Philosophisches Seminar der Universität Münster

Sommersemester 2023

Modul: Bachelorarbeit

Erstprüfer: Prof. Dr. Niko Strobach

Zweitprüferin: Dr. Eva-Maria Jung

Functions as Graphs, or Functions as Rules?

A Philosophical Introduction to the λ-calculus

Münster, den 27.07.2023

Vitus Schäfftlein

Zwei-Fach-Bachelor Philosophie und

Kommunikationswissenschaft

Matrikelnummer: 438 467

vitus.schaefftlein@uni-muenster.de

Contents

1 Introduction

In analytic philosophy, talk of functions is ubiquitous. Whether one discusses the meaning of an expression, the truth-conditions of a complex statement or differences in properties across possible worlds – one will have a very hard time avoiding words like "function", "argument" and "value" when spelling things out in a clear and precise manner. For this reason, it is not surprising that a philosopher needs to understand the meaning of the word "function", which is why in any introductory textbook on logic, one will find its standard definition:

(Functions-as-Graphs) A function is a set of argument-value pairs such that every argument has a unique value.

Under this definition, functions are identified with their set-theoretic graphs, as, for example, the successor function:

(σ_G) $\sigma = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, ...\}$

While this way of looking at functions does its job well enough for the purposes mentioned above, it still might not be satisfactory: Our intuitive understanding of functions does not involve sets of argument-value pairs. In fact, students apply and even define complex functions at school without ever having heard the word "set theory". One possible explanation for this circumstance is that what we *do* have in our minds when thinking about functions are *rules*:

(Functions-as-Rules) A function is a rule which specifies what to do with a given number of objects.

This paradigm, in contrast to (Functions-as-Graphs), emphasizes the process of calculating: You do something with some objects and possibly receive some object. To stay with the example of the successor function, an advocate of (Functions-as-Rules) would define it this way:

(σ_R) $\sigma =$ that rule which needs one argument x and returns $x + 1$

The same way that (Functions-as-Graphs) is the view of functions from the perspective of set theory, the idea of (Functions-as-Rules) is the foundation for the systems of λ-calculus, and while there is a plethora of philosophical introductions to set theory, to my knowledge, there is no such introduction to the λ-calculus.

The aim of this thesis is, thus, twofold: On the one hand, I intend to close the just mentioned gap by providing explanations to the most important concepts of λ-calculus, with a particular emphasis on the informal ideas behind the formal machinery, which, to my concern, are not covered in the common introductions to λ-calculus.

On the other hand, my aim is to argue that the question of whether functions are rules or graphs is not just of ontological interest, but also pivotal from a *formal* point of view: Some

graphs cannot be defined by a rule, and some rules have no corresponding graph.

The structure of this thesis is straight-forward: First of all, the prerequisites to understanding my position are covered. Chapter 2 deals with different accounts of what functions are, prominently including Frege's view, (Functions-as-Graphs) and (Functions-as-Rules). Secondly, Chapter 3 gives the just mentioned introduction to the formal machinery behind (Functions-as-Rules), the λ-calculus, with special emphasis on the intuitive ideas behind it. Then, in Chapter 4, I defend my main claim that even from a formal point of view, one cannot stay indifferent to the question whether functions are graphs or rules. Finally, Chapter 5 gives a brief summary of the thesis, discusses open ends and points to further areas of investigation.

Before we dive in, two final remarks are in order. First of all, the reader of this thesis is assumed to be trained in philosophical logic, set theory and the philosophy of language; a reader not proficient in these areas will have a very hard time advancing from this point onward. Secondly, for reasons of legibility, we adopt a quoting convention. First, quotation marks around formulas are left out unless the property of being an expression is necessary in the context at hand. For example, we say that $\forall \mathrm{x}\, \mathrm{Fx}$ is equivalent to $\neg\exists \mathrm{y} \neg \mathrm{Fy}$, but that "Fx" is composed of two letters. Secondly, schema letters are used instead of corner quotes and metavariables. To give an example, you will read $\lambda x.\, x$ or "$\lambda x.\, x$" instead of $\ulcorner \lambda \chi.\, \chi \urcorner$.

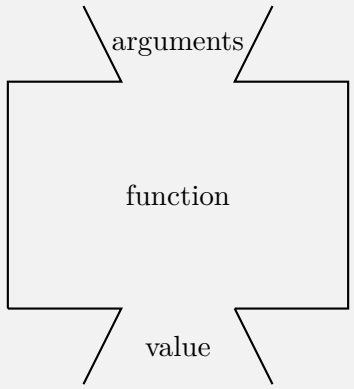
2 The Concept of Function

Before we start with the formal machinery of λ-calculus, let us have a brief look at the most basic question regarding functions: *What is a function, really?* As with almost every even remotely philosophical question, there is a myriad of different answers, and we shall not take a stance on this issue. Instead, this chapter gives an overview of the most important positions. Section 2.1 starts with a brief summary of Frege's view of functions. Then, Section 2.2 expands on the paradigm of (Functions-as-Graphs), its set-theoretic implementation and its connection to Frege's *Wertverläufe*. Lastly, Section 2.3 explains the intuitive idea behind (Functions-as-Rules) in an informal setting, paving the way to Chapter 3, in which the formal machinery of the λ-calculus is treated.

2.1 Frege's View of Functions

Let us start with Frege's view of functions first, since it has been extraordinarily influential in set theory and logic. Frege's ontology is divided into *functions* and *objects*. Whereas an object is complete in its own right, a function is some kind of *unsaturated* entity; it needs objects to saturate it, the *arguments*, and if it is saturated, it yields another object, the *value*, as Figure 2.1 illustrates.

Figure 2.1: Input-Output-Behavior of Functions



Objects are denoted by what Frege calls *proper names*. We speak of *singular terms* nowadays, but the expressions do not mean the same since Frege prominently includes sentences as proper names, claiming that their denotations are truth-values. Functions are denoted by *function signs*, which are just as unsaturated as the functions they denote (Frege, 1904, p. 665). The same way a function in combination with some objects yields yet another unique object, a function sign in combination with some proper names yields yet another proper name.

The *arity* of a function is the number of objects needed to complete it; *mutatis mutandis* for function signs. Lastly, one can create an n-ary function sign by replacing n proper names in some other proper name by a placeholder. Frege suggests using empty parentheses to mark unsaturatedness and avoid confusing them with variables (p. 664). The table below gives a brief overview of Frege's theory of functions.

Vocabulary	**Explanation**	**Ex. in English**	**Ex. in Logic**
$\text{Eigenname}_\text{F}$	singular term, sentences included, denotes a Gegenstand	"Frege", "the philosopher"	"a", "$\iota x\, Px$"
Gegenstand	saturated entity, too basic to be defined (Frege, 1891, p. 18)	Frege, Aristotle	Frege, Aristotle
Funktionszeichen	denotes a function	"the father of ( )"	"f", "+", "$[\![...]\!]$"
Funktion	denotation of a Funktionszeichen	being-the-father-of-( )	–
Wertverlauf	object (!) with information about a function's arguments and values	$\dot{\varepsilon}(\varepsilon - 4)$	$\{\langle 0, -4\rangle, \langle 1, -3\rangle, ...\}$
Begriffswort	function sign denoting a Begriff_F; primarily predicates	"x is young and smart"	"F"
Begriff_F	function mapping objects to truth-values; sometimes called "Eigenschaft"	( )-being-young-and-smart	–
Begriffsumfang	Wertverlauf of a Begriff_F, set theory: characteristic function	$\dot{\varepsilon}(B\varepsilon)$	$\{\langle \text{Joe}, 0\rangle, \langle \text{Bob}, 1\rangle, ...\}$

There is a lot more to say about Fregean functions and their problems. For example, Frege does not allow partial, co-partial or multi-valued functions, and it is unclear what the denotation of a function sign is exactly (cf. "–" in the table), never mind its sense. While these issues are surely relevant and interesting, they are out of the scope of this thesis.

In the following, we will stay in line with the historical development of the concept of a function and keep the input-output idea but discard the idea of saturatedness along with its metaphysical baggage. Both (Functions-as-Graphs) and (Functions-as-Rules) claim that functions are *objects*; graphs are sets and sets are objects, and rules are denoted by singular terms like "that rule which needs one input and returns it", so they are objects as well. Instead of gaps in expressions like "the father of ( )", we will simply make use of variables, as in "the father of x". Clearly, "x" is a term, and while the denotation of "the father of x" depends on the valuation of "x", in contrast to "the father of ( )", it always denotes an object.

2.2 Functions as Graphs

Next up, then, let us have a look at the most well-known and established view of functions:

(Functions-as-Graphs) A function is a set of argument-value pairs such that every argument has a unique value.

To explain the essence of this view, we consult Frege again. Next to his actual unsaturated functions, which we will not consider anymore, he also proposes corresponding *objects* that contain information about arguments and values. These objects, which he calls *Wertverläufe*, are characterized by his infamous Basic Law V, whose natural language counterpart he mentions in *Funktion und Begriff* (1891, p. 10):

> Daß es nun möglich ist, die Allgemeinheit einer Gleichung zwischen Funktionswerten als eine Gleichung aufzufassen, nämlich als eine Gleichung zwischen Wertverläufen, ist, wie mir scheint, nicht zu beweisen, sondern muß als logisches Grundgesetz angesehen werden.

To put it in simple terms, the Wertverläufe of two functions are identical if and only if the functions' values are the same for every argument. In set theory, this idea of a function's Wertverlauf was implemented by objects known as *graphs*. Graphs are just sets of argument-value pairs, so they are the closest set-theoretic relatives to Wertverläufe. Analogous to Basic Law V, they are characterized by a property that is commonly referred to as *extensionality*:

(Extensionality of Graphs) $\forall x\, f(x) = g(x) \leftrightarrow \{\langle x, y\rangle \mid y = f(x)\} = \{\langle x, y\rangle \mid y = g(x)\}$

Note that this usage of "extensionality" is not co-extensional with the usage in metaphysics and modal logic; see Alama and Korbmacher, 2021, pp. 5–8. In set theory, as opposed to Frege's *Begriffsschrift*, (Extensionality of Graphs) does not lead to inconsistency. But set theory goes one step further than just reconstructing Fregean Wertverläufe – it *identifies* them with functions:

⚙ Definition 2.1: Function in Set Theory

A *function* is a left-total and right-unique relation.

Since relations are subsets of cartesian products and cartesian products are sets of tuples, functions are defined as left-total and right-unique sets of tuples – i.e., graphs. This makes functions objects and discards the idea of unsaturated entities. Note that left-totality rules out partial functions, right-uniqueness multi-valued functions. This definition allows one to shorten

(Extensionality of Graphs) by identifying graphs with functions, yielding the principle of extensionality the way we know it:

(Extensionality of Functions) $\forall x\, f(x) = g(x) \leftrightarrow f = g$

Sometimes, only the left-right-implication is mentioned since the right-left implication is easily derivable. Frege would be turning in his grave seeing (Extensionality of Functions) and claim that it was syntactical nonsense because "f" and "g" are used as singular terms on the right side of the biconditional. But since functions are objects in set theory and "f" and "g" are names for such objects, they are, in fact, singular terms.

2.3 Functions as Rules

In contrast to (Functions-as-Graphs), there is another way of looking at functions:

(Functions-as-Rules) A function is a rule which specifies what to do with a given number of objects.

The prime idea of (Functions-as-Rules) is that functions stand in close relation with *operations*. For example, $2+3$ tells you to add 3 to 2, and in general, $x+y$ tells you to add y to x. This way of thinking about functions is quite intuitive, at least when looking back at the way functions were introduced at school: You have some given objects, the *inputs*, and perform the operations the function specifies, usually (but not always, as we will see) resulting in some *output*.

A critical reader might frown at this way of specifying functions and feel uneasy about (Functions-as-Rules): If functions are rules, what is a rule, then? Using a non-primitive word in the definiens of a definition is not exactly eye-opening, after all. In the following, we will assume that the same way the question of what a graph is is sufficiently answered by the axioms of set theory, the question of what a rule is is sufficiently answered by the axioms of λ-calculus. If one still feels uneasy about rules, one should feel the same about graphs, since the words "graph" and "rule" the way we use them are defined by formal systems in the same vein. For a discussion of this issue, see also Section 5.2.

Apart from this rather formal answer, rules have some distinct characteristics that are worth pointing out:

Definition 2.2: Characteristics of Rules

(1) A rule can be *described*.

(2) A rule specifies operations on a given number of *objects*.

(3) Given a rule and some objects, it can be *followed*.

First, rules can be described using definite descriptions, as, for instance, "that rule which needs one object as input and whose output is the result of adding 1". We can give rules names, too, as for example "σ", but as we will see in Chapter 3, this is usually not done.

As a an important note, (Functions-as-Rules) discards the idea of functions as unsaturated entities the same way (Functions-as-Graphs) does, and point (1) makes this clear: Rules can be described via definite descriptions, which are singular terms, so their denotations must be saturated objects. Here, we do not have any gaps either: Rules are complete the way they are, they simply need objects for them to be followed.

