



ulm university universität
uulm

**Fakultät für Mathematik
und Wirtschaftswissenschaften**
Institut für Algebra und
Zahlentheorie

**Fakultät für Ingenieurwis-
sensschaften, Informatik und
Psychologie**
Institut für theoretische Informatik

Cartesian Closed Categories and the Simply Typed Lambda Calculus

Abschlussarbeit an der Universität Ulm

Vorgelegt von:

Simone Kilian
simone.kilian@uni-ulm.de
1035995

Betreuer:

Prof. Dr. Jeroen Sijsling

Gutachter:

Prof. Dr. Jacobo Torán

2023

Fassung August 15, 2023

© 2023 Simone Kilian

Satz: PDF-L^AT_EX 2_ε

Contents

1	Preface	3
1.1	Motivation	3
1.2	Content	5
2	Lambda Calculus	6
2.1	On the Untyped Lambda Calculus	7
2.2	Intuitionistic Propositional Logic	20
2.3	The Simply Typed Lambda Calculus λ_{\rightarrow}	27
2.3.1	The Curry-Howard Isomorphism	36
2.3.2	Extended Curry-Howard Isomorphism	39
3	Categories	45
4	Cartesian Closed Categories and a Typed Lambda Calculus	57
4.1	Cartesian Closed Categories	57
4.1.1	Products in Arbitrary Categories	57
4.1.2	Exponential Objects	59
4.1.3	Cartesian Closed Category	61
4.2	Typed λ Calculus and Categories	62
4.2.1	(Simply) Typed λ -Calculus to Cartesian Closed Category .	64
4.2.2	Cartesian Closed Category to (Simply) Typed λ -Calculus .	67
4.2.3	Similarities in Logic, Typed λ -Calculus, and Category Theory	69
5	Conclusion	70
A	Appendices	73
A.1	Basic Definitions	73
A.1.1	Sets	73

<i>Contents</i>	

A.1.2	Functions	74
A.1.3	Graphs	75
A.1.4	Homomorphisms Of Graphs	77
A.1.5	Categories	79
A.2	Fixed Points in Untyped λ -Calculus and Cartesian Closed Categories	81
A.2.1	Fixed Points in the Untyped Lambda Calculus	86
A.2.2	Fixed points in cartesian closed categories	91
Bibliography		93

Abstract

Connections between type theory and category theory have been studied since the 80s. This connection is part of a bigger interconnection linking fields like algebraic geometry, computability theory, homotopy theory, set theory, topology, and topos theory within mathematical logic. In this work we analyze the connection between cartesian closed categories and typed λ -calculi, a correspondence discovered by Lambek and Scott [1]. This involves showing that we can associate a cartesian closed category to a typed λ -calculus. Conversely, the language of a cartesian closed category is a typed λ -calculus. If we reconstruct a category from this internal language, then the obtained category is equivalent to the initial cartesian closed category. Lambek and Scott [1] also show that if we construct a corresponding category from a typed λ -calculus, then the internal language and the original typed λ -calculus are equivalent. We focus on the simply typed λ -calculus and its corresponding intuitionistic logic that is part of the Curry-Howard isomorphism. This approach differs a little bit from the set theoretic approach on corresponding types of Lambek and Scott. As a result, we obtain a correspondence between intuitionistic logic, (simply) typed λ -calculus and category theory.

I would like to thank Jeroen Sijssling for the supervision and the good comments, advice, and detailed feedback. I would like to thank Jacobo Torán for the work that comes with the assessment of the thesis. Having this construction for a thesis posed extra work on both of you - thank you for this agreement that made it possible for me to work with this interesting topic.

1 Preface

1.1 Motivation

Type theories include lower and higher order λ -calculi, combinatory calculus (traditionally called combinatory logic), Gödel's system \mathcal{T} with its correspondence to Kreisel's modified realizability and Heyting arithmetic, polymorphic type theory like for example System F ($\lambda 2$), $\lambda\omega$, where we have type constructors (kinds), and the Calculus of Constructions where the other systems of the λ -cube are embedded. Some λ -calculi and type theories can be seen as abstract models of computation. Some of them relate to weaker and some to stronger computability concepts. The approach of combinatory logic is that we can represent typed constants as logical axioms forming combinators. These combinators combine terms by means of an application. Both the λ -calculus and systems of combinatory logic were introduced as a foundation of mathematics and logic and turned out to be inconsistent due to the presence of arbitrary fixed points. Nevertheless, the systems give us, for example, useful subsystems to study. They turned out to be helpful in the theory of programming languages forming the *functional programming paradigm*.

