The Lambda Calculus—Historical Developments*

Samuel Frontull   Christina Kohl   Fabian Mitterwallner

September 5, 2022

Contents

1 Introduction **1**

2 Pure λ-Calculus **1**
2.1 λ-terms and β-reduction . . . 2
2.2 Church encoding . . . 3

3 Early Days of Lambda Calculus **7**
3.1 The Kleene–Rosser Paradox . . . 8

4 A Selection of Important Achievements Related to λ-Calculus **9**
4.1 The Church–Rosser Theorem . . . 9
4.2 The Church–Turing Thesis . . . 9
4.3 Solution to Hilbert's Entscheidungsproblem . . . 10

5 Curry Paradox **11**
5.1 Connection to Lambda calculus . . . 12
5.2 Possible solutions . . . 13

6 Conclusion **14**

1 Introduction

One could say that lambda calculus is as old as computer science itself. In fact many programming languages are based on its key ideas, many proof assistants are built on it, and various research fields—such as proof theory, category theory, or linguistics—use it as a tool [Acc19]. In this paper we trace the evolution of λ-calculus from its birth in the 1930s to some important results in the field obtained in the late 30s and early 40s. We believe that investigating the history of this prevalent field of research can give us substantial insight into more recent applications. In accordance with the overarching topic of the course, which prompted us to write this paper, special emphasis is placed on the emergence and treatment of different paradoxes in (variants of) λ-calculus.

We start by describing λ-calculus as it is defined today in Section 2. There we first give a definition of its terms and reduction rule and then show how numbers and arithmetic operations can be encoded in it. In Section 3 we start to dive into the history of λ-calculus. We describe the original idea behind its invention, the problems that arose early on, and how they are connected to various well-known paradoxes. In Section 4 we present three important results concerning λ-calculus: the Church–Rosser theorem, the Church–Turing thesis, and the solution to Hilbert's Entscheidungsproblem. We devote a separate section to the Curry paradox, which evolved out of the paradoxes described in Section 3, and which permits great insights into potentially inconsistent logics. In Section 5 we describe this paradox and its relation to λ-calculus.

2 Pure λ-Calculus

The λ-calculus is a very abstract, formal system which in its simplicity makes it difficult to think about something concrete. In the λ-calculus functions are first class citizens, i.e., in principle

*Final Essay for 705501 VU Paradoxes, University of Innsbruck

everything is a function. This is also the core concept of functional programming languages. This paradigm seems to be far away from our intuition. How can lists, truth values or natural numbers be encoded as functions? This question requires a high level of abstraction in λ-calculus. In the λ-calculus we basically only have variables, abstractions, applications, and a single computation rule called β-reduction. It is surprising that a formal system with just these ingredients can be so expressive. In this section we give an introduction to the pure λ-calculus and, as an illustration of its expressive power, we show how Peano arithmetic can be encoded in it.

To get started a little easier, let's look at the following `add` function:

```
add(x,y) := x + y
```

This function takes *any* two numbers as input and returns their sum, e.g. `add(1,3) = 4`. In the λ-calculus we could express this function as λ-term[1]:

$$\lambda xy.+ \, x \, y$$

where the λxy can be seen as the function head and $+ \, x \, y$ as the function body. If we want to compute the sum of the number 1 and 3 we can supply them to the function as arguments:

$$((\lambda xy. + x\,y)\; 1)\; 3 \quad \triangleq \quad add(1,3)$$

which corresponds to the expression `add(1,3)` from above. It is clear that the argument 1 is meant to belong to the parameter x and the argument 3 to the parameter y. When evaluating such λ-terms, these variables in the function body are replaced by the respective argument supplied to them. However, in contrast to the usual function calls, where we pass all parameters at once, in the λ-calculus this is done step by step:

$$((\lambda xy. + x\,y)\; 1)\; 3 \quad \rightarrow \quad (\lambda y. + 1\,y)\; 3 \quad \rightarrow \quad + \, 1 \, 3$$

The application of the arguments (substitution in the function body) is what we call β-reduction. Even if it has been pretended so far, it is not true that the expression $((\lambda xy. + x\,y)\; 1)\; 3$ is a valid λ-term. The $+$-symbol and the numbers 1 and 3 are not part of the syntax of the λ-calculus. Therefore, we need to find a way to represent these symbols as λ-terms. In Section 2.2 we will look at how this can be done, but before that we will discuss the various components of the λ-calculus in a bit more detail.

2.1 λ-terms and β-reduction

In the λ-calculus we only have variables, abstractions and applications. The set Λ of λ-terms is defined by the following grammar:

$$M ::= x \mid (\lambda x.M) \mid (M\;M)$$

variable (for x), abstraction (for $(\lambda x.M)$), application (for $(M\;M)$)

where the application associates to the left and nested abstractions can be combined for readability purposes.

Example 1. The following are valid λ-terms. On the rhs we omit "useless" brackets and combine nested abstractions:

$$(x\,y) \triangleq x\,y$$
$$(\lambda x.(x\,y)) \triangleq \lambda x.x\,y$$
$$(((\lambda x.(\lambda y.(x\,y)))\,y)\,x) \triangleq (\lambda xy.x\,y)\,y\,x$$

Two λ-terms are said to be α-equivalent ($\equiv_\alpha$) when they vary only by the names of the bound variables.

Example 2. We have $\lambda xy.x \equiv_\alpha \lambda yz.y$ but $\lambda xy.y \not\equiv_\alpha \lambda yz.y$

As already suggested above, whenever we have a λ-term of shape $(\lambda x.M)\,N$, we can supply the argument N to the function $\lambda x.M$ (β-reduction). We call such terms *redexes*.

Example 3. $(\lambda x.x)\,y$ is a redex, $\lambda x.x\,y$ is not a redex.

[1] it actually is not a valid λ-term because $+$ is not part of the language

Formally, the contraction of a redex is defined as follows:

$$C[(\lambda x.M)\,N] \;\to_\beta\; C[M[\![x \mapsto N]\!]]$$

where C denotes an arbitrary context and $M[\![x \mapsto N]\!]$ the capture-avoiding substitution that replaces all free occurrences of the variable x in M by N.