Secondly, rules always require a specific number of objects to operate on. This number is specified in a rule description, as is done with the phrase "which needs one object" for σ: We cannot add a number *per se*, and we cannot, strictly speaking, add a number to *several* numbers at once – we can only add a number to *exactly one* number.

Lastly, one can *follow* a rule by performing the operations it specifies on some given objects. For example, given σ and the number 5, we can follow it and receive 6 as a result. Note that when following a rule, one does *not* necessarily receive some result. The reason for this is that some rules describe genuine Sisyphean tasks, and we will encounter some of them with (ΩΩ) and nobase.

In the next chapter, we will come back to the three characteristics described in Definition 2.2 to see that they correspond to the formal notions of λ-abstraction, λ-application, and a sequence of β-reductions in an attempt to reach β-normal form.

3 Introduction to the λ-Calculus

So far, we have dealt with the ideas behind (Functions-as-Rules) and (Functions-as-Graphs), and while the reader is assumed to be familiar with the set theory behind (Functions-as-Graphs), the theory behind (Functions-as-Rules) is yet to be covered. The aim of this chapter is to do so, which mostly fixes its structure. First of all, Section 3.1 explains the most important ideas behind the λ-calculus from an intuitive point of view to pave the way to the formal sections: Section 3.2 is concerned with the syntax of the λ-calculus. It explains how to *construct* λ-terms and λ-equations, the thoughts behind the definitions, and some useful concepts related to them. In contrast, Section 3.3 is about how to *manipulate* existing λ-terms: It examines the proof theory behind the λ-calculus by explaining its axioms and proof rules.

Before we dive in, though, we briefly touch on some historical remarks. λ-terms first occurred in print in 1932/1933 when Alonzo Church tried to formulate an alternative to set theory suited as a foundation for mathematics. His students Kleene and Rosser quickly proved it to be inconsistent, though (see Kleene and Rosser, 1935; Curry, 1942), and it was revised into the system we now know as the *pure λ-calculus*, which we are covering in this thesis. Note that this system does not allow constants either, which is important for the interpretation: In general, λ-terms denote rules, so that, in particular, you abstract from rules and you apply rules to rules. For a discussion, see Section 5.2.

Concerning the notation of the λ-calculus, there are several hypotheses why Church uses the sign "λ", the most prominent being that of his student J. Barkley Rosser from 1984, p. 338:

> Church was struck with certain similarities between his new concept and that used in Whitehead and Russell (1925) for the class of all x's such that $f(\mathrm{x})$; to wit, $\hat{\mathrm{x}}\, f(\mathrm{x})$. Because the new concept differed quite appreciably from class membership, Church moved the caret from over the x down to the line just to the left of the x; specifically, $\wedge\, \mathrm{x} f(\mathrm{x})$. Later, for reasons of typography, an appendage was added to the caret to produce a lambda; the result was $\lambda \mathrm{x}\, f(\mathrm{x})$.

For further information on the history of the λ-calculus, the reader is referred to Seldin, 2009 as well as Cardone and Hindley, 2009.

3.1 The Ideas behind λ-Calculus

Before we delve into the formal details, this section covers the most important ideas behind λ-calculus: abstraction, application and β-reduction. In the more formal sections to come, we will discover their formal counterparts.

3.1.1 Abstraction

As we all know, an n-ary function sign in combination with n singular terms again yields a singular term. In reverse, then, a singular term yields an n-ary function sign when removing n of the singular terms it contains. Usually, we give names to those function signs, but how do we *describe* them?

First of all, we can create an open term by substituting a variable for at least one singular term inside a closed term. For instance, consider the definite description "the mother of Catherine", which is a complex singular term. It contains another singular term, "Catherine", so by substituting that term by a variable, say x, we get "the mother of x". Note that this expression still does not denote a function, but the *value* of that function which needs one object and returns its mother for the valuation of x; here, "x" still has a denotation, it is just relative to a valuation.

In order to actually denote the function itself, we need to get rid of this denotation and make clear that x is a parameter representing the yet unknown object to be passed to that function. In order to do so, we prepend the open term with the letter "λ" followed by the variable we used to replace some of its singular terms as well as a dot, and wrap the resulting expression in parentheses. The variable then becomes *bound* by the abstraction operator. To stay with the example, we receive "(λx. the mother of x)", where x is bound. We can now rightfully read this as "that function which needs one input and returns its mother".

In the following, we will call such constructs *descriptive function signs*. The reason for this is that they denote functions, so they are function signs, but in contrast to usual function signs which are proper names for functions, we here *describe* functions without the need for naming them. Note that, in contrast to Frege, the descriptive function sign we just created is a *saturated* one, since it is a definite description and, thus, denotes an object.

The process we have just had a look at is commonly called *functional abstraction*. Its name hits the bull's eye, since "abstraction" comes from Latin "abstrahere", which means "to drag away", and in a certain sense, this is just what we do: We drag away the meaning of a singular term within another singular term. This way of dealing with variables to abstract from their meaning is common in logic as well; for example, when a variable is bound by a quantifier, the same way, its valuation does not matter for the formula's meaning.

As a limiting case, note that "contains" is used reflexively here: Proper names can also be abstracted. To give an example, "Frege" can be abstracted to (λx. x), a descriptive function sign which can be read as "that function which needs one input and returns it", and which obviously denotes the identity function.

To stay within the paradigm of (Functions-as-Rules), since functions are rules and we have found a way to describe functions, we have found a way to *describe rules* as well; descriptive

function signs contain information about how to follow a rule. This corresponds to point (1) of Definition 2.2.

3.1.2 Application

So far, we have found a way to describe functions using abstraction. The purpose of functions is to *apply* them to some objects, though, so we need a way to describe this in our language as well. To give an example, we can apply "(λx. the mother of x)" to "Catherine" again. In the notation we know, if "m" was the function sign denoting the function which returns its input's mother and "y" was the variable denoting Catherine, we would write "m(y)".

The notation we will now use, though, is a bit different: We simply append the input to the descriptive function sign and, again, wrap it in parentheses. In our example, we would get "((λx. the mother of x)y)", which we would read as "that function which takes one argument and returns its mother, applied to Catherine". We call the left part of the term the *applicans* and the right one the *applicandum*. Functional application corresponds to point (2) of Definition 2.2.

3.1.3 β-reduction

Now we have a description of a function being applied, but we are missing something: Abstraction and application should be inverses for the same singular term: If you have a singular term, abstract some singular term from it and apply that very same term to the resulting descriptive function sign again, you should end up with the same singular term you began with. To stick with our example, "that function which takes one argument and returns its mother, applied to Catherine" should again result in "the mother of Catherine". What we should do, then, is to allow such lengthy application terms to *reduce* to less lengthy expressions. Staying with our notation, the same way we *introduce* λs using abstraction, we should be able to *eliminate* them again after the descriptive function sign was applied to some singular term.

This can be done by reverse-engineering functional abstraction. Here, we replace one or more singular terms by a variable and prepend it with an occurrence of "λ" to the term we abstract. Eliminating an abstraction, then, means removing the occurrence of "λ" followed by the variable and replacing all of its occurrences formerly bound by "λ" by the applicandum again. To illustrate, ((λx. the mother of x)y) then reduces to just "the mother of y". The rule which makes this possible is called β-*reduction*, and is part of *following a rule* – the third property mentioned in Definition 2.2.

Note that there actually is a somewhat similar concept of inverse rules in logic, namely that of universal and existential instantiation. The elimination rules also allow us to omit the abstraction operator and replace all occurrences of the now free variable. There are two big differences,

though. First, universal and existential instantiation allow to replace a variable by a *variable*, while β-reduction allows to replace a variable by an *applicandum*, which is a λ-term; it may, but does not need to be a variable. Second, while UI end EI have inverses on the level of *proof theory* – UG and EG –, β-reduction has no inverse on the level of proof theory since functional abstraction, as we will see, is a *syntax rule*.

3.2 Syntax of the λ-calculus

Now that we have an intuitive idea of the three most important notions in λ-calculus, we can start building up a formal system. To do so, we first of all need a syntax, which we cover in this section.

3.2.1 Constructing λ-Terms

Alphabet

The alphabet of the λ-calculus is quite minimalistic; it only needs seven symbols in total.

Definition 3.1: The Vocabulary of the Lambda Calculus

variable	x
hook-sign	′
abstractor	λ
parentheses	), (
dot	.
identity-sign	=

The first five symbols are needed to construct λ-terms. The dot is, strictly speaking, not needed, but added for reasons of legibility. The identity symbol is used to create equational formulas.

Variables

Variables are defined recursively the well-known way:

Definition 3.2: Variables of the Lambda Calculus

(1) "x" is a variable.

(2) If "x" is a variable, so is "x'".

(3) Nothing else is a variable.

The set of all variables is commonly referred to by "V". Note the difference in markup between the occurrence of "x" in clause (1) and the ones in clause (2). We agree on abbreviating "x′" by "y", "x″" by "z" etc.

Lambda Terms

Now we have the means to define the set of λ-terms, denoted by "Λ", which is, of course, also defined recursively:

Definition 3.3: Lambda Terms

(1) Every variable is a λ-term.

(2) If "x" is a variable and "M" is a λ-term, so is "$(\lambda x.\,\mathrm{M})$".

(3) If both "M" and "N" are λ-terms, so is "(MN)".

(4) Nothing else is a λ-term.

Clause (1) makes variables the most basic building blocks of λ-terms. Clause (2) defines the syntax of *functional abstraction*, whose basic idea we have covered in Section 3.1.1. It is the formal counterpart to point (1) in Definition 2.2. Clause (3) is the formalization of *functional application* which we covered informally in Section 3.1.2. It corresponds to point (2) in Definition 2.2. We read "$\lambda x.\,\mathrm{M}$" as "that function which takes one input x and whose output is M", "(MN)" as "M applied to N", where we call M the *applicans* and N the *applicandum*. Figure 3.1 gives a simple example of every rule in Gentzen-style notation.

Figure 3.1: The most Basic Ways of Forming λ-Terms

$$\frac{}{\mathrm{x}}\,(1) \qquad (2)\ \frac{\dfrac{}{\mathrm{x}}\,(1)}{(\lambda \mathrm{x}.\,\mathrm{x})} \qquad (3)\ \frac{\dfrac{}{\mathrm{x}}\,(1) \qquad \dfrac{}{\mathrm{y}}\,(1)}{(\mathrm{xy})}$$

Note that functional abstraction is a permissible rule for *any* variable and λ-term, so one can abstract a variable's meaning although it does not occur in a λ-term; for example, $\lambda \mathrm{y}.\,\mathrm{x}$ also is a λ-term. Other examples of λ-terms are $(\mathrm{x}(\lambda \mathrm{x}.\,\mathrm{x}))$, $((\mathrm{xy})\mathrm{z})$, $(\lambda \mathrm{x}.(\lambda \mathrm{x}.\,\mathrm{x}))$, and $(\lambda \mathrm{x}.\,\mathrm{x}(\lambda \mathrm{x}.\,\mathrm{x}))$.

3.2.2 Bracket Conventions

As Definition 3.3 makes clear, there is a great number of parentheses involved in complex λ-terms. They are most precious when disambiguating two λ-terms, as with $(\lambda x.(xy))$ and $((\lambda x.\, x)y)$, but they also make them hard to read. For this reason, we adopt the following four conventions to make λ-terms more legible:

Definition 3.4: Bracket Conventions

(1) Outermost parentheses may be omitted.

(2) Application is left-associative: $(M\,N\,L) \stackrel{\text{def}}{=} ((M\,N)\,L)$.

(3) Abstraction is right-associative: $(\lambda x.\lambda y.\, M) \stackrel{\text{def}}{=} (\lambda x.(\lambda y.\, M))$.

(4) Application binds stronger than abstraction: $(\lambda x.\, M\,N) \stackrel{\text{def}}{=} (\lambda x.\, (M\,N))$.

Rule (1) is analogous to the one we adopt for well-formed formulas in logic: The same way we abbreviate "$(\alpha \wedge \beta)$" by "$\alpha \wedge \beta$", we abbreviate "$(\lambda x.\, M)$" by "$\lambda x.\, M$" and "$(M\,N)$" by "$M\,N$". Rules (2) and (3) ensure that we can read λ-terms in the usual left-to-right fashion. One might wonder why this goal is reached by making abstraction *right*-associative. The reason is that one constructs a λ-term whose inputs are $x_1, ..., x_n$ by abstracting the arguments in *reverse* order. For example, consider $f(x, y, z) = x^2 + y \times z$: It takes three inputs: x, y and z. The corresponding λ-term is $(\lambda x.(\lambda y.(\lambda z.\, x^2 + y \times z)))$, which we will understand in Section 3.3.5. By defining abstraction to be right-associative, we can leave out the brackets nested from right to left and retain the reading from left to right. In our example, we get $(\lambda x.\lambda y.\lambda z.\, x^2 + y \times z)$, which is much more legible.