The Curry-Howard isomorphism shows a correspondence between systems of formal logic (like intuitionistic logic) and (computational) typed λ -calculi forming the *propositions as types* or *proofs as programs* paradigm. This isomorphism shows that both the logic that relates to a specific type theory as well as the type theory are constructive approaches, since they are defined through a deductive system corresponding to a constructive logic. In particular, there is an important idea that we regard as the basic idea behind the logics relating to type theories. We introduce it as the BHK (Brouwer, Heyting and Kolmogorov) interpretation. Here we understand a function from A to B as a procedure that transfers proofs of A

into proofs of B . This constructiveness can be found in intuitionistic logic. This interpretation is an important foundation in the understanding of λ -systems but also in the understanding of intuitionistic logic. At least from a classical point of view intuitionistic logic is very unintuitive. Through the correspondence, we get a better understanding of the constructive nature of this particular logic.

Another reason to study the connection between the λ calculus and category theory is to better understand λ -definability. For the untyped λ -calculus, λ -definability is equivalent to Turing computability. The λ -definable raw terms in the simply typed λ -calculus are the ones that always terminate and can be derived through the deductive system from the empty set. This is a far weaker computability concept. We obtain another view of computability theory, also because we relate the simply typed λ -calculus to logic and later to category theory. This might lead to deeper insight and to another perspective.

Category theory is the study of mathematical structures and their relations that appear throughout mathematics. It is an expression of the *morphism composition paradigm*. Category theory does not focus on the elements of the objects but on the morphisms between them. Natural transformations (morphisms between functors) were the reason why categories were introduced.

We will see that typed λ -calculi and cartesian closed categories naturally relate to each other. Given a (simply) typed λ -calculus we can construct a category that is cartesian closed. Implying that the transformation from a cartesian closed category to a typed λ -calculus is innocuous to the initial category. Conversely, the internal (logical) language of a cartesian closed category is a typed λ -calculus. If we reconstruct a category from this internal language, then the obtained category is equivalent to the initial cartesian closed category. Lambek and Scott [1] show that if we construct from a typed λ -calculus a corresponding category, then the internal language and the original typed λ -calculus are equivalent. The transformation of a typed λ -calculus into a cartesian closed category is innocuous to the typed λ -calculus. We obtain that we can regard cartesian closed categories as models for a typed λ -calculus.

These results imply that these two concepts are extensionally equivalent, which allows us to transfer results from one to the other. As a result, we obtain a correspondence between logic, typed λ -calculus and category theory.

1.2 Content

In *Chapter 2* we give an introduction to the untyped λ -calculus, propositional intuitionistic logic, and the simply typed λ -calculus. We introduce the extended correspondence between propositional intuitionistic logic and the simply typed λ -calculus that is part of the Curry-Howard isomorphism.

In *Chapter 3* we introduce some of the basic notions of category theory. We will introduce functors (morphisms between categories), terminal and initial objects, and some useful properties and concepts.

In *Chapter 4* we introduce products, exponential objects and cartesian closed categories. We connect the exponential object with currying. We will see that the λ -calculus and cartesian closed categories relate to each other. Given a (simply) typed λ -calculus we can construct a category that is cartesian closed. Conversely, the internal language of a cartesian closed category is a typed λ -calculus.

Reconstructing a category from this internal language, we obtain a category that is equivalent to the initial cartesian closed category. Consequently, transforming a cartesian closed category to a typed λ -calculus is innocuous to the initial category. Conversely, if we construct from a typed λ -calculus a corresponding category, then the internal language and the original typed λ -calculus are equivalent (Lambek and Scott [1]). Hence, we obtain that transformation from a typed λ -calculus into a cartesian closed category is innocuous to the typed λ -calculus. This shows that we can regard cartesian closed categories as models for a typed λ -calculus.