To understand why we need a *capture-avoiding* substitution, we will show how variables can be confused in the λ-calculus and what happens if we do not avoid this so-called *variable capture*. Suppose we have a capture-permitting substitution $M[x \mapsto N]$ as defined next.

Definition 4. The capture-permitting substitution $M[x \mapsto N]$ is inductively defined as follows:

$$\begin{array}{rcl} x[x \mapsto N] & := & N \\ y[x \mapsto N] & := & y \\ (e_1\, e_2)[x \mapsto N] & := & e_1[x \mapsto N]\, e_2[x \mapsto N] \\ (\lambda x.e)[x \mapsto N] & := & \lambda x.e \\ (\lambda y.e)[x \mapsto N] & := & \lambda y.e[x \mapsto N] \end{array}$$

From a simplistic point of view, α-equivalent expressions are the same [dB72]. It doesn't matter whether we call the parameters of the add function x, y or a, b, as illustrated next:

$$\texttt{add(x,y) := x + y} \qquad \equiv_\alpha \qquad \texttt{add(a,b) := a + b}$$

We therefore want α-equivalent terms to be interchangeable. More precisely, if we substitute a free variable x in a λ-term M or in an α-equivalent one M', by a λ-term N, then we want the following property to be true: $M[x \mapsto N] \equiv_\alpha M'[x \mapsto N]$. Unfortunately this property is not true for the capture-permitting substitution as shown in Example 5.

Example 5. Consider the λ-terms $M = \lambda y.x$ and $M' = \lambda z.x$. We have $M \equiv_\alpha M'$, but $M[x \mapsto N] \not\equiv_\alpha M'[x \mapsto N]$ for $N = y$, as $M[x \mapsto N] = \lambda y.y$ and $M[x \mapsto N'] = \lambda z.y$.

Allowing such variable captures would would lead to an inconsistent system. We therefore need to avoid them. This is done in the capture-avoiding substitution, where abstractions are renamed before applying the substitution, whenever this could be the case.

Definition 6. The capture-avoiding substitution $M[\![x \mapsto N]\!]$ is inductively defined as follows:

$$\begin{array}{rcl} x[\![x \mapsto N]\!] & := & N \\ y[\![x \mapsto N]\!] & := & y \\ (e_1\, e_2)[\![x \mapsto N]\!] & := & e_1[\![x \mapsto N]\!]\, e_2[\![x \mapsto N]\!] \\ (\lambda x.e)[\![x \mapsto N]\!] & := & \lambda x.e \\ (\lambda y.e)[\![x \mapsto N]\!] & := & \lambda y.e[\![x/N]\!] \text{ if } y \notin N \\ (\lambda y.e)[\![x \mapsto N]\!] & := & \lambda z.e[\![y \mapsto z]\!][\![x \mapsto N]\!] \text{ if } y \in N, \\ & & \text{where } z \text{ is fresh for } e \text{ and } N \end{array}$$

The interesting cases are the last two, where we check whether variables may get captured or not. In the case we could have a variable capture (last case), we introduce a fresh variable z (does not occur in e and N) which we use to rename the "unsafe" abstraction.

Example 7. Consider again the λ-terms M, M' from Example 5. We have $M[\![x \mapsto N]\!] \equiv_\alpha M'[\![x \mapsto N]\!]$ also for $N = y$, as $M[\![x \mapsto N]\!] = \lambda z'.y$ and $M[\![x \mapsto N']\!] = \lambda z.y$.

2.2 Church encoding

Church encodings, due to Alonzo Church, are a way to represent data and operations in the λ-calculus. The Church numerals are a Church encoding of the natural numbers. In this section we will discuss how Church numerals are constructed and show how arithmetic operations can be defined on them. In the examples shown in this section we do not follow a specific reduction strategy when evaluating[2] λ-terms but try to reduce them in an order[3] that causes less "confusion".

[2]reduce them to normal form

[3]The reduction order is irrelevant, as the pure λ-calculus is confluent, as we will see later on.

Church numerals As mentioned before, in the λ-calculus we can't just use numbers the way we are used to, because numbers are not part of the language, but need to represent them via some λ-term. This representation has to be chosen in a way such that numbers behave the way we expect them to behave. The Church numerals are a way to represent the natural numbers in the λ-calculus. A number n is represented by a higher-order function which applies n times a function f to an input x (a generalization of the Peano numbers). Some examples:

$$\begin{aligned} 0 &\triangleq \lambda fx.x \\ 1 &\triangleq \lambda fx.f\,x \\ 2 &\triangleq \lambda fx.f\,(f\,x) \\ &\dots \\ n &\triangleq \lambda fx.f^n\,x. \end{aligned}$$

Successor One of the most basic operations on Church numerals is the successor function. Given a number n it returns the number $n+1$. In our representation this means to add one function application.

$$succ \triangleq \lambda nfx.f\,(nfx)$$

Example 8. In the following we show how the successor of the Church numeral 3 is computed. We highlight the redex that is contracted by underlining it:

$$\begin{aligned} succ\,2 &\triangleq \underline{(\lambda nfx.f\,(n\,f\,x))\,(\lambda fx.f\,(f\,x))} \\ &\to_\beta \lambda fx.f\,(\underline{(\lambda fx.f\,(f\,x))\,f}\,x) \\ &\to_\beta \lambda fx.f\,(\underline{(\lambda x.f\,(f\,x))\,x}) \\ &\to_\beta \lambda fx.f\,(f\,(f\,x)) \triangleq 3 \end{aligned}$$

