus*. Stanford Encyclopedia of Philosophy, Mar. 2018. URL: https://plato.stanford.edu/entries/lambda-calculus/.

[10] Per Martin-Löf. “An Intuitionistic Theory of Types: Predicative Part”. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by H. E. Rose and J. C. Shepherdson. Vol. 80. Logic Colloquium ’73. Elsevier, Jan. 1975, pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08719451.

[11] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Jan. 2002. ISBN: 978-0-262-16209-8.

[12] M. Schönfinkel. “On the building blocks of mathematical logic”. In: *J. van Heijenoort, From Frege to Gödel. A Source Book in Mathematical Logic* (1924), pp. 355–366.

[13] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. ISSN: 1460-244X. DOI: 10.1112/plms/s2-42.1.230.

[14] Stephen Wolfram. *Where Did Combinators Come From? Hunting the Story of Moses Schönfinkel*. arXiv:2108.08707. arXiv:2108.08707 [math] type: article. Aug. 2021. DOI: 10.48550/arXiv.2108.08707. URL: http://arxiv.org/abs/2108.08707.