We conclude that these results imply that the two concepts are extensionally equivalent, which allows us to transfer any result from one to the other.

Finally, we obtain a correspondence between logic, a typed λ -calculus and category theory.

2 Lambda Calculus

The Lambda notation has been introduced by Alonzo Church in 1932 to simplify combinatory logic in order to develop a general theory of functions and extend it with logic to provide a foundation of mathematical logic (for a detailed introduction see Barendregt [2]). Moreover, the λ -calculus is a model for the paradigm of functional programming languages that implement the λ -abstraction. Parameters of a procedure correspond to the bound variables in λ -abstractions. Although it serves as a model of functional programming, it is also a simple programming language itself.

The λ -calculus can be seen as a theory of abstract functions that focuses on the syntax and the intension of a function and manipulates functions as expressions (syntactically).

We compare two possible ways to define equality of functions: Two functions with the same domain and codomain that are applied on the same arguments and that return for every argument the same result, are equal. This is an application of the logical principle of extensionality. With regard to ZF set theory and the axiom of extensionality we might understand a function as a set of tuples (see definition A.1.1 for a definition). If so, then informally stated the two sets that represent functions are equal if their graphs contain the same elements.

In contrast, the older point of view is to see functions as formulae. These formulae are rules. Such a rule states how the formula shall be calculated. The familiar notion $f(x) = x^2$ is such a rule. We could also compare these rules in an intensional way. Two functions are intensionally equal if they essentially describe the same rule. This means that two functions f, g are intensionally equal if we can prove that $f = g$. This also implies that domain and codomain do not always have to be specified to compare a function. For the identity $\lambda x.x$, for example, it is not important which domain and codomain we have. It works on any domain. If we

take an example from computer science where we would say that two functions are intensionally equal if they implement the same source code up to substitution. This example makes the difference clearer. If two functions that behave the same, i.e. they are extensionally equivalent, are implemented differently, they are not intensional equivalent. One may compare trial division and the AKS prime test. Also the side effects may differ.

We define relations on terms (through reductions) in order to show that functions that seem to be different reduce to the same function and obtain a definitional equality by means of these reductions, especially through β -reduction.

Firstly, we will introduce the lambda calculus without types. We start defining the syntax, introduce reduction rules, and usefull properties as Church-Rosser, normalization and λ -definability. Finally, we introduce the simply typed lambda calculus that will become an example of a type theory that relates to cartesian closed categories. In parts, we follow Selinger [3] and Sørensen and Urzyczyn [4] in this introduction. A standard source for the λ -calculus is Barendregt for example in [2].

2.1 On the Untyped Lambda Calculus

Functions can be considered as programs, given to computing machines. These machines operate on them with no regard on which type exactly they are working with. Consider for example a turing machine working on a binary alphabet. This Turing machine can be regarded as a (computable) function. But if we have a Gödel number of a Turing machine we can also input it into a turing machine.

In the untyped λ -calculus it is also possible to input any term into a λ -abstraction. Every term can be regarded as a function of a certain arity. Of course we have also terms of arity 0. If a function f is applied to itself this is paradoxical in the same sense as Russel's paradox A.1.1. But this not necessarily a bad thing. The untyped λ -calculus gives us the possibility to do recursion via using fixed points using for example Curry's or Turing's fixed point combinator.

It is possible to define non-terminating functions in the untyped λ -calculus. This is similar to infinite loops or non-terminating recursion in programming. A theoreti-

cally desirable property of a programming language is that if we want a program written in this language to terminate it should terminate. Therefore, we are interested in the so called Church-Rosser property (confluence) and (strong) normalization in λ -calculi. Strong normalization (every reduction sequence is finite) and normalization (there is a reduction sequence ending in a normal form) gives us existence, and together with the Church Rosser property the uniqueness of a normal form (no further reducible form). We will introduce a specific reduction strategy that guarantees us to reduce to a normal form if it exists. In fact, in the simply typed λ calculus we will have that every reduction sequence ends in a normal form. There we obtain a stronger result, and together with the subject reduction property (a reduced term has the same type as the initial one) we get the termination of a typed term in a normal form of the same type.