Addition We can add two Church numerals n, m, by embedding one into the other so that at the end we have $n+m$ function applications. This can be done as follows:

$$add \triangleq \lambda nmfx.n\,f\,(m\,f\,x)$$

Example 9. In the following we show how we can add the number 3 to the number 1:

$$\begin{aligned} add\,1\,3 &\triangleq \underline{(\lambda nmfx.n\,f\,(m\,f\,x))\,(\lambda fx.f\,x)\,(\lambda fx.f\,(f\,(f\,x)))} \\ &\to_\beta \underline{(\lambda mfx.(\lambda fx.f\,x)\,f\,(m\,f\,x))\,(\lambda fx.f\,(f\,(f\,x)))} \\ &\to_\beta \lambda fx.\underline{(\lambda fx.f\,x)\,f}\,((\lambda fx.f\,(f\,(f\,x)))\,f\,x) \\ &\to_\beta \lambda fx.\underline{(\lambda x.f\,x)\,((\lambda fx.f\,(f\,(f\,x)))\,f\,x)} \\ &\to_\beta \lambda fx.f\,(\underline{(\lambda fx.f\,(f\,(f\,x)))\,f}\,x) \\ &\to_\beta \lambda fx.f\,(\underline{(\lambda x.f\,(f\,(f\,x)))\,x}) \\ &\to_\beta \lambda fx.f\,(f\,(f\,(f\,x))) \triangleq 4 \end{aligned}$$

Multiplication The multiplication of two Church numerals n, m can be expressed in a straight-forward way – by making n copies of the number m.

$$\lambda nmfx.n\,(m\,f)\,x$$

Example 10. In the following we show how we can multiply the numbers 2 and 3:

$$\begin{aligned}
mul\,2\,3 &\triangleq \underline{(\lambda nmfx.n\,(m\,f)\,x)\,(\lambda fx.f\,(f\,x))}\,(\lambda fx.f\,(f\,(f\,x)))\\
&\to_\beta \underline{(\lambda mfx.(\lambda fx.f\,(f\,x))\,(m\,f)\,x)\,(\lambda fx.f\,(f\,(f\,x)))}\\
&\to_\beta \lambda fx.(\lambda fx.f\,(f\,x))\,(\underline{(\lambda fx.f\,(f\,(f\,x)))\,f})\,x\\
&\to_\beta \lambda fx.\underline{(\lambda fx.f\,(f\,x))\,(\lambda x.f\,(f\,(f\,x)))}\,x\\
&\to_\beta \lambda fx.\underline{(\lambda x.(\lambda x.f\,(f\,(f\,x)))\,((\lambda x.f\,(f\,(f\,x)))\,x))\,x}\\
&\to_\beta \lambda fx.(\lambda x.f\,(f\,(f\,x)))\,(\underline{(\lambda x.f\,(f\,(f\,x)))\,x})\\
&\to_\beta \lambda fx.\underline{(\lambda x.f\,(f\,(f\,x)))\,(f\,(f\,(f\,x)))}\\
&\to_\beta \lambda fx.f\,(f\,(f\,(f\,(f\,(f\,x))))) \triangleq 6
\end{aligned}$$

Exponentiation The exponentiation of two Church numerals n, m can be encoded surprisingly simple. By applying the number m to n we get as result n^m.

$$exp \triangleq \lambda nm.m\,n$$

Example 11. In this example we show how we can compute 2^3 in the λ-calculus:

$$\begin{aligned}
exp\,2\,3 &\triangleq \underline{(\lambda nm.m\,n)\,(\lambda fx.f\,(f\,x))}\,(\lambda fx.f\,(f\,(f\,x)))\\
&\to_\beta \underline{(\lambda m.m\,(\lambda fx.f\,(f\,x)))\,(\lambda fx.f\,(f\,(f\,x)))}\\
&\to_\beta \underline{(\lambda fx.f\,(f\,(f\,x)))\,(\lambda fx.f\,(f\,x))}\\
&\to_\beta \lambda x.(\lambda fx.f\,(f\,x))\,((\lambda fx.f\,(f\,x))\,(\underline{(\lambda fx.f\,(f\,x))\,x}))\\
&\to_\beta \lambda x.(\lambda fx.f\,(f\,x))\,(\underline{(\lambda fx.f\,(f\,x))\,(\lambda y.x\,(x\,y))})\\
&\to_\beta \lambda x.(\lambda fx.f\,(f\,x))\,(\lambda y.(\lambda y.x\,(x\,y))\,(\underline{(\lambda y.x\,(x\,y))\,y}))\\
&\to_\beta \lambda x.(\lambda fx.f\,(f\,x))\,(\lambda y.\underline{(\lambda y.x\,(x\,y))\,(x\,(x\,y))})\\
&\to_\beta \lambda x.\underline{(\lambda fx.f\,(f\,x))\,(\lambda y.x\,(x\,(x\,(x\,y))))}\\
&\to_\beta \lambda x.\lambda y.(\lambda y.x\,(x\,(x\,(x\,y))))\,(\underline{(\lambda y.x\,(x\,(x\,(x\,y))))\,y})\\
&\to_\beta \lambda x.\lambda y.\underline{(\lambda y.x\,(x\,(x\,(x\,y))))\,(x\,(x\,(x\,(x\,y))))}\\
&\to_\beta \lambda x.\lambda y.x\,(x\,(x\,(x\,(x\,(x\,(x\,(x\,y))))))) \triangleq 8
\end{aligned}$$

Predecessor Initially it was not clear whether it was possible to compute the predecessor of a Church numeral in the λ-calculus. This problem remained open for years until Kleene [Kle35] was able to find a solution for it. This was not an easy task and it is maybe even a bit paradoxical that it can be done, because somehow one has to derive the Church numeral $n-1$ by applying a function n times. However, this problem can be circumvented by deriving the predecessor in a bottom up manner using pairs of numbers. A pair $\langle a, b\rangle$ can be defined as follows:

$$pair = \lambda abs.s\,a\,b$$

It is a function that takes the two elements and a selector function. The selector function allows to select the first (*fst*) or the second (*snd*) element of the pair:

$$fst = \lambda xy.x \qquad snd = \lambda xy.y$$

Example 12. Suppose we construct a pair of the Church numerals 1 and 2. Then it results in:

$$\begin{aligned}
pair\ 1\ 2 &= \underline{(\lambda abs.s\,a\,b)\,(\lambda fx.f\,x)}\,(\lambda fx.f\,(f\,x))\\
&\to_\beta \underline{(\lambda bs.s\,(\lambda fx.f\,x)\,b)\,(\lambda fx.f\,(f\,x))}\\
&\to_\beta \lambda s.s\,(\lambda fx.f\,x)\,(\lambda fx.f\,(f\,x))
\end{aligned}$$