Rule (4) makes sure that abstraction by default has the widest scope. This is very natural, since when describing a function by means of a descriptive function sign, its input parameters should range over the whole term. For example, $\lambda x.\lambda y.\lambda z.\, xyz$ should be read as $(\lambda x.(\lambda y.(\lambda z.\, ((xy)z))))$ instead of $((\lambda x.(\lambda y.(\lambda z.\, x)y))z)$.

3.2.3 Free and Bound Variables

As we saw in Section 3.1.1, functional abstraction makes the abstracted variable *bound* while all variables that are not abstracted are *free*. In order to have a precise notion of bound and free variables, we use an inductive definition:

Definition 3.5: Free and Bound Variables of λ-Terms

$\boxed{\mathrm{FV} : \Lambda \to \mathbb{P}(\mathrm{V})}$

(1) $\mathrm{FV}(x) = \{x\}$

(2) $\mathrm{FV}((\lambda x.\, \mathrm{M})) = \mathrm{FV}(\mathrm{M}) \setminus \{x\}$

(3) $\mathrm{FV}((\mathrm{M}\,\mathrm{N})) = \mathrm{FV}(\mathrm{M}) \cup \mathrm{FV}(\mathrm{N})$

$\boxed{\mathrm{BV} : \Lambda \to \mathbb{P}(\mathrm{V})}$

(1') $\mathrm{BV}(x) = \emptyset$

(2') $\mathrm{BV}((\lambda x.\, \mathrm{M})) = \mathrm{BV}(\mathrm{M}) \cup \{x\}$

(3') $\mathrm{BV}((\mathrm{M}\,\mathrm{N})) = \mathrm{BV}(\mathrm{M}) \cup \mathrm{BV}(\mathrm{N})$

To put it in simple terms: The bound occurrences of a variable in a λ-term are those which are in the scope of a λ-abstraction of that variable. The free occurrences of that variable are those that are not in said scope.

Note that the set of free and the set of bound variables need to be defined separately, since it might happen that in the same λ-term, one occurrence of a variable is bound and another occurrence of the same variable is free. As an example, consider the λ-term $\mathrm{x}(\lambda \mathrm{x}.\lambda \mathrm{y}.\, \mathrm{x})$: The first occurrence of "x" is free, but the second is bound. In particular, we have

(1) $\mathrm{BV}(\mathrm{x}(\lambda \mathrm{x} \lambda \mathrm{y}\, \mathrm{x})) = \{\mathrm{x}, \mathrm{y}\}$

(2) $\mathrm{FV}(\mathrm{x}(\lambda \mathrm{x} \lambda \mathrm{y}\, \mathrm{x})) = \{\mathrm{x}\}$

Note, lastly, that if a λ-term M does not have free occurrences (i.e., $\mathrm{FV}(\mathrm{M}) = \emptyset$), M is said to be *closed*; closed terms are sometimes called *combinators* and the class of all such terms is Λ_0. There is a whole theory about this called *combinatory logic*, but we will not consider it here; the reader is referred to Curry and Feys, 1958 instead.

3.2.4 Head, Body, and the Notation $\vec{x}$

As should be clear by now, if the last rule to construct a λ-term was functional abstraction, we are dealing with a λ-term *describing* a function. If a λ-term describes a function with n arguments, functional abstraction was used n times at the end of constructing the λ-term. For example, $\lambda \mathrm{x}.\lambda \mathrm{y}.\, \mathrm{y}$ can be understood as a descriptive function sign of a two-argument function which returns the second argument (more on that in Section 3.3.5). It was constructed by taking y and using functional abstraction twice – first for y, then for x. This knowledge allows us to regard an abstraction term as having two parts: the abstraction part, which lists the argument(s) of the function, and the part which actually says what do with them. The former is called the λ-term's *head*, the latter the λ-term's *body*. We can define this formally:

Definition 3.6: The Notation $\vec{x}$

Let M be an arbitrary λ-term. We define:

(1) "$\vec{x}$" abbreviates "$x_1, ..., x_n$".

(2) "$\lambda\vec{x}.\,M$" abbreviates "$\lambda x_1 ... \lambda x_n.\,M$".

(3) Given "$\lambda\vec{x}.\,M$", "$\lambda\vec{x}$" is its *head*, "M" its *body*.

3.2.5 λ-Equations

Now we know what λ-*terms* are, but the interest of λ-calculus lies in *formulas* containing such λ-terms. Thus, we need a formal definition of a formula in the λ-calculus. In predicate logic, there are two types of formulas: *atomic* formulas, which are the most basic formulas composed of singular terms and relation symbols, and *complex* formulas, which are formulas built up from atomic formulas using quantifiers and connectives.

In general, it is very much possible to form such kinds of statements in the λ-calculus, and when Church first proposed his version of λ-calculus in 1932/1933, he actually did so by introducing quantifiers, connectives, and even a primitive ι-operator. Such systems have an enormous expressive power, though, and it is no wonder that Church's students Kleene and Rosser proved his theory to be inconsistent. There are a few theories of closed λ-terms allowing to add such extra symbols called *illative combinatory theories* (cf. Curry, 1972, p. 163), but they are both so rare and so advanced that they are beyond the scope of this thesis. We are concerned with the *pure* λ-calculus, whose only formulas are equations:

Definition 3.7: λ-Equations

Let M and N be λ-terms. A λ-*equation* is any string of the Form "M = N".

The place well-formed formulas have in predicate logic is assumed by λ-equations in the λ-calculus. Examples of λ-equations are $\lambda x.\,x = \lambda y.\,y$ and $(\lambda x.\,y)x = y$.

3.3 The Theory λ

So far, we have defined how to *construct* λ-terms and λ-equations, but we do not have rules which allow us to *manipulate* them. In this chapter, we will explain and motivate the rules of the most common proof system, which is denoted by just "λ".

3.3.1 Substitution

The Problem of Variable Capture

Everyone familiar with first-order logic knows what substitution of free variables in a formula is: You replace every free occurrence of the variable to be substituted by the variable to substitute. This concept is used both for universal and existential instantiation. Also, one should be familiar with the fact that, as opposed to UG, EI does not allow one to freely choose the variable to substitute the one bound by the quantifier. In fact, for a formula of the form $\exists x\, A$, there are tight restrictions on what variable to choose for substitution:

(1) You may not substitute by a variable which occurs *free* in $\exists x\, A$ or some other formula in the proof.

(2) You may not substitute by a variable which occurs *bound* in A.

In many introductory textbooks, condition (1) is treated with intense scrutiny. The reason for this is that substituting by a variable which occurs free already is the most common mistake students of logic make when applying EI. It neglects the fact that when eliminating an existential quantifier, one needs to choose a term of whose bearer there is no information yet. However, this restriction is not of our concern.

What is of our concern, though, is the second condition, although it is often times not discussed in detail: If you substitute by a variable which already occurs bound in the formula to be instantiated, at least one of its occurrences will get bound as well, changing the formula's meaning. Here is a tableau example of violating condition (2):

$$\begin{array}{lcl} (1) & \exists x \exists y(Fx \vee Gy) & \\ & \Big| & \\ (2) & \exists y(Fy \vee Gy) & (1),\ (EI) \end{array}$$

In this case, x is substituted by y although y occurs bound in $\exists y(Fx \vee Gy)$, resulting in $\exists y(Fy \vee Gy)$. Although we have just eliminated the quantifier binding x, the substitution of x by y makes the formerly free variable bound again. This phenomenon is called *variable capture*, since a variable that should be free is bound in the process of substitution, and a lot of our definitions will have extra-clauses to avoid it.

Defining Substitution in the λ-Calculus

As in almost every formal system, we also need to know what substituting free occurrences of a variable means in the λ-calculus. We now address this need by providing a formal definition.

Before we actually start defining substitution, though, let us briefly compare the substitution processes involved in logic and λ-calculus. In logic, you need substitution when removing quantifiers, and we have looked at the example of the existential quantifier in the last section. Furthermore, substitution only involves variables: You substitute variables by variables.

In λ-calculus, you also use substitution in the process of removing an abstraction operator, and we have seen this in Section 3.1.3. There is one bigger difference, though: There are *two* purposes for which we need substitution in the λ-calculus, which we will get to know in the next two sections: One is for equating λ-terms which only differ by the names of their bound variables, which is done by (α), and one is for removing an occurrence of "λ", which is done by (β). While (α) only allows replacing variables by *variables*, (β) allows replacing variables by λ-*terms* (which might, but do not need to be, variables; see Definition 3.3).

Thus, our definition of substituting free variables needs to cover substituting variables by λ-terms in general, and include substituting by variables in particular. Moreover, it should avoid the problem of variable capture. Definition 3.8 provides a definition which does just that.

Definition 3.8: Substitution in the λ-Calculus

(1) $x[x := \mathrm{M}] \equiv \mathrm{M}$

(2) $y[x := \mathrm{M}] \equiv y$, if y is not the same as x

(3) $(\lambda x.\, \mathrm{M})[x := \mathrm{N}] \equiv \lambda x.\, \mathrm{M}$

(4) $(\lambda y.\, \mathrm{M})[x := \mathrm{N}] \equiv \lambda y.\, (\mathrm{M}[x := \mathrm{N}])$, if $\mathrm{x} \notin \mathrm{FV(M)}$ or $\mathrm{y} \notin \mathrm{FV(N)}$

(5) $(\lambda y.\, \mathrm{M})[x := \mathrm{N}] \equiv \lambda z.(\mathrm{M}[y := z])[x := \mathrm{N}]$, if $x \in \mathrm{FV(M)}$, $y \in \mathrm{FV(N)}$, and z a new variable

(6) $(\mathrm{LM})[x := \mathrm{N}] \equiv \mathrm{L}[x := \mathrm{N}]\mathrm{M}[x := \mathrm{N}]$

The definition is defined by induction on the construction of λ-terms: Rules (1) and (2) cover variables, rules (3), (4) and (5) abstraction and rule (6) application. We read "$\mathrm{M}[x := \mathrm{N}]$" as "the result of replacing all free occurrences of x in M by N".

Since substitution will haunt us for the rest of this thesis, let us have a look at the rules one by one. Rule (1) is very straight-forward: It states that if the λ-term at hand is a variable and we are to substitute that variable by a λ-term, we receive just that λ-term. Thus, for example, $\mathrm{x}[\mathrm{x} := \lambda \mathrm{y.\, y}] \equiv \lambda \mathrm{y.\, y}$.

Rule (2) governs what happens if we are to substitute every free occurrence of a variable in a λ-term which happens to be a variable different from the one to be substituted – the result

is the variable itself, and it is clear why: If there is *no* occurrence of a variable in a λ-term, substituting *every* free occurrence of it does not change anything. An example of rule (2) is $\mathrm{x}[\mathrm{y} := \lambda\mathrm{x}.\,\mathrm{x}] \equiv \mathrm{x}$. The condition that y is not the same as x is needed since if they were the same, rule (1) would hold.

Rule (3) tells us what to do if the variable to be substituted is the same variable that is bound by its last λ-abstraction: Just as with rule (2), the λ-term stays the same, since the λ-abstraction makes every occurrence of the variable to be substituted *bound*, but substitution only operates on *free* variables. As an example of rule (3), $(\lambda\mathrm{x}.\,\mathrm{xy})[\mathrm{x} := \lambda\mathrm{x}.\,\mathrm{x}]$ stays $\lambda\mathrm{x}.\,\mathrm{xy}$.

In contrast to rule (3), rule (4) covers the case that the variable to be substituted is *not* the same as the one that is bound by the last λ-abstraction: Here, the substitution shifts into the scope of the abstraction operator. For example, $(\lambda\mathrm{y}.\,\mathrm{x})[\mathrm{x} := \mathrm{z}] \equiv (\lambda\mathrm{y}.\,\mathrm{x}[\mathrm{x} := \mathrm{z}])$, which is, using rule (1), $\lambda\mathrm{y}.\,\mathrm{z}$.

For the first time in our definition, imposing a condition on the rule is necessary to avoid variable capture: x may not occur free in M, or y may not occur free in N. To see why the condition is needed, consider the term $(\lambda\mathrm{y}.\,\mathrm{x})[\mathrm{x} := \mathrm{yz}]$. Here, $\mathrm{M} = \mathrm{x}$ and $\mathrm{N} = \mathrm{yz}$, $x = \mathrm{x}$ and $y = \mathrm{y}$. Obviously, x occurs free in x and y occurs free in yz, so the condition is violated. If we now applied rule (4) anyway, we would get $(\lambda\mathrm{y}.\,\mathrm{x}[\mathrm{x} := \mathrm{yz}])$, which is $\lambda\mathrm{y}.\,\mathrm{yz}$ by rule (1). Here, y is captured.