The untyped λ -calculus and its syntax are the basis of typed λ -calculi. Functions are anonymous and are not acting on a defined domain and codomain.

The language of the λ -calculus consists of individual variables, λ -functions of the form $\lambda x.t$ and parentheses.

Definition 2.1.1 (λ -Terms). The set of λ -terms Λ is recursively defined as follows.

- (i) Every individual variable x, y, z, \dots is a λ -term.
- (ii) If $s, t \in \Lambda$, then the application $(s\ t)$ is a λ -term.
- (iii) If $t \in \Lambda$ and x is an individual variable then the λ -abstraction $(\lambda x.t) \in \Lambda$.
Note that a λ -abstraction binds precisely one variable.
- (iv) Nothing else is a λ -term.

On the one hand every λ -term is a function and every function is a formula, on the other every λ -term can be an argument to another function.

Definition 2.1.2 (Abbreviations). Let $s, t, u \in \Lambda$.

Application $s\ t\ u$ is by convention left-associative. We may write $s\ t\ u$ for $((s\ t)\ u)$. $\lambda xyz.z\ x\ y$ is an abbreviation for $\lambda x.\lambda y.\lambda z.z\ x\ y$.

Outer parenthesis as in $(\lambda x.x) = \lambda x.x$ are omitted.

The scope of a λ -abstraction extends to the closing parenthesis of that specific λ -abstraction. For example, the first λ -abstraction λx in $\lambda x.\lambda y.x\ y\ (\lambda z.z)\ x = (\lambda x.\lambda y.x\ y\ (\lambda z.z)\ x)$ binds the last x in the λ -term.

Example 2.1.1. The λ -term $x\ z\ \lambda x y z.x\ \lambda z.z\ y$ is the abbreviation of $((x\ z)\ (\lambda x.(\lambda y.(\lambda z.((z\ (\lambda z.(z)))\ y))))))$.

Definition 2.1.3 (Bounded and Free Occurrences of Variables). An occurrence of a variable x in a term t is bound if and only if it is within an occurrence in t of the form $\lambda x.s$. Otherwise it is free.

Example 2.1.2. In the term $x\ \lambda x.x\ y\ \lambda x.z\ x$ the variables x, y, z occur freely. The variable x is bound by two different λ -abstractions. The first λ -abstraction binds the second occurrence of x and the second λ -abstraction binds its last occurrence. The first x is not bound at all.

A term with no free variables is called a *combinator*, for example the identity $I := \lambda x.x$ is a combinator.

Terms can occur in a form that still can be reduced to a most minimal form. This form is no further reducible. Later we will call this most minimal form the normal form. Other terms are not reducible to a most minimal form at all. The notion of evaluation is captured by the idea of a reduction. This means we like to define evaluation as a reduction or a sequence of reductions to a non-reducible form. Later we will see that it is not possible to find a non-reducible form for every λ -term. Terms can be reduced via β -reductions and η -reductions. In order to reduce a term we sometimes have to rename bound variables. This is what we call an α -conversion.

In order to perform β -reductions we need to define substitution rules that do no harm the intension of the specific term.

Definition 2.1.4 (Substitution Rules). Let s, t, u denote terms and x, y, z individual

variables. We inductively define substitutions as follows

$$\begin{aligned}
 x[s/x] &= s \\
 y[s/x] &= y \\
 (t \ u)[s/x] &= (t[s/x] \ u[s/x]) \\
 (\lambda x.t)[s/x] &= (\lambda x.t) \\
 (\lambda y.t)[s/x] &= \lambda y.(t[s/x]), & \text{if } y \text{ does not occur freely in } s \\
 (\lambda y.t)[s/x] &= \lambda z.(t[z/y][s/x]), & \text{if } y \text{ occurs freely in } s, z \text{ is a new variable}
 \end{aligned}$$

The particular choice of a variable does not matter. Note that $[x := s]$ (used in the context of the λ calculus), $[s/x]$ (used in the context of natural deduction) and $[x/s]$ (in the context of computer science) are different notations for the substitution where we replace x by s . We use the notation $[s/x]$ throughout the thesis. An α -conversion is the renaming of bound variables in a λ -abstraction.