With the selector function we could then for example extract the first element of this pair by applying the *fst* function to it:

$$\begin{aligned}(\mathit{pair}\ 1\ 2)\ \mathit{fst} &= \underline{(\lambda s.s\,(\lambda fx.f\,x)\,(\lambda fx.f\,(f\,x)))\ (\lambda xy.x)}\\ &\rightarrow_\beta \underline{(\lambda xy.x)\,(\lambda fx.f\,x)}\,(\lambda fx.f\,(f\,x))\\ &\rightarrow_\beta \underline{(\lambda y.(\lambda fx.f\,x))\,(\lambda fx.f\,(f\,x))}\\ &\rightarrow_\beta \lambda fx.f\,x \triangleq 1\end{aligned}$$

i	fst	snd
0	$\langle 0,$	$0\rangle$
1	$\langle 0,$	$1\rangle$
2	$\langle 1,$	$2\rangle$
3	$\langle 2,$	$3\rangle$

Figure 1: Derivation of the predecessor – Idea illutrated.

By using such pairs, it is possible to derive the predecessor of a Church numeral. This approach works bottom up starting from a pair $\langle 0,0\rangle$ and then following the procedure[4] illustrated in Figure 1. Starting from the pair $\langle 0,0\rangle$, the subsequent pairs are generated by copying the second element to the first one and increasing the second element by one. This is what is encoded in the *next* function:

$$\mathit{next} = \lambda p.\mathit{pair}\ (p\ \mathit{snd})\ (\mathit{succ}\ (p\ \mathit{snd}))$$

In this definition and in the following definitions and examples we will use the name of already defined functions to make everything easier to read – the reader should be aware that in these places we should actually write the corresponding λ-term. For example, the *next* function written out actually looks like:

$$\mathit{next} = \lambda p.(\lambda abs.s\,a\,b)\ (p\ (\lambda xy.y))\ ((\lambda nfx.f\,(nfx))\ (p\ (\lambda xy.y)))$$

The predecessor of a specific Church numeral n can then be computed by applying the *next* function n times to the pair $\langle 0,0\rangle$ and then returning the first element of the final pair.

$$\mathit{pred} = \lambda n.(n\,\mathit{next}\ (\mathit{pair}\,0\ 0))\,\mathit{fst}$$

Example 13. In this example we show how the predecessor of the Church numeral 2 is computed:

$$\begin{aligned}\mathit{pred}\,2 &= \underline{(\lambda n.(n\,\mathit{next}\ (\mathit{pair}\,0\ 0))\,\mathit{fst})\ (\lambda fx.f\,(f\,x))}\\ &\rightarrow_\beta ((\underline{\lambda fx.f\,(f\,x))\ \mathit{next}}\ (\mathit{pair}\,0\ 0))\,\mathit{fst}\\ &\rightarrow_\beta (\underline{(\lambda x.\mathit{next}\,(\mathit{next}\,x))\ (\mathit{pair}\,0\ 0)})\,\mathit{fst}\\ &\rightarrow_\beta (\mathit{next}\,(\underline{\mathit{next}\,(\mathit{pair}\,0\ 0)}))\,\mathit{fst}\\ &\rightarrow_\beta (\underline{\mathit{next}\,(\mathit{pair}\,0\ 1)})\,\mathit{fst}\\ &\rightarrow_\beta (\mathit{pair}\,1\ 2)\,\mathit{fst}\\ &\rightarrow_\beta 1\end{aligned}$$

Subtraction With the predecessor function it is then pretty straightforward to implement the subtraction operation. To compute the subtraction $a-b$ we apply b times the *pred* function to the Church numeral a which is encoded in the following function:

$$\mathit{sub} = \lambda ab.b\,\mathit{pred}\,a$$

[4]similar to the one of the *Euclidean* algorithm

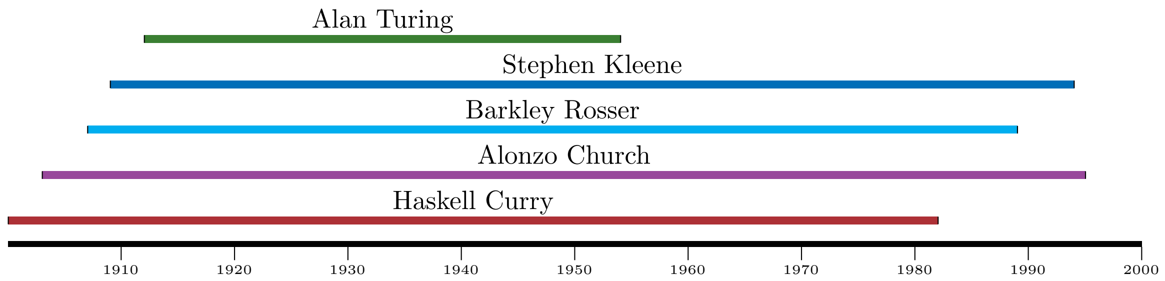

Figure 2: Lifetime of the protagonists in this paper.