Let us now see why one of the alternates of the condition suffices: First, If x did not occur free in M, we would have a case of vacuous substitution. For example, $(\lambda\mathrm{y}.\,\mathrm{z}')[\mathrm{x} := \mathrm{yz}] \equiv (\lambda\mathrm{y}.\,\mathrm{z}'[\mathrm{x} := \mathrm{yz}]) \equiv \lambda\mathrm{y}\,.\mathrm{z}'$, using rules (4) and (2). The λ-term does not change, so we do not capture a variable. Second, if y did not occur free in N, we would not have a problem either. Since "not occurring free" means occurring bound or not occurring at all, we need two examples. As to the case of occurring bound, consider $(\lambda\mathrm{y}.\,\mathrm{x})[\mathrm{x} := \lambda\mathrm{y}.\,\mathrm{yz}]$. We get $(\lambda\mathrm{y}.\,\mathrm{x}[\mathrm{x} := \lambda\mathrm{y}.\,\mathrm{yz}])$ by rule (4) and $\lambda\mathrm{y}.\lambda\mathrm{y}.\,\mathrm{yz}$ by rule (1). Here, although we have the peculiar case of vacuous abstraction, no variable is captured since y in yz is already bound by the inner occurrence of λ. Concerning the second case of y not occurring at all in N, consider the term $(\lambda\mathrm{y}.\,\mathrm{x})[\mathrm{x} := \mathrm{z}'\mathrm{z}]$. This becomes $(\lambda\mathrm{y}.\,\mathrm{x}[\mathrm{x} := \mathrm{z}'\mathrm{z}])$ by rule (4) and $\lambda\mathrm{y}.\,\mathrm{z}'\mathrm{z}$ by rule (1), not resulting in a variable capture either.

As we have just seen, rule (4) only covers substitution in abstraction terms if it is not the case that $x \in \mathrm{FV(M)}$ and $y \in \mathrm{FV(N)}$. But it might also happen that this *is* the case, and we have just discussed an example. This poses a problem, since we want to define substitution for λ-terms in general, without loss of generality – and without running into the problem of variable capture.

Luckily, there is a solution for such cases: We simply rename the variable bound by the λ-abstraction that would cause variable capture before substituting. This is what rule (5) formalizes. If we have an abstraction term of the form $\lambda y.\,\mathrm{M}[\mathrm{x} := \mathrm{N}]$ such that x occurs free in

M and y occurs free in N, we choose a new variable z which occurs neither in M nor in N, change λy to λz and replace every free occurrence of y in M by z. This way, we make sure that if we now substitute every free occurrence of x by N, no variable is captured. To stick with our example, $(\lambda y.\, x)[x := yz]$ now becomes $(\lambda z'.\, x)[x := yz]$ by rule (5), which becomes $\lambda z'.\, x[x := yz]$ by rule (4), resulting in $\lambda z'.\, yz$ using rule (1). Note that rule (5) is, strictly speaking, not needed, and we will see why we choose to keep it in Proof 3.4.

Lastly, rule (6) determines what do if the λ-term at hand was constructed using functional application last: We distribute the substitution, applying it to both parts of the application term. To give a step-by-step example, consider the case of $((\lambda x.\, x)\lambda y.\lambda z.\, x)[x := \lambda z.\, x]$. Here, $\lambda x.\, x$ is applied to $\lambda y.\lambda z.\, x$, so we distribute $[x := \lambda z.\, x]$ over these terms, resulting in $(\lambda x.\, x)[x := \lambda z.\, x](\lambda y.\lambda z.\, x)[x := \lambda z.\, x]$.

The result of $\lambda x.\, x[x := \lambda z.\, x]$ is, by rule (3), $\lambda x.\, x$ – simply because every occurrence of x is bound in $\lambda x.\, x$, so the substitution does not alter the λ-term. The substitution of $(\lambda y.\lambda z.\, x)[x := \lambda z.\, x]$ is pushed further down the λ-term to $\lambda y.\lambda z.\, x[x := \lambda z.\, x]$ by applying rule (4) twice. Now rule (1) can be applied, resulting in $\lambda y.\lambda z.\lambda z.\, x$. Now that we have applied the substitution to both terms of $((\lambda x.\, x)\lambda y.\lambda z.\, x)$, we apply the first to the second, resulting in $(\lambda x.\, x)(\lambda y.\lambda z.\lambda z.\, x)$.

3.3.2 λ-Theories

Now that we have a clear notion of substitution in the λ-calculus, we have all the means to define λ-theories. Definition 3.9 provides the necessary vocabulary:

Definition 3.9: λ-Proof and λ-Deductions

Let Γ be a set of λ-equations.

(1) A λ-*axiom* is a λ-equation stipulated to hold.

(2) A λ-*rule* from Γ to M is a rule which is stipulated to hold and allows one to write M below a sequence of λ-equations at hand if every member of Γ occurs at least once in it.

(3) A λ-*theory* $\mathcal{T}$ is a set of λ-axioms and λ-rules.

(4) A λ-*deduction* from Γ to M in $\mathcal{T}$, written $\Gamma \vdash_{\mathcal{T}} M$, is a sequence of λ-equations such that every λ-equation is…

  (a) an *axiom* of $\mathcal{T}$,

  (b) the result of applying a λ-*rule* of $\mathcal{T}$, or

  (c) a *premise*, i.e. a member of Γ.

(5) A λ-*proof* is a λ-deduction without premises, i.e. $\Gamma = \emptyset$.

(6) A λ-*theorem* is the last line of a λ-proof.

Since we are dealing with an axiom system, the definitions are very much akin to those found in Hilbert-Style deduction, with which the reader is assumed to be familiar. In the following, we will be concerned with only one theory, denoted simply by "λ" and summarized in Definition 3.10 below; the greek letters associated with the axioms originate in Curry and Feys, 1958, and are very common in the relevant literature.

Definition 3.10: Axioms of the Theory λ

(α)	$\{\}$	$\vdash_\lambda$	$\lambda x.\,\mathrm{M} = \lambda y.\,\mathrm{M}[x := y]$, if $y \notin \mathrm{FV}(\mathrm{M})$
(β)	$\{\}$	$\vdash_\lambda$	$(\lambda x.\,\mathrm{M})\mathrm{N} = \mathrm{M}[x := \mathrm{N}]$, if $\mathrm{BV}(\mathrm{M}) \cap \mathrm{FV}(\mathrm{N}) = \emptyset$
(ρ)	$\{\}$	$\vdash_\lambda$	$\mathrm{M} = \mathrm{M}$
(σ)	$\{\mathrm{M} = \mathrm{N}\}$	$\vdash_\lambda$	$\mathrm{N} = \mathrm{M}$
(τ)	$\{\mathrm{M} = \mathrm{N}, \mathrm{N} = \mathrm{L}\}$	$\vdash_\lambda$	$\mathrm{M} = \mathrm{L}$
(μ)	$\{\mathrm{M} = \mathrm{N}\}$	$\vdash_\lambda$	$\mathrm{ML} = \mathrm{NL}$
(ν)	$\{\mathrm{M} = \mathrm{N}\}$	$\vdash_\lambda$	$\mathrm{LM} = \mathrm{LN}$
(ξ)	$\{\mathrm{M} = \mathrm{N}\}$	$\vdash_\lambda$	$\lambda x.\,\mathrm{M} = \lambda x.\,\mathrm{N}$

Whereas (α), (β) and (ρ) are axioms, (σ), (τ), (μ), (ν) and (ξ) are proof rules. Since there is no notion of contradiction in λ-calculus, a theory $\mathcal{T}$ is considered *consistent* if not all λ-equations are provable. This is just the idea behind consistency proofs using the principle of explosion. The theory λ which we cover is consistent, which one can prove using the Church-Rosser theorem (see Section 3.3.4).

Before the axiomatic system of Definition 3.10 is explained in detail, a quick note on notation: Deductions and proofs – whether in predicate logic or λ-calculus – are displayed in table form, following Quine, 1969, p. 221. For deductions, dependency on assumption is marked using the star sign "⋆". For proofs in λ-calculus, the star column is left out. The reason for this is that there is no way to eliminate assumptions, so you will never find a starred λ-term in a λ-proof.

3.3.3 Dealing with Variable Names: (α)

Starting with the first axiom, let us make an observation: In the realms of logic, two formulas A and B are equivalent if they only differ by the names of their bound variables; for example,

∃x(Fx ∧ Gz) and ∃y(Fy ∧ Gz). The reason for this is that bound variables do not uniquely denote a member of the domain – their meaning is abstracted once they stand in the scope of a quantifier. In predicate logic, you can easily prove the equivalence of such formulas by using the instantiation rule immediately followed by the generalization rule for the respective quantifier:

Proof 3.1: Change of Bound Variables in PL

Stars		Line	Equation	Reference	Rule
⋆		1	∃x(Fx ∧ Gz)	–	Hyp
⋆		2	Fy ∧ Gz	1	EI in y
⋆		3	∃y(Fy ∧ Gz)	2	EG
		4	∃x(Fx ∧ Gz) → ∃y(Fy ∧ Gz)	1,3	I→
	⋆	5	∃y(Fy ∧ Gz)	–	Hyp
	⋆	6	Fx ∧ Gz	5	EI in x
	⋆	7	∃x(Fx ∧ Gz)	6	EG
		8	∃y(Fy ∧ Gz) → ∃x(Fx ∧ Gz)	5,7	I→
		9	∃x(Fx ∧ Gz) ↔ ∃y(Fy ∧ Gz)	4,8	$\text{Def}_{\leftrightarrow}$

Proof 3.1 is possible because instantiation and generalization are *inverse* rules: Applying instantiation first and generalization afterwards does not change a formula's meaning.

Analogous to formulas in predicate logic, it might also occur that two λ-terms only differ by the names of their bound variables. Since the λ-operator abstracts a variable's meaning in just the same way as ∀ or ∃, such λ-terms should mean the same as well. For example, "λx. x" and "λy. y" both denote that function which needs one argument and returns it. As opposed to predicate logic, though, there are no inverse rules for removing and adding λ-abstractions; as was explained in Section 3.1.3, we can only make sure that abstracting a singular term and then applying it again results in the same expression by means of β-reduction. Abstraction and application are on the level of *syntax*, though, while β-reduction is on the level of *deduction*, so we cannot prove that two λ-terms which only differ by the names of their variables do mean the same. For this reason, we explicitly introduce an axiom:

(α) $\{\} \vdash_{\lambda} \lambda x.\, \mathrm{M} \equiv \lambda y.\, \mathrm{M}[x := y]$, if $\mathrm{y} \notin \mathrm{FV(M)}$

The relation established by (α) is commonly referred to as α-*conversion*. It is possible to define an equivalence closure of α-convertibility, which is commonly called α-*congruence*. Since for our purposes, this is not needed, the reader is referred to Révész, 1988, p. 20.

Intuitively, (α) makes sure that if we replace all occurrences of a variable in a λ-term by occurrences of another (strictly speaking: possibly the same) variable, the resulting λ-term denotes the same function as the one to begin with. There is just one catch: The rule does not allow replacing a variable by another one which occurs free in the function body.

First of all, one might wonder why (α) allows changing the *bound* variables of a λ-abstraction, since variable capture only happens with *free* variables. The answer to this can be found in (β): The only way for a formerly bound variable to become free is by β-reduction (see Section 3.1.3), so applying (α) allows changing the variable that becomes free once (β) is used, fulfilling its purpose. We will get a better grasp in the following chapters, and especially in Proof 3.4.

The reason the condition is imposed on (α) is, once again, to avoid variable capture. To give an example, consider the λ-term "λx. y"; it denotes that function which takes an argument and regardless of what that is always returns the denotation of "y". If we now allowed replacing all occurrences of a variable in a λ-term by another one without respecting the above condition, we would be allowed to prove that $\lambda x.\, y = \lambda y.\, y$.

To illustrate why this is problematic, we can read the λ-terms as introduced in Section 3.1.1: That function which takes x as an argument and returns y is that function which takes y as an argument and returns y. In effect, we would say that a function which always returns the same value and the identity function are the same, which is obviously not the case. To avoid this problem of variable capture, we need the condition that $y \notin FV(M)$.

As we have just seen, (α) has two main purposes, namely

(1) equating λ-terms which only differ by the names of their bound variables,

(2) providing a way to avoid the problem of variable capture.

Another way to achieve these two things was made prominent by Hendrik P. Barendregt in 1981/1984, p. 26. To address problem (2), he introduces the so-called *Barendregt Convention*:

⚙ Definition 3.11: Barendregt Convention

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

Since variable capture only occurs if in the process of substituting, a free variable is accidentally bound, we can avoid the problem altogether by making sure that bound and free variables are always different. This also make clause (5) of Definition 3.8 obsolete.