Definition 2.1.5 (α -Conversion). Let $\lambda x.t$ be a λ -abstraction and γ a variable. If γ does not occur freely in t , then $\lambda x.t[\gamma/x]$ is the result of replacing every free occurrence of x in t with γ and replacing x with γ in this λ -abstraction. This is called an α -conversion. We denote the α -conversion of the term $\lambda x.t$, where x is replaced by α , as $\lambda x.t \rightarrow_\alpha \lambda \gamma.(t[\gamma/x])$. This is captured by the following rule

$$\frac{\gamma \notin t}{\lambda x.t =_\alpha \lambda \gamma.(t[\gamma/x])} \ (\alpha)$$

We assume that the reader is familiar with logical notation. Here, we mean an implication in classical logic that is commonly used in mathematics. The consequent can be understood as a rule in a rewriting system, where $=_\alpha$ or \rightarrow_α denotes a single rewriting step. The following examples are done via this notation. Of course, a form that can no further be α -reduced can never be obtained via α -conversions. Sørensen and Urzyczyn [4] define the notion of a term up to α -equivalence. They call what we call terms, raw terms, and put this raw terms into α -equivalence classes. Each α -equivalence class on a raw term is what they call a term.

Example 2.1.3. The conversions $\lambda x.x \lambda x.x \rightarrow_\alpha \lambda y.y \lambda x.x$ and $\lambda x.\lambda x.y \ x \rightarrow_\alpha \lambda z.\lambda x.y \ x$ are innocuous. In contrast, the following conversions are not allowed. They harm the intended semantics of a term.

In $\lambda x.x \lambda x.x \not\rightarrow_\alpha \lambda z.z \lambda x.z$ we would have substituted an a variable with the same name, but bound to another abstraction.

In $\lambda x.\lambda x.y \ x \not\rightarrow_\alpha \lambda y.\lambda x.y \ x$ we would bind the former free variable y to the initial λ -abstraction.

Definition 2.1.6 (Relations on λ -terms cf.[3]). A relation on λ -terms is an equivalence relation if it is *reflexive*, *transitive* and *symmetric* (see below). An relation on λ -terms is a congruence relation if it also satisfies the rules (*cong*) and (ξ) in the following definition.

Definition 2.1.7 (Rules for Equivalence and Congruence relations for $=_\alpha$ on the set of λ - terms cf.[3]).

$$\begin{array}{c} \frac{}{s =_\alpha s} \text{ (refl)} \\ \frac{s =_\alpha t}{t =_\alpha s} \text{ (sym)} \\ \frac{s =_\alpha t \quad t =_\alpha u}{s =_\alpha u} \text{ (trans)} \\ \frac{s =_\alpha s' \quad t =_\alpha t'}{s \ t =_\alpha s' \ t'} \text{ (cong)} \\ \frac{t =_\alpha t'}{\lambda x.t =_\alpha \lambda x.t'} \text{ (\xi)} \end{array}$$

Additionally to 2.1.5 we introduce the notion of α -equivalence that is a special congruence.

Definition 2.1.8 (α -Equivalence cf.[3]). α -equivalence is the smallest congruence relation $=_\alpha$ on the set of λ -terms such that for all terms t and all individual variables γ , the rules for equivalence and congruence and

$$\frac{\gamma \notin t}{\lambda x.t =_\alpha \lambda \gamma.(t[\gamma/x])} \text{ (\alpha)}$$

are satisfied. We denote α -equivalence with $\lambda x.t =_\alpha \lambda \gamma.(t[\gamma/x])$.

Within the λ -calculus β -reductions are the most important concept used for function application. The main idea of a β -reduction is to apply arguments to a λ -abstraction in order to evaluate it. This is done via substitution. We will later state that we can apply a special kind of reduction in order to obtain a normal form. A normal form is a most reduced form of a reducible expression that can no further be β -reduced. Not every term is reducible to a normal form at all. We will study this after definition 2.1.11 in more detail.