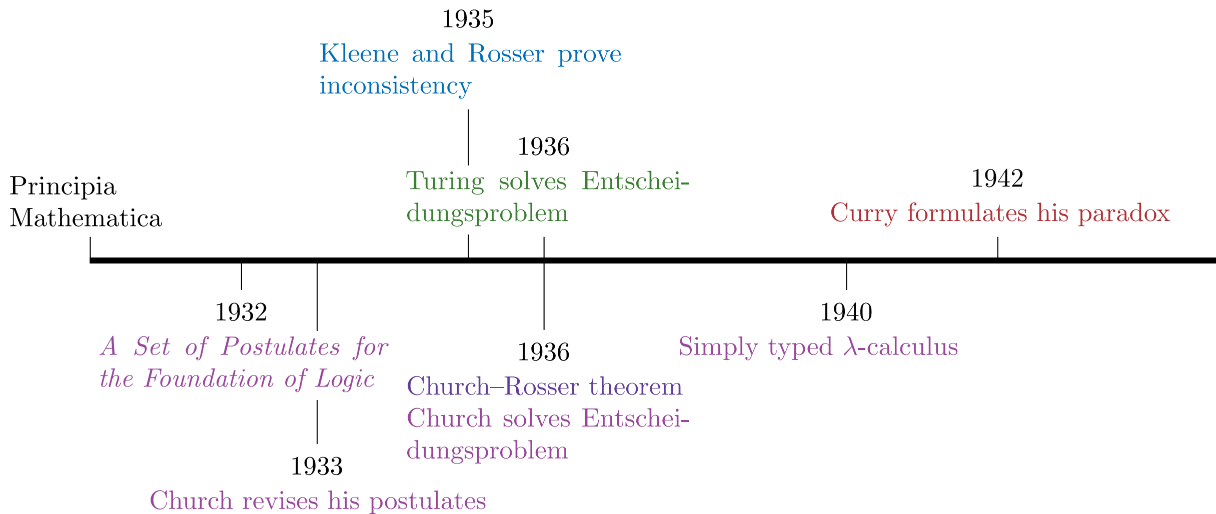

Figure 3: Timeline of the events covered in this paper. Colors correspond to color of the protagonists in Figure 2.

Example 14. In this example we show how we can compute the subtraction $3 - 1$:

$$\begin{aligned}
sub\,3\,1 &= \underline{(\lambda ab.b\,pred\,a)\,(\lambda fx.f\,(f\,(f\,x)))}\,(\lambda fx.f\,x) \\
&\to_\beta \underline{(\lambda b.b\,pred\,(\lambda fx.f\,(f\,(f\,x))))\,(\lambda fx.f\,x)} \\
&\to_\beta \underline{(\lambda fx.f\,x)\,pred}\,(\lambda fx.f\,(f\,(f\,x))) \\
&\to_\beta \underline{(\lambda x.pred\,x)\,(\lambda fx.f\,(f\,(f\,x)))} \\
&\to_\beta \underline{pred\,(\lambda fx.f\,(f\,(f\,x)))} \\
&\twoheadrightarrow_\beta \lambda fx.f\,(f\,x)
\end{aligned}$$

3 Early Days of Lambda Calculus

In this section we give details about the birth of lambda calculus in the 1930s. We describe Church's original aim, when first inventing it, and the problems with his formulation. Figure 2 provides an overview of the lifetime of the protagonists. Important events covered in this and the following sections are displayed in Figure 3.

In 1932 Alonzo Church introduced a formal theory intended as a foundation for logic which was based on the notion of function [Chu32]. His atomic constants were all logical operators, Π for a

relative universality, Σ for existence, & for conjunction, $\sim$ for negation, a descriptive operator ι[5], and A for a kind of abstraction operator. Well-formed formulas were formed from these atomic constants and variables by means of application and λ-abstraction [Sel06]. Church states in the introduction of his paper "…it is hoped that the system will turn out to satisfy the conditions of adequacy and freedom of contradiction". Church hoped to avoid contradictions by introducing a restriction on the law of excluded middle. For example in the case of Russell's paradox the system is designed such that the truth value of the contradictory proposition is undefined. However shortly after the publication, Church already discovered a contradiction in the form of a modified version of Russel's paradox. So just one year later in 1933 he presented a revised version of his postulates [Chu33], omitting the incriminating postulate and several others with it and therefore reducing the overall number of postulates from 37 to 32. Also in this paper Church explicitly mentions his hope of being able to prove the consistency of his system even though he was aware of Gödel's incompleteness results.

> [Gödel's] argument, however makes use of the relation of implication U between propositions in a way which would not be permissible under the system of this paper, and there is no obvious way of modifying the argument so as to make it apply to the system of this paper. [Chu33]

3.1 The Kleene–Rosser Paradox

As it turned out, Church's system was problematic after all. As Kleene, who was one of Church's students at that time, later put it [Kle81]

> Indeed, Church was right! In his system there is a proof of its own consistency, since in fact it is inconsistent.

The inconsistency was shown by Stephen Cole Kleene together with another of Church's students John Barkley Rosser in 1934/35 [KR35]. Kleene had worked on the development of Peano arithmetic in Church's system for his PhD (Section 2.2), while Rosser had worked on its connection to Curry's combinatory logic. In their joint paper *The Inconsistency of Certain Formal Logics* they showed that Church's system of logic as well as combinatory logic are inconsistent "in the sense that every formula in their notation is provable, irrespective of its meaning under the interpretation intended for the symbol" [KR35]. They did this by deriving a variant of Richard's paradox which we briefly describe here.

Richard's Paradox Richard's paradox is a paradox of definability often used to illustrate the importance of distinguishing between mathematics and metamathematics. It was first described by Jules Richard in 1905 [Ric05]. Its construction strongly resembles Cantor's diagonal argument for the uncountability of the real numbers and goes as follows [Cur41]:

We can construct a list of all English phrases that unambiguously define a positive real number. Now we order this infinite list of (finite) phrases first by length of the phrases and then—if two phrases have the same length—lexicographically. This yields the infinite list of real numbers

$$r_1, r_2, r_3, \ldots$$

Now consider the following definition: *Let r be the number whose integer part is 0 and whose nth decimal place is 1 if the nth decimal place of r_n is not 1, and 2 if the nth decimal place of r_n is 1.*

If it is indeed possible to construct this list $r_1, r_2, r_3, \ldots$ then this clearly is an unambiguous definition of a real number and must therefore also appear in the list. However by construction $r \neq r_n$ for all n.