Also, instead of using (α), Barendregt identifies α-convertible λ-terms on a syntactic level by postulating them to be identified, which satisfies (1) and allows dropping (α) altogether. This

makes meta-proofs about the λ-calculus shorter. For our purposes, though, we like to be as explicit as possible, and choose to keep both clause (5) of Definition 3.8 and (α).

3.3.4 Reducing λ-Terms: (β)

As we already introduced in Section 3.1.3, when given a rule and some objects, we need to be able to *follow* it. This process happens by using the objects provided with our rule. The next axiom on our list takes care of that:

$$(\beta) \qquad \{\} \mathrel{\vdash\!\!\!\!\!\!\lambda} (\lambda x.\, \mathrm{M})\mathrm{N} = \mathrm{M}[x := \mathrm{N}]$$

(β) allows us to reduce a descriptive function sign applied to a λ-term by removing its outermost occurrence of "λ", including the variable that it binds, to then replace every free occurrence of that variable by the applicandum. To stick with the example from Section 3.1.3, (β) is the axiom that allows ((λx. the mother of x)y) to reduce to "(the mother of x)[x := y]", which is just "the mother of y". In this sense, the argument is used. Although (β) only ever removes a single applicandum, one can use it to for functions which need more than one argument as well, as we will see in the next section.

To show how the axiom works, consider the term (λx.λy. x)y. (β) allows us to deduce that (λx.λy. x)y = (λy. x)[x := y], which is equivalent to λz. y, as one can see by applying the substitution rules (5), (1), (4) and (2) in that order (cf. Definition 3.8). Here, clause (5) allows us to rename the bound variable to avoid variable capture. If it was not for that clause, y would get captured and the equation would be false. For a proof without rule (5), see Proof 3.4.

If (β) allows us to write a formula of the form M = N, we say that M β-*reduces* to N, call M the *redex* (which is short for "reducible expression"), N the *contractum* and write $\mathrm{M} \to_\beta \mathrm{N}$; for instance, $(\lambda \mathrm{x}.\, \mathrm{xy})\mathrm{z} \to_\beta \mathrm{zy}$ where (λx. xy)z is the redex and zy the contractum, since (λx. xy)z = xy[x := z] ≡ zy by (β). In the same vein, we say that M *many-step*-β-*reduces* to N, written $\mathrm{M} \to_\beta^\star \mathrm{N}$, if there is a (possibly empty) sequence of β-reductions from M to N. To put it formally, many-step-β-reduction is the reflexive and transitive closure of β-reduction.

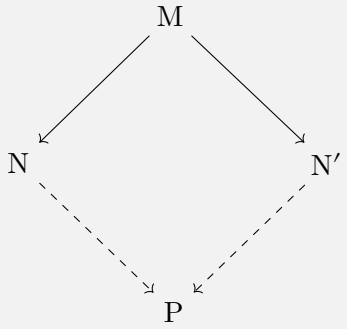
If (β) cannot be applied to a λ-term, we say that this term is in β-*normal form*, and can interpret it as the *result* of following all steps of a rule; see Section 3.1.3. Specifically, then, variables and descriptive function signs on their own are always in β-normal form, and application terms are only in β-normal form if the applicans is not a descriptive function sign. Looking back at Definition 2.2 and point (3) in particular, the process of β-reducing a λ-term to β-normal form corresponds to *following* a rule, and every β-reduction corresponds to one step in that process.

An important fact about β-normal forms is that *if* a λ-term has one, it is *unique*. This can be easily proved using the Church-Rosser theorem:

Definition 3.12: Church-Rosser Theorem

Let M, N, N′ and P be λ-terms.

If $M \to_\beta^\star N$ and $M \to_\beta^\star N'$, then there exists a λ-term P such that $N \to_\beta^\star P$ and $N' \to_\beta^\star P$.



To drive the point home, assume that a λ-term M has at least two β-normal forms N and N′. Then N and N′ both β-reduce to a common λ-term by the Church-Rosser theorem, so they are not in normal-form. This is a contradiction, so M does not have at least two β-normal forms. Thus, M has either exactly one or no β-normal form.

While having exactly one β-normal form is common, there are also λ-terms which have no β-normal form. We will encounter them with (ΩΩ) and nobase. This, in turn, means that there are some rules which one would need to follow ad infinitum, so they do not have an output, as we hinted at in Section 2.3.

Lastly, despite its name, β-reduction does not always need to *reduce* a λ-term in the sense that it gets smaller; nobase is an example of such a term. It even might happen that a λ-term gets larger in some β-reduction-step although it has a β-normal form, as recursive functions, which are covered in Section 4.1.1, show. However, it cannot happen that a redex one-step-β-reduces to a larger β-normal form.

3.3.5 Schönfinkelization

When looking at the syntax of the λ-calculus, one quickly realizes that *all* functions must be unary: On the one hand, functional abstraction only ever binds a single variable. On the other hand, β-reduction allows only for a single variable to be substituted. At first glance, this poses a problem, since there are rules which operate on two or more arguments.

All worries fade away, though, when realizing that it is possible to represent n-ary functions using unary functions only. The process of doing so is called *Schönfinkelization*, which is due to Schönfinkel, 1924, who was the first to do it this way; note, however, that Gottlob Frege had a less

general, but very similar way of Schönfinkelization long before 1924; see his *Doppelwertverlauf* in Frege, 1883, §36. Schönfinkelziation is – especially in the realms of computer science – also called *Currying*, since Haskell Curry made prominent use of it. Curry himself, though, notes that Schönfinkel discovered the idea about six years before him (Curry, 1980, p. 86).

To illustrate Schönfinkelization, let $f(x, y)$ be a binary function. We can define this as the unary function $\lambda x.\lambda y.f(x, y)$ – that function which, if x is given as an input, returns that function which, if y is given as an input, results in $f(x, y)$. To see how this works, we show that $(\lambda x.\lambda y.f(x, y)b)a = f(a, b)$. By (β), we get $(\lambda y.f(a, y))b$ and, by applying (β) again, $f(a, b)$, so in particular, $(\lambda x.\lambda y.f(x, y)b)a \twoheadrightarrow_\beta^\star f(a, b)$. Note that Schönfinkelization of an n-ary function produces $n - 1$ functions not necessarily relevant for their own sake, but necessary so as intermediate steps. These are called *partial applications*, because not all intended inputs have been applied yet. They play an important rule in computer science, but are not covered in this thesis.

Now that we know about Schönfinkelization, we have all the means to discuss the formal counterparts to Definition 2.2. Starting with the first characteristic, a rule can be described by a descriptive function sign whose formal counterpart is an abstraction term. Given a rule in need of n objects, the corresponding λ-term's head (see Section 3.2.4) was constructed using n instances of functional abstraction. This brings us to the second characteristic: Functional application is used to provide the objects the rule needs to be followed – one instance per object. Lastly, concerning the third characteristic, the process of *following* a rule is that of β-reducing the applicanda one by one, and the *result* of following the rule (if there is one, cf. (ΩΩ) and nobase) is a λ-term in β-normal form. While properties (1) and (2) find their implementation on the level of *syntax*, property (3) is an axiom on the level of *proof theory*.

3.3.6 β-Equivalence: (ρ), (σ), (τ)

As the name "λ-equation" for a string of the form "M = N" suggests, = is supposed to be an equivalence relation, i.e. a reflexive, symmetric and transitive relation. The following three axioms make sure this is the case:

(ρ) $\{\} \vdash_\lambda M = M$

(σ) $\{M = N\} \vdash_\lambda N = M$

(τ) $\{M = N, N = L\} \vdash_\lambda M = L$

"ρ", "σ" and "τ" are the greek letters corresponding to the initial letters of "reflexivity", "symmetry" and "transitivity", respectively. (ρ) actually follows from (β), (σ) and (τ), as the following

proof shows, so we do not need to add it:

Proof 3.2: Reflexivity

Line	Equation	Reference	Rule
1	$(\lambda x.\mathrm{M})\mathrm{N} = \mathrm{M}[x := \mathrm{N}]$	–	(β)
2	$\mathrm{M}[x := \mathrm{N}] = (\lambda x.\mathrm{M})\mathrm{N}$	1	(σ)
3	$\mathrm{M}[x := \mathrm{N}] = \mathrm{M}[x := \mathrm{N}]$	2,1	(τ)

Note that for any λ-term L, there exist λ-terms M and N as well as a variable x such that $\mathrm{L} \equiv \mathrm{M}[x := \mathrm{N}]$. Since we are not concerned with the brevity of meta-proofs, we leave (ρ) to be explicit about it.

3.3.7 β-Congruence: (μ), (ν), (ξ)

Now we made sure that $=$ is an equivalence relation, but there is more to it: If we know that M and N are equivalent rules, and if M occurs in another, more complex rule L, L should be the same rule when N replaces M in L, and we should be able to prove this. At the moment, though, this is not possible, so we introduce three new axioms which allow us to do so:

(μ) $\quad \{\mathrm{M} = \mathrm{N}\} \vdash_{\lambda} \mathrm{ML} = \mathrm{NL}$

(ν) $\quad \{\mathrm{M} = \mathrm{N}\} \vdash_{\lambda} \mathrm{LM} = \mathrm{LN}$

(ξ) $\quad \{\mathrm{M} = \mathrm{N}\} \vdash_{\lambda} \lambda x.\,\mathrm{M} = \lambda x.\,\mathrm{N}$

The idea behind (μ), (ν) and (ξ) is the following: There are only three rules for constructing λ-terms – variables, application and abstraction (see Definition 3.3). Since variables do not have subterms, the only way of constructing λ-terms which contain subterms is to use functional application or functional abstraction. Thus, in order to make sure that in *any* λ-term, one is allowed to replace a subterm by a β-equivalent λ-term, we need to introduce axioms which allow replacing subterms in application and abstraction terms.

Starting with functional application, there always is an applicans and an applicandum, so we need to introduce an axiom for each of them. (μ) allows the replacement of the applicans, (ν) that of the applicandum. The letter "μ" resembles the first letter of "monotonicity", "ν" is simply the one following "μ" in the greek alphabet. (μ) is sometimes called "left monotony", (ν) "right monotony". Monotone relations are a topic of their own, so the reader is referred to Curry and Feys, 1958, p. 59.

The only construction rule left is that of λ-abstraction. This is handled by (ξ), whose name follows "ν" in the greek alphabet. (ξ) states that if you have two equal λ-terms, you may use functional abstraction with the same variable on both of them and they stay equal.

These three axioms alow that for *any* λ-term, you can replace a subterm by a β-equivalent λ-term. This property is referred to as *compatibility*, and it plays an important role for the λ-calculus, since it makes $=$ a *congruence relation*, which is just a compatible equivalence relation.

While what (μ) and (ν) are there for is pretty straight-forward, the merits of (ξ) are not as obvious, so we are looking at them now. There are two important things to note. First, given a λ-term $\lambda\vec{x}.\,M$ and the equation that $M = N$, we can apply (ξ) repeatedly to receive $\lambda\vec{x}.\,N$, as the following example shows:

Proof 3.3: Weak Extensionality

Line	Equation	Reference	Rule
1	$(\lambda v.\, v)x = x$	–	(β)
2	$\lambda z.\, (\lambda v.\, v)x = \lambda z.\, x$	1	(ξ)
3	$\lambda y.\lambda z.\, (\lambda v.\, v)x = \lambda y.\lambda z.\, x$	2	(ξ)
4	$\lambda x.\lambda y.\lambda z.\, (\lambda v.\, v)x = \lambda x.\lambda y.\lambda z.\, x$	3	(ξ)

If we want to show that we are allowed to replace $(\lambda v.\, v)x$ by x in $\lambda x.\lambda y.\lambda z.\, (\lambda v.\, v)x$, we begin by stating that $(\lambda v.\, v)x = x$. Now (ξ) allows us to add the necessary λ-abstractions on both sides. Although axiomatically, we do not *actually* replace a redex by its contractum in a λ-term, the result of equating redex and contractum with (β) to then screw on the λs we need effectively results in the desired equation.

This application of (ξ) now allows us to prove that $(\lambda x.\lambda y.\, x)y = \lambda z.\, y$ without making use of clause (5) of Definition 3.8 (see Section 3.3.4):

Proof 3.4: Using Several Axioms at Once

Line	Equation	Reference	Rule
1	$\lambda y.\, x = \lambda z.\, x$	–	(α)
2	$\lambda x.\lambda y.\, x = \lambda x.\lambda z.\, x$	1	(ξ)
3	$(\lambda x.\lambda y.\, x)y = (\lambda x.\lambda z.\, x)y$	2	(μ)
4	$(\lambda x.\lambda z.\, x)y \equiv (\lambda z.\, x)[x := y] = \lambda z.\, y$	–	(β)
5	$(\lambda x.\lambda y.\, x)y = \lambda z.\, y$	3,4	(τ)

First, (α) is used to prevent variable capture, then (ξ) to screw on the necessary λ. (μ) identifies the *applicantes*, so that we can get the desired equation by means of (β) and (τ). Note that (α) is necessary despite of clause (5), but clause (5) is no necessary in light of (α); it is added to make proofs more concise, though, since with it, we can prove the equation above with β-reduction only (cf. Section 3.3.4).