Definition 2.1.9 (β -Reduction). Let $s, t \in \Lambda$ and x be a variable, then a single-step β -reduction is the smallest relation \rightarrow_β on the set of λ -terms, that satisfies the following rules

$$\begin{array}{c} \frac{}{(\lambda x.t) s \rightarrow_\beta t[s/x]} (\beta) \\ \frac{s \rightarrow_\beta s'}{s t \rightarrow_\beta s' t} (cong_1) \\ \frac{t \rightarrow_\beta t'}{s t \rightarrow_\beta s t'} (cong_2) \\ \frac{s \rightarrow_\beta s'}{\lambda x.s \rightarrow_\beta \lambda x.s'} (\xi) \end{array}$$

Remark 2.1.4. cf. [3] Let $f, g \in \Lambda$. We write $f \twoheadrightarrow_\beta g$ if f β -reduces to g in finitely many steps, i.e. $f \rightarrow_\beta \cdots \rightarrow_\beta g$ is the long form of $f \twoheadrightarrow_\beta g$. In particular, \twoheadrightarrow_β denotes the reflexive transitive closure of \rightarrow_β .

Reducibility is defined as a one-way relation. In general, it is not possible that if $f \rightarrow_\beta g$ then $g \rightarrow_\beta f$. By allowing both normal reduction steps and inverse reduction steps $f \rightarrow_\beta g$ or $g \rightarrow_\beta f$ we make the relation symmetric. The reflexive symmetric transitive closure, is called β -equivalence and denoted with $=_\beta$. This means that we write $s =_\beta t$ if s can be transformed into t if by finitely many reduction steps and/or inverse reduction steps.

For example $(\lambda x.(\lambda y.y x)) a b \twoheadrightarrow_\beta b a$ but $b a \not\rightarrow_\beta (\lambda x.(\lambda y.y x)) a b$. Since $(\lambda x.(\lambda y.y x)) a b \twoheadrightarrow_\beta b a$ also $(\lambda x.(\lambda y.y x)) a b =_\beta b a$.

Example 2.1.5. We reduce the following term via β –reductions

$$\begin{aligned}
 & (\lambda x.(\lambda y.y \ x) \ y) \ ((\lambda x.x) \ y) \\
 \rightarrow_{\beta} & (\lambda x.(\lambda y.y \ x) \ y) \ (x[y/x]) \\
 = & (\lambda x.(\lambda y.y \ x) \ y) \ y \\
 \rightarrow_{\beta} & ((\lambda y.y \ x) \ y)[y/x] \\
 \rightarrow_{\alpha} & ((\lambda z.z \ x) \ y)[y/x] \\
 = & ((\lambda z.z \ x)[y/x] \ y[y/x]) \\
 = & ((\lambda z.z[y/x] \ x[y/x]) \ y) \\
 = & ((\lambda z.z \ y) \ y) \\
 \rightarrow_{\beta} & (z[y/z] \ y[y/z]) \\
 = & (y \ y)
 \end{aligned}$$

We have already introduced the principle of extensionality at the beginning of this chapter. It says that two objects are equal if their observed properties are the same. This logical principle can be applied on functions. If two functions are applied on the same arguments and they return the same result for every argument, they are equal. This second idea motivates η –reduction. That we add η –reduction does not mean that we have extensionality in general. If we want to have extensionality we have to add the axiom of extensionality to the untyped λ –calculus.

Definition 2.1.10 (η –Reduction). Let x be a bound variable in t or a variable that does not occur in t , then a single step η –reduction is the smallest relation \rightarrow_{η} satisfying

$$\begin{aligned}
 & \overline{\lambda x.t \ x \rightarrow_{\eta} t} \ (\eta) \\
 & \frac{s \rightarrow_{\beta} s'}{s \ t \rightarrow_{\eta} s' \ t} \ (cong_1) \\
 & \frac{t \rightarrow_{\beta} t'}{s \ t \rightarrow_{\eta} s \ t'} \ (cong_2) \\
 & \frac{s \rightarrow_{\beta} s'}{\lambda x.s \rightarrow_{\eta} \lambda x.s'} \ (\xi)
 \end{aligned}$$

Example 2.1.6. Consider $\lambda x.(\lambda y.y) x \rightarrow_\eta \lambda y.y$.