Kleene–Rosser Paradox The connection of Richard's paradox to lambda calculus and combinatory logic lies in the ability of both of these systems to characterize and enumerate their definable and total numerical functions on natural numbers. The following—very simplified—account of it is given in [Cur41]. Let

$$\phi_1(x), \phi_2(x), \phi_3(x), \ldots$$

be such an enumeration. Then consider the function

$$f(x) = \phi_x(x) + 1 \tag{1}$$

[5] $\iota(F)$ expresses "the x such that $F(x)$"

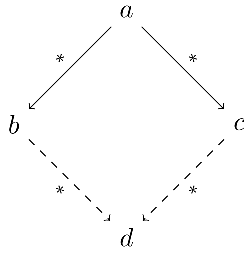

Figure 4: The Church–Rosser theorem.

If f is a definable function then it must appear at some position n in the sequence, so $f = \phi_n$. However, substituting ϕ_n for f and n for x in (1) yields

$$\phi_n(n) = \phi_n(n) + 1$$

which is a contradiction.

The actual proof of this result relies heavily on previous results by Kleene and Rosser and is not so easy to follow. Curry later managed to identify the crucial ingredients of the calculi that allowed the construction of this paradox, and used these to construct a much simpler paradox, now known as Curry's paradox, which we describe in Section 5.

Church, Kleene, and Rosser responded to the inconsistency by focusing on the part of the system involving the formation of lambda terms and the rules for reduction and conversion, without the inconsistent system of logic, i.e., what is now known as the *pure lambda calculus*. This is a surprisingly simple yet powerful system—the notion of *λ-definability* is now accepted as a mathematical description of the class of effectively computable functions (see also Section 4.2 on the Church–Turing thesis). Therefore this simple system is often used to formally prove results in computability theory, like showing that certain well-defined functions are not computable.

4 A Selection of Important Achievements Related to λ-Calculus

We present three of the early landmark results related to λ-calculus.

4.1 The Church–Rosser Theorem

In 1936 Church and Rosser published a proof of the following statement[6] [CR36]:

Theorem (Church–Rosser). If a lambda term a can be β-reduced to terms b and c by some arbitrary sequences of reductions, then there must exist some term d such that both b and c reduce to d.

This statement is depicted in Figure 4. In modern rewriting terminology we would say the reduction rules of lambda calculus are *confluent*. As a consequence of the theorem every lambda term has at most one normal form, and whenever two terms a and b have different normal forms the equation $a = b$ is not a theorem of λ. So the λ-calculus is consistent in the sense that not every equation is a theorem. (Contrary to the original set of postulates proposed by Church, where indeed every statement was a theorem.)

4.2 The Church–Turing Thesis

The *Church–Turing thesis* is a thesis about the nature of *computable functions* or as they were called by Church, Kleene, and Rosser *effectively calculable functions*. A thorough definition of this concept of computability was needed find an answer to the *Entscheidungsproblem* as we will see in the next section. In the 1930s most computations were carried out by humans—not machines, so the term "effectively calculable" was intended to refer to functions that can be calculated by idealized human computers. An attempt to state the meaning of an "effective method" more rigorously could for example look like this [Cop20]:

A method M for achieving some desired result is called *effective* if and only if

1. M is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);

[6]The original theorem was stated by referring to Church's postulates of 1932. We present the modern version.

2. M will, if carried out without error, produce the desired result in a finite number of steps;
3. M can (in practice or in principle) be carried out by a human being unaided by any machinery except paper and pencil;
4. M demands no insight, intuition, or ingenuity, on the part of the human being carrying out the method.

This is still a relatively informal notion, and in order to provide rigorous proofs about what is and is not effectively computable a formal model of computation needed to be agreed upon. From the Church–Rosser consistency property it was clear to Church that every λ-definable function is effectively computable. He also came up with many examples of (intuitively) effectively calculable functions as well as operations that yielded new effectively calculable functions and let his student Kleene prove that all these functions are λ-definable and that the operations preserve λ-definability. This led Church to propose to use λ-definable functions as a definition for effective calculability. This has later been called *Church's thesis* by Kleene. Based on that definition Church was able to answer the Entscheidungsproblem [Chu36a] (see also Section 4.3).

According to a letter (from Church to Kleene), Gödel regarded Church's thesis as "thoroughly unsatisfactory" [Kle81]. Gödel came up with his own formalization of effectiveness based on an idea by Herbrand. The result became known as *Herbrand-Gödel general recursiveness*. Gödel himself was very skeptical about the expressiveness of his formalization, later stating that "However, I was, at the time [1934], not at all convinced that my concept of recursion comprises all possible recursions". Kleene was able to prove equivalence of λ-definability and recursiveness in 1936 [Kle36]. Still, Gödel remained unconvinced of the equivalence of effective calculability, either with recursiveness or with λ-definablity. According to Martin Davis [Dav82] this only changed after Turing published his work concerning *Turing machines* and computability.

In 1936 Turing introduced his so-called *computing machines* in order to provide a (negative) answer to Hilbert's Entscheidungsproblem. This was published mere months after Church had already provided the same answer using λ-definability. Turing also established that the concept of λ-definability coincides with his concept of computability [Tur36]. In the same paper he also provides compelling philosophical arguments (I and II in Section 9 of his paper) for why it should be accepted that the informal notion of computability coincides with computable by Turing machines.

The work by Church et al. on computability can thus be summarized as follows: A function is λ-computable if and only if it is Turing computable and if and only if it is general recursive. If the Church–Turing thesis holds then all functions that would naturally be regarded as computable are computable in all three formalisms.

Several other formalisms for effective calculability have been proposed over the years and have strengthened the thesis by being shown equivalent to the concepts above [Cop20]. Any such formalism is said to be *Turing complete*. As Davis [Dav82] states this is quite remarkable:

> Although it is certainly interesting to attempt to recover the order of events in this fascinating drama of ideas, what is much more interesting than who did what first is the remarkable fact that all of the proposed answers to the question: "Which functions are effectively calculable?" turned out to be correct and equivalent to one another.

4.3 Solution to Hilbert's Entscheidungsproblem

The *Entscheidungsproblem* was first posed by David Hilbert [7] and Wilhelm Ackermann in their 1928 book about logic [HA28]. It was originally formulated specifically for the first-order predicate calculus

(called *"engere Funktionenkalkül"* by Hilbert and Ackermann). but can also be stated more generally [Chu36a]:

> By the Entscheidungsproblem of a system of symbolic logic is here understood the problem to find an effective method by which, given any expression Q in the notation of the system, it can be determined whether or not Q is provable in the system.