As a second point, (ξ) plays an important role in terms of extensionality, since it is the only axiom which – as opposed to (ρ), (σ), and (τ), in a non-obvious way – allows equating descriptive function signs which denote, on the face of it, different rules. To illustrate, consider the term $\lambda y.(\lambda x.\, x)y$. It can be read as "that function which takes y as an argument and returns the result of applying it to the identity function". Since $(\lambda x.\, x)y \rightarrow_\beta y$ and since $\lambda y.(\lambda x.\, x)y$ is of the form $(\lambda x.\, M)$, $\lambda y.(\lambda x.\, x)y = \lambda y.\, y$ follows from (ξ). But "$\lambda y.\, y$" denotes that function which simply returns y. Obviously, the rules the two functions operate on are different: One rule just says "return the input", without performing any other operation on it. The other rule says "apply the input to that function which returns it, and return the value of that function".

While it is most obvious that the two rules always yield the exact same output, they are, from an intuitive point of view, not one and the same rule, since the operations and the number of steps you perform when following them are different. (ξ), thus, already imposes some form of extensionality on the λ-calculus, which is why it is commonly called the *axiom of weak extensionality*. Scott, 1973, p. 165 even goes as far as claiming that (ξ) *is* the principle of extensionality. Another reason for its name is that it is needed in combination with another axiom, (η), to incorporate the idea behind (Extensionality of Functions) into the λ-calculus. Extensional λ-calculi, however, are not covered in this thesis.

4 Functions as Graphs, or Functions as Rules?

Now that we are familiar with λ-calculus, and since the reader is assumed to be familiar with set theory, we know the formal theories behind both (Functions-as-Graphs) and (Functions-as-Rules). This allows us to discuss the differences between these positions both form an intuitive and a formal perspective. This is important, since on the face of it, one might be tempted to think that the discussion of whether functions are graphs or rules does not lead anywhere since formally, no matter how you think about them, you always end up with the same functions. Let us call this position an *indifferentist view of functions*, and its advocate a *functional indifferentist*. A functional indifferentist would claim that there is no relevant formal difference between (Functions-as-Rules) and (Functions-as-Graphs), which amounts to the conjunction of the following claims:

(Rules-as-Graphs) For every rule, one can construct a corresponding graph.

(Graphs-as-Rules) For every graph, one can construct a corresponding rule.

The aim of this chapter is to investigate such an indifferentist view of functions, and in particular the following problem: Both (Rules-as-Graphs) and (Graphs-as-Rules) claim connections between graphs and rules, but rules understood as instructions to be executed on objects face the restrictions of λ-calculus, while graphs are sets and thus restricted to the axioms of set theory. Do these axioms stand in the way of the functional indifferentist?

In the following, we will provide reasons why this question is to be answered in the affirmative for both claims. In Section 4.1, (Rules-as-Graphs) is examined and it is shown that the principle which backs it up from an intuitive point of view is both false and requires another principle to entail (Rules-as-Graphs), which is itself false. In Section 4.2, the consequences of (Graphs-as-Rules) are made transparent. In particular, it is pointed out that a functional indifferentist is forced to reject the axiom of choice, even in its weaker intuitionistic form. Section 4.3 discusses the results and brings them in connection to the typed λ-calculus. To avoid confusion, from now on, the word “function” is only used to stay agnostic about what functions are; to discern the objects the two notions of functions commit themselves to, we use the words “rule” and “graph”, respectively.

4.1 Rules as Graphs

On the face of it, it seems easy to show that (Rules-as-Graphs) holds, since there is an intuitively plausible principle in support of it:

(Rule-to-Graph) For any rule R and any inputs I, you can construct R's graph for I.

Note that "inputs" is chosen to be a *plural term* denoting not a single set, but several objects at once. This is necessary in order to not weave in the axioms of set-theory; see Section 4.1.2.

The idea behind (Rule-to-Graph) is simple: The output of a rule for an input object is simply the result of applying it to that object, so if you know the input and the rule, you can compute the output. And if you know both input and output, you can construct a corresponding input-output pair. Thus, if you know every input object, you can construct a set of all input-output pairs, which is just the graph you are looking for. For example, the rule $\lambda x.\, 2x$ yields the graph $\{\langle 2, 2 \times 2\rangle, \langle 8, 2 \times 8\rangle\}$ for the arguments 2 and 8.

The aim of this section is to unveil the problems with (Rule-to-Graph). On the one hand, we will show that it is plain false because it involves two universal quantifications – one over rules, one over inputs –, and both fail because of self-referring rules which do not yield an output for its input. On the other hand, we will show that even if (Rule-to-Graph) was true, it would not be enough to entail (Functions-as-Rules) on its own. It needs another assumption, (Rule-Inputs), in order to do so, which requires yet another assumption violating the axiom of foundation: (All Inputs).

4.1.1 The Problems with Quantifiers

Really *any* Rule?

First, then, let us have a look at the problems that arise with (Rule-to-Graph) quantifying over *all* rules. If it was true, there existed a graph even for every self-referring rule. A functional indifferentist might react to this worry by pointing out that self-reference *itself* is not problematic. In fact, there is a large set of self-referring rules which are harmless. As an example, consider the following factorial rule:

$$x! = \begin{cases} 1, & \text{if } x = 0 \\ x \times (x-1)! & \text{else} \end{cases}$$

Obviously, the rule refers to itself in the else-clause, so in the construction of the graph, the factorial rule would be mentioned as well. To stay with our example, we would have

$$! = \{\langle 0, 1\rangle, \langle 1, 1 \times 0!\rangle, \langle 2, 2 \times 1!\rangle\}$$

One might now fear that the axiom of foundation renders this impossible, since it prohibits a set from containing itself or an infinitely nested member, and the symbol "!" occurs both in the definiens and in the definiendum. When taking a closer look, though, we see that this fear is ungrounded. Although the sign "!" is used in the definition of !, ! *itself* – the set of ordered pairs – does not occur. Rather, the result of applying ! is mentioned in the construction of the graph. This does not cause a problem, though, since we can resolve this using our base case, ending up with a graph neither containing ! nor the symbol "!":

$$\begin{aligned} \{\langle 0, 1\rangle, \langle 1, 1 \times 0!\rangle, \langle 2, 2 \times 1!\rangle, ...\} &= \{\langle 0, 1\rangle, \langle 1, 1 \times 1\rangle, \langle 2, 2 \times 1 \times 1\rangle, ...\} \\ &= \{\langle 0, 1\rangle, \langle 1, 1\rangle, \langle 2, 2\rangle, ...\} \end{aligned}$$

As ! shows, a rule calling itself does not necessary pose a problem, and one can prove for every function commonly defined to be recursive that it is constructable in both set theory and λ-calculus; see Neumann, 1928 for set theory and Curry, 1972, p. 212 for λ-calculus.

While there are certainly *some* self-referring rules which do not cause problems, this does not mean that *all* self-referring rules are as harmless. For sake of clarity, let us call a rule which reoccurs in its own definition a *reoccursive* rule. So far, we have only considered a subset of the reoccursive rules, namely the *recursive* rules – those which can *run back* to a base case. What we have not considered yet are the *procursive* rules – those which can only *run forward*: no stopping, no turning back. Let us have a look at an example of such a procursive rule:

$$\text{nobase(x)} = \begin{cases} \text{nobase(x + 1)}, & \text{if x is even} \\ \text{nobase(x + 3)}, & \text{if x is odd} \end{cases}$$

Obviously, no matter what the input is, nobase will keep adding numbers, and there will never be a result. As an example, here is what happens if you apply nobase to 2:

$$\text{nobase}(2) = \text{nobase}(2 + 1) = \text{nobase}(3) = \text{nobase}(3 + 3) = \text{nobase}(6) = \text{nobase}(6 + 1) = ...$$

As should be clear, nobase keeps calling itself ad infinitum, so it cannot produce a value for 2 either, and the same goes for any number whatsoever. Of course, then, there is no graph for nobase, although it is a rule, so there is at least one rule and some inputs such that you *cannot* construct the rule's graph for the inputs – simply because some rules do not have an output for some inputs. This contradicts (Rule-to-Graph), which was the reason to believe (Rules-as-Graphs) in the first place.

Really *any* Input?

As a second point, let us discuss the problems with the universal quantification over inputs. Analogous to rules, there are problems with self-references. This time, though, it is not a rule referring to itself in its own definition, but a rule *applied to itself*. In λ-calculus, the rule which does just that is as simple as

(Ω) $\lambda x.\, xx$

While there certainly are inputs which yield an output for (Ω), as $\lambda x.\, x$ (see (II)), applying a self-applying rule to itself yields problems. To be specific, consider the term (ΩΩ): It β-reduces to itself ad infinitum, creating a never-ending loop:

(ΩΩ) $(\lambda x.\, xx)(\lambda x.\, xx) \rightarrow_\beta (\lambda x.\, xx)(\lambda x.\, xx) \rightarrow_\beta ...$

Analoguous to nobase, there is no value for ΩΩ, so one cannot construct an argument-value pair for every argument of Ω, and thus not a graph either. The only difference is that while (Ω) does not have a value for *some* inputs, nobase does not have a value for *any* input.

As a side comment, note that the case of (ΩΩ) makes clear that while (Functions-as-Rules) emphasizes the *procedural* character of functions, (Functions-as-Graphs) treats them as if they were *static*. While you *do* something with rules – applying them to arguments –, there is nothing to do with graphs; they have properties and they exist, and that's it.

A functional indifferentist might try to circumvent this problem by claiming that a rule should always be restricted to all of its sensible inputs. This way, one could claim (Ω) not to be a sensible input for itself, and rule out such loop-cases. As is shown in the next section, there is another reason which forces the functional indifferentist to endorse this claim, which will later be referred to as (All Inputs). However, no matter for what reason (All Inputs) is adopted, it gets the functional indifferentist caught up in set-theoretic antinomies.

4.1.2 The Implications of (Rule-to-Graph)

As we have just shown, self-referring rules provide overwhelming evidence that (Rule-to-Graph) is false, and while this suffices to render (Rules-as-Graphs) false as well, let us assume that the functional indifferentist was able to find a good argument in support of (Rule-to-Graph). Even under this assumption, they would still run into serious problems, though. The reason for this ist that (Rule-to-Graph) entails (Rules-as-Graphs) only under an important condition, namely

(Rule-Inputs) Every rule is associated with some inputs.

After all, if there was just a single rule of which we did not know the objects to which we can apply it, we would not know the first components of the input-output pairs, so we could not construct a corresponding graph for that rule, which is incompatible with (Rules-as-Graphs). Even if (Rule-to-Graph) held: As long as (Rule-Inputs) did not, the truth of (Rules-as-Graphs) would not follow.

The Necessity of (All Inputs)

The functional indifferentist would agree that (Rule-Inputs) is needed, but they would not see any problem and simply argue for it. In fact, they would point out that it is not sensible for every rule to be applied to any object whatsoever. After all, what is the result of multiplying stuff like water by 2? There is none, so one should not allow a rule to be applied to such cases.

Endorsing this view would explain (Rule-Inputs), but a functional indifferentist would then be forced to explain what is meant by "is associated with" in (Rule-Inputs). Does it mean that domains are parts of rules? If so, would $\lambda x.\, 2x$ on 0 and 1 be a different rule than $\lambda x.\, 2x$ on 2 and 3? A functional indifferentist could respond this way:

> Of course, inputs are not parts of rules. You described one and the same rule, namely $\lambda x.\, 2x$, you just applied it to different inputs. But where is the problem with my use of "is associated with"? Every time you use a rule, you say which inputs you apply it to. It is as simple as that.

This position allows one to keep rule and domain separate and retain our intuition of what a rule is. Nonetheless, if one really regarded the domain as separate from the rule, one would not need to find a graph for every rule associated with a domain, but one for every rule *on its own* to fulfill (Rules-as-Graphs). This gives rise to the question what the graph corresponding to a rule is, irrespective of the domain associated with it. The only sensible thing a functional indifferentist could do now is to endorse

(All Inputs) The graph corresponding to a rule R is that for all of R's sensible inputs.

The reasoning behind (All Inputs) is this: If you define the graph corresponding to a rule as that set of input-output pairs which results by applying the rule to *all* sensible inputs, every graph which is the result of applying a rule to *some* of these inputs is a (possibly improper) subset of that main graph. Endorsing (All Inputs) allows the functional indifferentist to keep one's intuition of what a rule is and simultaneously to support (Rule-Inputs). To stay with the rule mentioned above, they would simply claim that the graph corresponding to $\lambda x.\, 2x$ is that for the real or complex numbers as inputs.