We will have a look on a specific application. Assume there is an arbitrary but fixed term 2, and we apply it to the abstraction $(\lambda x.(\lambda y.y) x)$ than it is the same as applying to the abstraction $(\lambda y.y)$, and it is also the same as simply stating 2 (as a nullary abstraction). Indeed,

by $(\lambda x.(\lambda y.y) x) 2 \rightarrow_\beta (\lambda y.y) x[2/x] = (\lambda y.y) 2$ we get the same result as if we simply write the application $(\lambda y.y) 2$.

Remark 2.1.7. cf.[3] A single-step $\beta\eta$ -reduction is defined as either a single-step β -reduction or a single-step η -reduction. Multi-step η -reduction is denoted with \rightarrow_η , multi-step $\beta\eta$ -reduction by $\rightarrow_{\beta\eta}$. $=_\eta$ -equivalence and $=_\beta$ η -equivalence are defined as for β -reduction.

Example 2.1.8. It is possible that for terms s, t , $s \not\rightarrow_\beta t$, but $s \rightarrow_{\beta\eta} t$:

$\lambda x.(\lambda y.(\lambda x.x)y) x \not\rightarrow_\beta \lambda y.y$, but $\lambda x.(\lambda y.(\lambda x.x)y) x \rightarrow_{\beta\eta} \lambda y.y$.

A normal form is a most reduced form of a reducible expression that can no further be β -reduced. In general, it is not possible to evaluate a λ -term to its most reduced form - the normal form.

Definition 2.1.11 (Normal Form). A term s is in β (η or $\beta\eta$) normal form (NF) iff there is no term t such that $s \rightarrow t$. We call a reduction to a normal form a normal order reduction.

β -normal forms have a special form.

Lemma 2.1.9. A term is in NF if and only if it is an abstraction $\lambda x.s$, where $s \in NF$ or it is of the form $x s_0 \dots s_n$ with $n \geq 0$ and all $s_i \in NF$.

Proof. For the backwards direction, we remark that terms of the two forms are indeed in NF. Conversely, we proceed via induction on the complexity of a term. If x is a variable we are done.

For the abstraction, consider a term $\lambda x.s$ that is in NF. By the definition of a normal form, also s is not reducible, otherwise proceed via a reduction, contradicting our assumption. This implies that s was already in NF. Indeed, $\lambda x.s$ is of the first

form.

For the application $s t$ with some terms s, t , assume that $s t$ is in NF. s and t are both in normal form, otherwise perform a reduction. By IH they are of the two forms mentioned in the consequent. In the case that s is an abstraction, also the antecedent already failed. Consequently, for $s := \lambda x.u$ with a term u in NF, indeed $s t$ is not of the first or the second form. It is reducible. No matter if t is of the first or the second form by IH. This covers the two cases where the first term is an abstraction.

Now, assume that s is of the second form. In particular, this means that $s := x s_0 \dots s_n$. By IH it is either of the first or the second form. It does not play a role if it is of the first or the second form. In any case we conclude that $s t$ is indeed of the second form $s := x s_0 \dots s_n t$, since t is already in NF by the antecedent. \square

Example 2.1.10. The combinator $\Omega = \omega \omega$ where $\omega = \lambda x.x x$ is the self application combinator. This means $\Omega = (\lambda x.x x) (\lambda x.x x) \rightarrow_\beta (\lambda x.x x) (\lambda x.x x) \rightarrow_\beta \Omega$. It has no normal form.

Theorem 2.1.11 (Church-Rosser-Theorem (CR) (1936) cf.[3]). *Let \twoheadrightarrow denote \twoheadrightarrow_β or \twoheadrightarrow_η . If s, t, u are λ terms with $s \twoheadrightarrow t$ and $s \twoheadrightarrow u$, then there is a λ term z such that $t \twoheadrightarrow z$ and $u \twoheadrightarrow z$. We call this the Church-Rosser property or confluence.*

For a detailed proof by Tait and L  f consider for example Selinger [3], whom we also follow in the following corollaries.