Church proved in 1936 that—assuming the *effective methods* correspond exactly to λ-definable functions—this is impossible in first-order logic. This result was published a few months before Turing arrived at the same conclusion—using Turing machines instead of λ-definability to model effective methods. Church's proof consisted of the following ingredients [Chu36b]:

[7] The same Hilbert that had also presented 23 open problems at a congress in 1900. Asking questions seemed to be his specialty.

1. Church showed how the natural numbers can be encoded as λ-terms (see also Section 2.2) and then defined the notion of λ-definability for numerical functions.
2. He devised a Gödel encoding for lambda terms, so that every lambda term corresponds to exactly one natural number. It is important to note that the encoding and its reverse—computing the lambda term corresponding to a given natural number if that number is a valid encoding—can be effectively computed.
3. Then Church assumes that there exists an effectively computable function to determine whether a given term has a normal form or not. If such a function exists then there also exists an enumeration $A_1, A_2, A_3, \ldots$ of the λ-terms which have a normal form.
4. Then one can define a function E such that $E(n) = m + 1$ if $A_n(n)$ reduces to some natural number m and 1 otherwise. The function E is effectively computable and has a normal form.
5. The previous two items lead to a contradiction since E should appear in the enumeration $A_1, A_2, A_3, \ldots$ but $E(n) \neq A_n(n)$ for all n.
6. Hence it can be concluded that it is undecidable whether a given lambda term has a normal form[8].
7. It can be concluded that the Entscheidungsproblem is unsolvable for any logic where the predicate "the lambda term t has a normal form" can be expressed. This includes in particular Hilbert's system.

Turing solved the same problem by a reduction to the halting problem, which he first proved undecidable [Tur36].

5 Curry Paradox

In [Cur41] Curry simplified Kleene's and Rosser's result, showing that Church's lambda calculus is inconsistent. As mentioned in Section 3, it is based on Richard's Paradox and the proof is still very technical. One year later (1942), Curry published another related paper [Cur42]. In it he presents a significantly simpler construction, showing "the inconstistency of certain formal logics". This simpler paradox is now known as *Curry's paradox*[9], and will also be presented in this section.

At the heart of the Curry paradox lie so called *Curry sentences*. These are self referential sentences, similar to the liars paradox which don't rely on negation but on implication. They have the shape

$$A = A \to B$$

for some statement B. Here $\to$ is implication, and $\alpha \to \beta$ is read as "if α, then β".

Example In an attempt to convince reviewers we construct the sentence:

if this sentence is true, then all my papers will be accepted

Assuming such a sentence exists in our theory, and we can replace the sentence by its definition (and vice versa) when reasoning, we can now prove that in fact "all my papers will be accepted". But first we show that the Curry sentence itself holds under these assumptions.

To prove a sentence of the structure "if A then B", we would first assume that the statement A holds. If we can then find a proof that B holds, we have shown that "if A then B", or often also called A implies B.

So to prove the above Curry sentence, we first assume that

this sentence is true (2)

This in turn, by the definition of "this sentence", means that the full sentence

if this sentence is true, then all my papers will be accepted (3)

[8] Undecidable means that there does not exist an effectively computable function whose value indicates whether the given input has a normal form.

[9] A good starting point for further reading is the entry in the Stanford Encyclopedia of Philosophy: https://plato.stanford.edu/entries/curry-paradox/

is also true. We are not done yet, since up until now this only holds under the assumption of (2). However, given (2) and (3) we can then use the *modus ponens* to prove

all my papers will be accepted (4)

Since we have shown that (4) holds under the assumption of (2), we have proven our Curry sentence.

if this sentence is true, then all my papers will be accepted (5)

Using this result we then show that indeed "all my papers will be accepted", by first using the definition of the sentence yielding:

this sentence is true (6)

By applying the *modus ponens* once more to (5) and (6), we finally get

all my papers will be accepted

This conclusion does not immediately seem paradoxical, but that may just be wishful thinking. However, any smart reviewer would likely notice that no part of the argument depends on the content of the "then" part of the sentence. In fact we could use the Curry sentence

if this sentence is true, then none of my papers will be accepted

to prove that "none of my papers will be accepted", by the exact same argument.

Inconsistency To generalize, for some statement B if our theory contains the Curry sentence $A = A \to B$, we can prove B. This must not be a problem, since B may in fact be true. On the other hand, it obviously becomes paradoxical as soon as we have a Curry sentence where B is in fact false. Moreover, if our theory allows us to construct Curry sentences for arbitrary statements, then every statement becomes provable making the logic trivial.

5.1 Connection to Lambda calculus

The Curry paradox is not limited to Churches lambda calculus, but applies to any logic which allows us to construct Curry sentences, and allows the logical inference steps used in the above example. In [Cur41] Curry names two properties of a theory which together make this possible. Or in his words, "both of them are desirable properties of formal systems of mathematical logic", however "any system which possesses both of them is inconsistent".

Deductive completeness This is related to proving correctness of *implications*, and is informally defined as ([Cur41]):

> A theory is deductively complete if whenever we can derive a proposition B on the hypothesis that another proposition A holds then we can derive without hypothesis a third proposition expressing this deducibility.

This third proposition is written as $A \to B$ or "if A, then B" in our case, and directly corresponds to the step (5) in the previous example.

Combinatorial completeness This is related to the construction of terms in our theory, and is informally defined as ([Cur41]):

> A theory is combinatorially complete if and only if every expression M formed from the terms of the system and auxiliary [..] variable x, can be represented within the system as a function of x (i.e., we can form in the system a function whose value for any argument is the same as the result of substituting any argument for x in M).

In our context this means that if M is a valid term the lambda abstraction $\lambda x.M$ is also a valid terms, and the meaning/truth-value of a term is preserved under β-reduction. (i.e. $(\lambda x.M)N = M[N/x]$).