Problems with (All Inputs)

While using (All Inputs) works for most rules, there are some rules which pose a problem. The reason for this is that the graph corresponding to a rule is a set, so the graph's domain must be a set as well. This entails with (Rule-to-Graph) that all sensible inputs of a rule must form a set. In the following, we will show that this is *not* the case: There are some rules such that their sensible inputs do not collectively form a set.

To drive this point home, consider the most basic rule there is, $\lambda x.\, x$. Obviously, it is sensible to apply the identity rule to *anything*, be it concrete or abstract. Afterall, everything is identical with itself, so the identity rule should be able to map every object to itself as well. This gives rise to not just one, but to two antinomies at once: On the one hand, the domain associated with the identity rule would be the set of all entities whatsoever, which is, formally speaking, a universal set in a set theory with urelements. Universal sets are ruled out even in pure set theory, though, by restricted comprehension and the axiom of foundation.

On the other hand, the graph corresponding to the identity rule would need to have an input-output pair with its first *and* second component being the graph itself; after all, you can apply the identity rule to itself, which unsurprisingly again yields the identity rule. In λ-calculus, this corresponds to the one-step β-reduction

(II) $\quad (\lambda x.\, x)(\lambda x.\, x) \to_\beta \lambda x.\, x$

Set-theoretically, though, there would be an infinite nesting of the identity graph in itself, which is, of course, ruled out by the axiom of foundation. For this reason, a functional indifferentist cannot simply identify every rule with the largest set of sensible inputs, and would thus have trouble finding a graph corresponding to a rule.

Note that, although most prominent, the identity rule is not the only one to pose a problem for (All Inputs). For example, the domain associated with the singleton rule $\lambda x.\, \{x\}$ would also be the universal set in pure set theory, since the axiom of pairing in combination with the subset axiom guarantees that for every set, there exists its singleton.

To put it in a nutshell, a functional indifferentist needs both (Rule-Inputs) and (Rule-to-Graph) for (Rules-as-Graphs), but (Rule-Inputs) poses the problem of identifying a rule *itself* with a graph. The only sensible option to solve this problem, (All Inputs), gives rise to antinomy in standard set theory, though, so there is no satisfactory way to associate a rule to with some inputs. As a result, (Rules-as-Graphs) is false, even though one might have thought otherwise to begin with.

4.1.3 The Problem with (Rules-as-Graphs)

As we have shown in the last two sections, a functional indifferentist needs (Rule-to-Graph) in order to argue for (Rules-as-Graphs), which is not just plain false, but does not even suffice for (Rule-to-Graph); it requires additional assumptions to imply (Rule-to-Graph), and these assumptions cause set-theoretic antinomies. As a consequence, there are very good reasons to believe that (Rules-as-Graphs) is false.

Concerning the set-theoretic antinomies invoked by some rules, it is worth noting that the problem of too large domains was an enormous obstacle in finding proper semantics for the λ-calculus. In fact, although the λ-calculus was known already in the 1930s, the first one to find a sufficiently satisfactory mathematical model for it was Dana Scott in 1969, who in an unpublished but well-circulated paper defined models on topological spaces – and even then, the interpretation of λ-terms was restricted to those denoting (topological) continuous functions. Until today, there is no satisfactory model for the λ-calculus which allows the interpretation of every single λ-term. As Barendregt, 1981/1984, p. 5 points out, "for cardinality reasons, this is impossible". Since all the functions relevant from a point of view of mathematics or computer science get their semantic share, this fact is usually not mentioned.

4.2 Graphs as Rules

In the last section, we have seen that (Rules-as-Graphs) does not hold, and analogously, we will now show that (Graphs-as-Rules) does not hold either. What a functional indifferentist might say in defense of this claim is that we can define all graphs using a rule, and in fact, we see such definitions often times in the mathematical literature. There, you see notations like

$(\sigma_{\text{Setrule}})$ $\quad \sigma : \mathbb{N} \mapsto \mathbb{N}, x \mapsto x + 1$

Here, the rule of adding one to its argument is explicitly mentioned. This way of notation is justified by the subset axiom:

(Subset) $\quad \forall x \exists y \forall z (z \in y \leftrightarrow z \in x \wedge A[z])$

(Subset) guarantees that for any set M, there exists a subset M′ whose only members are those members of M which satisfy a condition expressible in the language of set theory. Since, by the axiom of extensionality, a set is uniquely determined by its members, we can talk about *the* subset M′ of M whose members are those members of M which satisfy a condition expressible in the language of set theory. Now rules can be turned into conditions quite easily by prepending

a variable and the identity sign. To stick with our example, we can define the graph of σ by subsetting this way:

$(\sigma_{\text{Subset}}) \quad \sigma = \{\langle x, y\rangle \in \mathbb{N} \times \mathbb{N} \mid x + 1 = y\}$

Although clearly, σ is a a set of ordered pairs, we *defined* it using its rule that for any input x, it returns $x + 1$.

4.2.1 Uniqueness

A functional indifferentist might try to push things further now, claiming that it is possible to find such a rule for *every* graph. This position takes us down a rabbit hole, which we are more than happy to leave caved in at the end of this section.

First and foremost, it is worth mentioning that even if there was a possibility to find *some* rule for every graph, one had to concede that there is no *unique* rule for every graph. As an example, there are quite some rules which yield the graph $\{\langle 2, 4\rangle\}$ for the input 2:

(1)	multiply the argument by 2	$\lambda y.\, 2 \times y$
(2)	multiply the argument by itself	$\lambda y.\, y \times y$
(3)	always return 4	$\lambda y.\, 4$

This is nothing a functional indifferentist could deny, but they might shrug their shoulders and point out that this is not a requirement of (Graphs-as-Rules). After all, it reads "a" and not "a unique".

4.2.2 Random Finite Graphs

Where things start to turn nasty is when looking at graphs with randomly assembled argument-value pairs, as for example

(Random) $\{\langle 0, \pi\rangle, \langle 34, e\rangle, \langle 135.5, 2\rangle\}$

Looking at the graph gives a good feeling that there is no general rule for (Random), and even if there was, one could keep adding random argument-value pairs until no such rule could be found. This poses a problem for the functional indifferentist, but they might respond this way:

> Yes, there is no *general* rule for (Random), but I never claimed there existed one. What I *did* claim with (Graphs-as-Rules) was that there is simply *some* rule. Show me any graph you like and I will find a rule for you. I can even give you an algorithm to do so: For every argument-value pair $\langle y, z\rangle$, add a conditional statement of the form "if $x = y$, then $f(x) = z$".

Applying this algorithm to (Random) would yield

$$f(x) = \begin{cases} \pi, & \text{if } x = 0 \\ e, & \text{if } x = 34 \\ 2, & \text{if } x = 135.5 \end{cases}$$

Although this move reminds one of a sleight of hand, there is no magic involved: For every graph you dictate, you can create a pseudo-rule using conditional statements and the identity sign. It surely is not an attractive solution, but it is a solution nonetheless.

4.2.3 Random Infinite Graphs

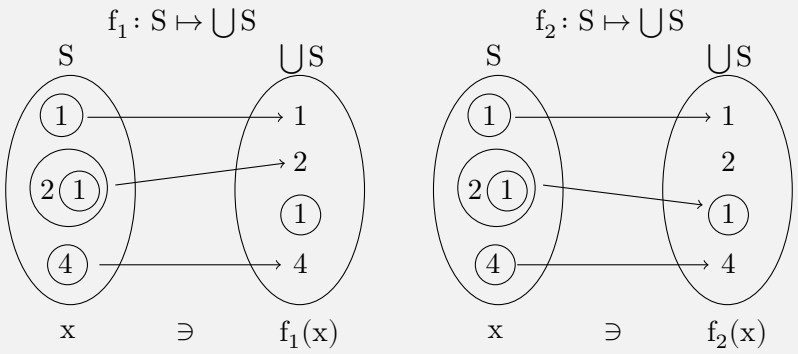
Let us, then, administer the coup de grâce to (Graphs-as-Rules). As point (1) of Definition 2.2 makes clear, an inherent property of rules is that they can be described, and we were just provided with an algorithm to describe rules corresponding to random graphs. The crux, however, is that you need to add a conditional statement for every single argument-value pair, and since you can only write down finitely many such conditional statements, finding a description of a rule corresponding to a random graph only works if it is *finite*; rules for random *infinite* graphs do not exist because they cannot be described.

A functional indifferentist might lull themselves into a false sense of security by claiming that this is not a problem because infinite graphs are only produced by rules. This is not correct, though, since in standard set theory, the existence of such infinite graphs not defined by a rule is guaranteed by the axiom of choice:

(Choice) $\quad \forall x(\emptyset \notin x \rightarrow \exists f(f : x \mapsto \bigcup x \wedge \forall y(y \in x \rightarrow f(y) \in y)))$

(Choice) states that for every set S not containing the empty set, there exists a *choice graph* whose domain is S, and whose value for every member $s \in S$ is some member of s. We use the word "choice graph" instead of the common "choice function" here to avoid an implicit endorsement of (Functions-as-Graphs). To give an example of (Choice) in action, consider the set $S = \{\{1\}, \{2, \{1\}\}, \{4\}\}$. Figure 4.1 shows the two choice graphs one of whose existence (Choice) guarantees.

Figure 4.1: An Example of the Axiom of Choice



Now one does not need the axiom of choice for finite sets, since it is easy to prove that every choice graph for every finite set exists. For *infinite* sets, things look different, though – in ZF, there is a myriad of infinite sets without a choice graph. The axiom of choice guarantees a choice graph for *any* set not containing the empty set – even for uncountably infinite ones.

While one could certainly write whole books about the axiom of choice (cf. Herrlich, 2006 or Jech, 2008), we just need one crucial fact about it: It only guarantees the existence of *some* choice graph for every set not containing the empty set, without further specifying it. This non-constructive character suffices already to take the wind out of the functional indifferentist's sails. To drive the point home, we show that there exists a graph whose domain, range and members cannot be determined. For the sake of the argument, consider that set M whose members are all the pairwise disjunct doubletons containing consecutive natural numbers:

(M) $\{\{0,1\},\{2,3\},\{4,5\},...\}$

Now since, by construction, M does not contain the empty set, the axiom of choice guarantees that there exists a choice graph for M. Note that there are $2^{|M|} = |\mathbb{N}|$ *possible* choice graphs; the axiom of choice only guarantees the existence of *at least one* such choice graph. But how does it look like? What are its members? We don't know, and we never will.

If one wanted, one could even go a step further now: Let us call the choice graph whose existence is guaranteed by the axiom of choice c. Although we do not know the natural numbers c picks for the members of M, we know its domain, which is, by definition, M. We also know that c is a set of ordered pairs, and all ordered pairs are two-element sets, so c does not contain the empty set.

Thus, the choice graph c *itself* has a choice graph c' which to every member x of c assigns one of x′s members. Since we do not know c, and since c is the domain of c', we do not know the

domain of c' either. Thus, we know that a graph exists, but we know neither its domain nor its range, nor even a single one of its pairs. We only know by construction that each argument of c' is a doubleton consisting of a two-element set containing consecutive natural numbers and a one-element set which is a subset of the two-element set, and that each value of c' is either one of these two sets.

As we have just shown, the axiom of choice guarantees the existence of graphs whose argument-value pairs we do not know, but since we need to know a graph in order to find a rule corresponding to it, it is safe to say that there is no rule corresponding to these kind of graphs. Even though one might be able prove that every choice graph exists for (M) irrespective of (Choice), there are enough (and more complex) cases of choice graphs whose existence is guaranteed *only* by (Choice), and the pattern used above can be applied to them as well. Thus, there is at least one graph of which there is no corresponding rule, contradicting (Graphs-as-Rules).

4.2.4 Intuitionism to the Rescue?

A functional indifferentist could now, as a last resort, endorse an intuitionistic position, claiming that the axiom of choice does or should not hold. Even this does not work, though, since intuitionists usually use a weaker axiom instead, which produces the same problem:

(Countable Choice) $\forall x(\emptyset \notin x \wedge |x| \leq |\mathbb{N}| \rightarrow \exists f(f : x \mapsto \bigcup x \wedge \forall y(y \in x \rightarrow f(y) \in y)))$

(Countable Choice) differs from (Choice) only by the second conjunct in its antecedent, making sure that choice graphs exist for *countable* sets; the existence of choice graphs for *uncountable* sets is not stipulated. Intuitionists endorse this axiom in order to be able to prove intuitively valid statements, as for example the statement that the union of countably many countable sets is itself countable. Whether uncountable or countable, though, (Countable Choice) suffices to construct infinite random graphs for which you cannot find a rule, so a functional indifferentist could not even endorse a standard intuitionistic view and would need to commit oneself to the rather weak ZF version of set theory.