Note that we are not talking about pure λ-calculus, but an enrichment where we add $(\to)$ as a constant to the language, which is interpreted as *implication*. We will also use the infix expression $x \to y$ instead of $(\to)\,x\,y$ ("$\to$ applied to x applied to y"). Importantly the theory has both properties mentioned above. To prove that it is inconsistent, Curry uses two different methods. In both he shows that we can construct Curry sentences for arbitrary statements/terms, meaning we can prove anything.

Method 1 To construct a Curry sentence for some B, we first define the following three terms:

$$R = \lambda x.(x \to B)$$
$$Q = \lambda y.R\,(y\,y)$$
$$A = Q\,Q$$

Note that these three do not use any self-reference in the definitions. They can be seen just as a shorthand to make the presentation easier and everything would still work if we would always write out the full terms. Moreover, all three are valid terms due combinatorial completeness. Since β-steps do not change the meaning of the term A, we can reason a follows:

$$\begin{aligned} A &= (\lambda y.R\,(y\,y))\,Q \\ &(=_\beta R\,(Q\,Q) \\ &= R\,A \\ &= (\lambda x.(x \to B))\,A \\ &=_\beta A \to B \end{aligned}$$

Therefore, A and $A \to B$ have the same meaning, and we can replace one by the other. In other words, our theory contains Curry sentences for arbitrary terms B, making everything provable and the theory inconsistent.

Method 2 The second method uses a *Gödel numbering* of λ-terms to produce the self-referential Curry sentence. There exists a function n which uniquely maps every term M in our logic to a natural number $n(M)$. We call $n(M)$ the *Gödel number* of M. Similarly to a *universal Turing machine* there exists a lambda term T, which takes as input a Church numeral Z_m (see Section 2.2), and whenever m is a valid Gödel number for some term M (i.e. $m = n(M)$), we have

$$T\,Z_m =_\beta M \tag{7}$$

To spare the technical details of the construction of T we refer to [Cur41, Theorem 9.6] for the proof. Using this we construct the Curry sentence for an arbitrary B by first constructing the following term U with the Gödel number u.

$$U = \lambda x.(T\,x\,x \to B)$$
$$u = n(U)$$

The term representing the Curry sentence then is

$$A = U\,Z_u$$

From (7) and the fact that β we can show that our theory contains a Curry sentence.

$$\begin{aligned} A &= U\,Z_u \\ &=_\beta T\,Z_u\,Z_u \to B \\ &=_\beta U\,Z_u \to B \\ &= A \to B \end{aligned}$$

5.2 Possible solutions

To solve the problem of Curry's paradox, one has to limit our logics. Either by disallowing one of the necessary inference steps used in the earlier example, or by limiting the set of constructable terms so that no Curry sentence is contained in our theory.

The latter is usually done using type systems. Within these λ-terms can no longer be arbitrarily constructed, but have to adhere to some set of rules. These rules can be chosen to disallow some of the terms used in the two methods described by Curry. An example of such a type-system is *simply typed* λ-calculus. Simply typed λ-terms can be proven to be terminating (they don't allow infinite sequences of β-steps). In both methods introduced by Curry infinite reductions must exists, since we have $A \to_\beta \cdots \to_\beta (A \to B)$, and we can repeat the same steps on the inner A. However, these limitaitons also gives up on a lot of expressibility, since this system is no longer Turing-complete.

Another approach is to add a third kind of truth value. For example, this is done in *Map theory* [Gru92], which is a newer logic based on λ-calculus. This third truth value is usually called $\bot$ or *undefined* and from a computational point of view corresponds to non-terminating λ-terms like the ones obtained from Curry's paradox (or the Liar paradox).

6 Conclusion

In this paper we gave a short introduction into what λ-calculus is. We focused on its historical context as a formal theory introduced by Church [Chu32]. His hope was for the theory to be a foundation for mathematical logic, while being spared inconsistencies. To show how arithmetic could be done in such an abstract system, we described the structure and operations of Church numerals. Counter to Church's hopes multiple paradoxes were found to be constructible in the theory. The simplest of which is the Curry's paradox [Cur42] as described in this paper.

Even though Church's idea of finding a new foundation for mathematics ultimately failed, λ-calculus has given rise to some well known and interesting results, which are applicable in other areas. The two discussed here where that the calculus was used in the first refutation of Hilbert's Entscheidungsproblem, and as evidence for the Church-Turing theses. Moreover λ-calculus has found many applications in the study of computations (e.g. in modeling programs).

References

[Acc19] Beniamino Accattoli. A Fresh Look at the lambda-Calculus (Invited Talk). In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:20, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[Chu33] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 34(4):839–864, 1933.

[Chu36a] Alonzo Church. A note on the entscheidungsproblem. *The journal of symbolic logic*, 1(1):40–41, 1936.

[Chu36b] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Cop20] B. Jack Copeland. The Church-Turing Thesis. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 edition, 2020.

[CR36] Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[Cur41] Haskell B. Curry. The paradox of kleene and rosser. *Transactions of the American Mathematical Society*, 50(3):454–516, 1941.

[Cur42] Haskell B. Curry. The inconsistency of certain formal logics. *The Journal of Symbolic Logic*, 7(3):115–117, 1942.

[Dav82] Martin Davis. Why gödel didn't have church's thesis. *Information and control*, 54(1-2):3–24, 1982.

[dB72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[Gru92] Klaus Grue. Map theory. *Theoretical computer science*, 102(1):1–133, 1992.

[HA28] D Hilbert and W Ackerman. Theoretische logik. *Julius Springer, Berlin*, 1928.

[Kle35] S. C. Kleene. A theory of positive integers in formal logic. part ii. *American Journal of Mathematics*, 57(2):219–244, 1935.

[Kle36] S. C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340 – 353, 1936.

[Kle81] S. C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, 1981.

[KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[Ric05] Jules Richard. Les principes des mathématiques et le problème des ensembles. *Revue Générale des Sciences Pures Et Appliquées*, 12(16):541–543, 1905.

[Sel06] Jonathan P Seldin. The logic of curry and church. *Handbook of the History of Logic*, 5:819–873, 2006.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.