4.3 Functional Indifferentism Debunked

In the preceding two sections, we have shown that (Functions-as-Rules) and (Functions-as-Graphs) do not yield the same functions. We have done so by constructing the position of a functional indifferentist who claims the conjunction of (Rules-as-Graphs) and (Graphs-as-Rules), and examined these claims in isolation. As a result, both are hard to argue in favor of. (Rules-as-Graphs) causes trouble because it presupposes (Rule-to-Graph), which is plain false, and even if it was

true, one would need to make assumptions in order to deduce (Rules-as-Graphs), and they induce set-theoretic antinomies. (Graphs-as-Rules) poses the problem of finding rules for unknown infinite graphs whose existence is guaranteed even by the weaker intuitionistic version of the axiom of choice. While it might be possible to construct a theory of sets compatible with (Graphs-as-Rules), there is good reason to assume it would not be strong enough as a mathematician's tool.

These results show that (Functions-as-Rules) and (Functions-as-Graphs) are positions that are not just relevant from an ontological point of view, but also from a *formal* perspective: There are rules which do not have a graph, and there are graphs for which is there is no rule. Note in particular that the functions one commits oneself to with (Functions-as-Graphs) are not all among the ones one commits oneself to with (Functions-as-Rules), although the alternative labels "function-in-intension" and "function-in-extension" (cf. Révész, 1988, p. 5) suggest otherwise.

An alert reader proficient in the relevant mathematical literature may find the result of this chapter surprising. After all, λ-calculus and set theory are often treated as if they defined the same set of functions. However, there is two good reasons for this: On the one hand, in λ-calculus, you have some λ-terms which denote functions that are superfluous from a mathematician's perspective (and for which we do not even have a sensible interpretation; see Section 4.1.2). On the other hand, it is not interesting for a mathematician to find a rule for a graph, since they think of functions as rules anyway. The reason why the systems are often times treated as if they were equivalent is that the functions *relevant* as tools for a mathematician – continuous and effectively computable functions – can all be defined using both systems. To put it metaphorically: In λ-calculus, there are some tools in the shed that are not used in mathematical practice. In set theory, there are some tools in the shed that are used without even knowing what they do exactly.

As a last comment, note that (Functions-as-Rules) is still compatible with (Rule-Inputs). In fact, there is a formal machinery called the *typed λ-calculus* which does just that. There, you assign a type to every λ-term, which can be interpreted as its domain; this disallows terms like (Ω). While it would surely be interesting to investigate things further, it is beyond the scope of this thesis, and it suffices for us to know that the problem of infinite graphs generated by the axiom of choice still persists, which leaves functional indifferentist the extraordinarily unattractive view that it is. The reader interested in the typed λ-calculus is referred to Barendregt et al., 2013, and those interested in the connection between natural deduction proofs, the typed λ-calculus and the modeling of propositions as types is invited to consult Wadler, 2015.

5 Conclusion

The aim of this thesis was twofold: First and foremost, I argued that the question whether functions are graphs or rules is an important one, since the functions defined by (Functions-as-Graphs) are not the ones defined by (Functions-as-Rules). Secondly, on the way to this main goal, I provided an introduction to the λ-calculus suited for philosophers with a background in logic an set theory.

In retrospect, we quickly covered some very general properties of functions in Chapter 2. Then, in Chapter 3, we gave the just mentioned introduction to the standard system of λ-calculus by explaining its most important ideas and by having a close look at its syntax, substitution of free variables and its axiom system. Having made the necessary preparations, Chapter 4 then argues for my main claim by constructing the position of functional indifferentism, which claims that (Functions-as-Graphs) and (Functions-as-Rules) yield the same functions, to then debunk it in a strict and rigorous manner. Along the way, we contrasted the dynamic character of rules with the static character of graphs.

To conclude this thesis, I lay open which related topics it did not cover. Section 5.1 points to side issues for further reading, Section 5.2 discusses the areas of research concerning this topic that are in need of further investigation.

5.1 Some Loose Ends

Unfortunately, due to the space limitations of this thesis, I needed to make plenty painful decisions to leave out interesting and important points, most of them concerning Chapter 2 and Chapter 3. The whole area of combinatory logic was not covered, and the connection to the λ-calculus was left out (see Curry and Feys, 1958). Also, a discussion of representing logical connectives (cf. Hankin, 2011, pp. 41–44) in λ-calculus would have made clear how strong the system is, allowing to formalize !, nobase and the rule corresponding to (Random).

Related to this issue, recursion was not covered formally, and the search for fixed-points using the Y-Combinator did not find its place in this thesis either. In addition, the whole topic of semantics (Bimbó, 2020, pp. 47–53) for the λ-calculus was left out almost entirely. It only sufficed for some quick notes in Section 4.1.3; the typed λ-calculus (see Barendregt et al., 2013) suffered the same fate. Here, it would have been interesting to see how proofs in predicate logic can be translated into λ-terms, and why the law of excluded middle does not hold under this translation (see Wadler, 2015).

Last but not least, what ached the most was mentioning extensionality (see Hindley and Seldin, 2008, pp. 76–81) only very briefly. There are several competing axioms and proof rules claiming to make the system extensional, and subtle differences between them, which makes it interesting from a philosophical perspective. The ingenious difference-making λ-terms, as for example the class of universal generators – λ-terms which β-reduce ad infinitum in such a way that the Gödel number of every λ-term is subterm of one of its contracta (see Plotkin, 1974) –, had to surrender to the space requirements.

5.2 Desiderata

Next to the topics that were not covered in this thesis although there exists some corresponding literature, there are also some areas in need of further investigation. On the one hand, most texts that mention the history of the function concept treat it with shockingly little historical or scientific scrutiny. There is not a single introduction, companion, guide or handbook on this topic, forcing Youschkevitch, 1976, p. 37 to admit in one of the few somewhat partial overviews that "[u]p to now the history of functionality has remained insufficiently studied". In the almost fifty years up until now, this grievance was not done away with.

On the other hand, the question of what a rule is exactly is not even mentioned in the literature that introduces the λ-calculus as the theory behind (Functions-as-Rules). Formally, this thesis rests on the assumption that rules are sufficiently described by the axioms of the theory λ, but there are good reasons to doubt this. As a first point, there are theories competing with λ – some of them untyped, some of them typed –, and no one has ever investigated the differences with respect to (Functions-as-Rules): Which theories better suit our informal understanding of the word "rule", which problems arise, and how can you solve them?

As a second point, it is unclear which λ-terms denote rules: Of course, abstraction terms are the formal representation of descriptive function signs, so they denote rules, but what about application terms and variables? Application terms do not seem to denote rules themselves. However, they are often times – but not always, cf. "xy" – β-equivalent with abstraction terms. Also, what is the status of variables? Do they denote rules, or are they able to denote other objects as well? If so, why can you apply any variable to any other variable, even if they are both objects?

All of these questions were and had to be neglected in this thesis due to the space requirements, and in hindsight, they should have been developed in a less space-restricted setting. Nonetheless, this only emphasizes the amount of desiderata concerning this topic, and the need to address them, which makes me wish that it will receive the scholarly attention it truly deserves in the months, years and decades to come.

Bibliography

Alama, J., & Korbmacher, J. (2021). The Lambda Calculus. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2021). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/sum2021/entries/lambda-calculus/. (Cit. on p. 5)

Barendregt, H. P. (1984). *The lambda calculus: Its Syntax and Semantics* (Rev. ed). North-Holland. (Original work published 1981). (Cit. on pp. 22, 35)

Barendregt, H. P., Dekkers, W., & Statman, R. (2013). *Lambda calculus with Types*. Cambridge University Press. (Cit. on pp. 40, 41).

Bimbó, K. (2020). Combinatory Logic. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2020). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2020/entries/logic-combinatory/. (Cit. on p. 41)

Cardone, F., & Hindley, J. R. (2009). Lambda-Calculus and Combinators in the 20th Century. In *Handbook of the History of Logic* (pp. 723–817). Elsevier. https://doi.org/10.1016/S1874-5857(09)70018-4. (Cit. on p. 8)

Church, A. (1932). A Set of Postulates for the Foundation of Logic (1). *Annals of Mathematics*, *33*(2), 346–366. https://doi.org/10.2307/1968337 (cit. on pp. 8, 15)

Church, A. (1933). A Set of Postulates For the Foundation of Logic (2). *Annals of Mathematics*, *34*(4), 839–864. https://doi.org/10.2307/1968702 (cit. on pp. 8, 15)

Curry, H. B. (1942). The Inconsistency of Certain Formal Logics. *The Journal of Symbolic Logic*, *7*(3), 115–117. https://doi.org/10.2307/2269292 (cit. on p. 8)

Curry, H. B. (1972). *Combinatory Logic* (Vol. 2). Elsevier. (Cit. on pp. 15, 31).

Curry, H. B. (1980). Some Philosophical Aspects of Combinatory Logic. In *Studies in Logic and the Foundations of Mathematics* (pp. 85–101). Elsevier. https://doi.org/10.1016/S0049-237X(08)71254-0. (Cit. on p. 25)

Curry, H. B., & Feys, R. (1958). *Combinatory Logic* (Vol. 1). North-Holland. (Cit. on pp. 14, 20, 26, 41).

Frege, G. (1883). *Grundgesetze der Arithmetik*. Olms. (Cit. on p. 25).

Frege, G. (1891, January 9). *Funktion und Begriff*. Jena. (Cit. on pp. 4, 5).

Frege, G. (1904). Was ist eine Funktion? In S. Meyer (Ed.), *Festschrift Ludwig Boltzmann gewidmet zum sechzigsten geburtstage 20. Februar 1904* (pp. 656–666). J. A. Barth. (Cit. on p. 3).

Hankin, C. (2011). Lambda Calculi: A Guide. In D. M. Gabbay (Ed.), *Handbook of philosophical logic. Vol. 15* (2. ed, pp. 1–66). Springer. (Cit. on p. 41).

Herrlich, H. (2006). *Axiom of Choice*. Springer. (Cit. on p. 38).

Hindley, J. R., & Seldin, J. P. (2008). *Lambda-calculus and combinators, an introduction*. Cambridge University Press. (Cit. on p. 42).

Jech, T. J. (2008). *The Axiom of Choice* (Dover ed). Dover Publications. (Cit. on p. 38).

Kleene, S. C., & Rosser, J. B. (1935). The Inconsistency of Certain Formal Logics. *Annals of Mathematics*, *36*(3), 630–636. https://doi.org/10.2307/1968646 (cit. on p. 8)

Neumann, J. v. (1928). Über die Definition durch Transfinite Induktion und Verwandte Fragen der Allgemeinen Mengenlehre. *Mathematische Annalen*, *99*(1), 373–391. https://doi.org/10.1007/BF01459102 (cit. on p. 31)

Plotkin, G. D. (1974). The λ-calculus is ω-incomplete. *The Journal of Symbolic Logic*, *39*(2), 313–317. https://doi.org/10.2307/2272645 (cit. on p. 42)

Quine, W. V. (1969). *Methods of Logic*. Harvard University Press. (Cit. on p. 20).

Révész, G. E. (1988). *Lambda-calculus, Combinators, and Functional Programming*. Cambridge University Press. (Cit. on pp. 21, 40).

Rosser, J. B. (1984). Highlights of the History of the Lambda-Calculus. *Annals of the History of Computing*, *6*(4), 337–349. https://doi.org/10.1109/MAHC.1984.10040 (cit. on p. 8)

Schönfinkel, M. (1924). Über die Bausteine der Mathematischen Logik. *Mathematische Annalen*, *92*(3), 305–316. https://doi.org/10.1007/BF01448013 (cit. on p. 24)

Scott, D. (1973). Models for Various Type-Free Calculi. In P. Suppes, L. Henkin, A. Joja, & G. C. Moisil (Eds.), *Studies in Logic and the Foundations of Mathematics* (pp. 157–187). Elsevier. https://doi.org/10.1016/S0049-237X(09)70356-8. (Cit. on p. 28)

Seldin, J. P. (2009). The Logic of Church and Curry. In *Handbook of the History of Logic* (pp. 819–873). Elsevier. https://doi.org/10.1016/S1874-5857(09)70019-6. (Cit. on p. 8)

Wadler, P. (2015). Propositions as Types. *Communications of the ACM*, *58*(12), 75–84. https://cacm.acm.org/magazines/2015/12/194626-propositions-as-types/fulltext (cit. on pp. 40, 41)

Youschkevitch, A. P. (1976). The Concept of Function up to the Middle of the 19th Century. *Archive for History of Exact Sciences*, *16*(1), 37–85. https://doi.org/10.1007/BF00348305 (cit. on p. 42)

Eigenständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Arbeit *Functions as Graphs, or Functions as Rules? A Philosophical Introduction to the* λ*-calculus* selbstständig von mir und ohne fremde Hilfe verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind. Mir ist bekannt, dass es sich bei einem Plagiat um eine Täuschung handelt, die gemäß der Prüfungsordnung sanktioniert werden kann.

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in einer Datenbank einverstanden.

Ich versichere, dass ich die vorliegende Arbeit oder Teile daraus nicht anderweitig als Prüfungsarbeit eingereicht habe.

(Ort, Datum)	(Unterschrift)