Theorem 2.1.12 (Weak Church-Rosser-Theorem (WCR) cf. [4]). *If $s \rightarrow t$ and $s \rightarrow u$ for some $s, t, u \in \Lambda$, then there exists a $z \in \Lambda$ with $t \twoheadrightarrow z$ and $u \twoheadrightarrow z$.*

For a different approach for the proofs of the WCR and CR one might consider S  rensen and Urzyczyn [4].

Corollary 2.1.13. *If $s =_\beta t$ then there exists a z with $s, t \twoheadrightarrow_\beta z$, and similarly for $\beta\eta$.*

The following corollary states, that if a normal form of a λ -term exists, then there is a normal order reduction that leads to it.

Corollary 2.1.14. *Given a β -normal form n and $n =_\beta t$, then $t \rightarrow_\beta n$, and similarly for $\beta\eta$.*

By the former corollary and the definition of a normal form we obtain the following result.

Corollary 2.1.15. *Given the β -normal forms m, n with $m =_\beta n$, then $m =_\alpha n$, and similarly for $\beta\eta$.*

This means, if a normal form exists it is unique up to α -equivalence.

Corollary 2.1.16. *If $s =_\beta t$, then neither or both have a β -normal form, and similarly for $\beta\eta$.*

Proof. Let $s =_\beta t$. Assume wlog the case that t has a normal form n and s does not, then $t \rightarrow_\beta n$ and $t =_\beta n$. By $s =_\beta t =_\beta n$, also $s =_\beta n$. Finally, by corollary 2.1.14, $s \rightarrow_\beta n$. We obtained that both have a normal form what contradicts our assumption. Therefore, this case and its symmetric case do not occur and the other cases follow. \square

Definition 2.1.12 (Normalizing cf. [4]). A term s is normalizing ($s \in WN_\beta$) if there exists a reduction sequence starting in s that ends with a normal form. In this case s has a normal form.

Definition 2.1.13 (Strongly Normalizing cf. [4]). A term s is strongly normalizing ($s \in SN_\beta$) if all reduction sequences starting in s are finite.

In the case that $s \notin SN_\beta$, we write $s \in \infty_\beta$.

Every *strongly normalizing* term is *normalizing*. But clearly the backwards direction does not hold, since there can be infinite reduction sequences in addition to the finite one. As expected, problems such as determining whether a given term has a normal form or if a given term is strongly normalizing are semidecidable.

Example 2.1.17. $(\lambda z.x) \Omega \rightarrow_\beta x$ is finite and ends in a normal form while $(\lambda z.x) \Omega \rightarrow_\beta \dots \rightarrow_\beta (\lambda z.x) \Omega$ is infinite if we always reduce Ω . We see $(\lambda z.x) \Omega$ is normalizing but not strongly normalizing.

From the computer scientific point of view we would like to have some kind of algorithm that ensures that when reducing we end in a normal form if one exists.

Definition 2.1.14 (Leftmost Outermost Reduction cf. [4]). If for a term s a normal form n exists, then there is a leftmost outermost reduction that leads to it, $s \xrightarrow{\beta} n$. We denote leftmost outermost reductions with $\xrightarrow{\beta}$.

Proof. For a proof one might consider Sørensen and Urzyczyn [4], Thm. 1.5.8. \square

In the last example we have seen in the first reduction an example for leftmost reduction.

Untyped λ -calculus and Computability

There is also the concept of λ -definability. As proven for example by Turing in [5] the concept is equivalent to Turing computability or the equivalent μ -recursivity. This implies that the investigation of this λ -calculus is one of the equivalent means to study computability which is in general important to understand which functions can be computed by an algorithm.

Firstly, we will define Church numerals. Church numerals are natural numbers encoded in the λ -calculus. We see that they are defined in a very natural way: relating to set theoretic definitions of the natural numbers using the \emptyset , which is itself closely related to the concept of induction using the iterative application of modus ponens.

Definition 2.1.15 (Church Numerals cf. for example [4]). We encode the natural number $n \in \mathbb{N}_0$ with the Church numeral $c_n := \lambda f x. f^n(x)$.

For example, $c_0 = \lambda f x. x$ and $c_3 = \lambda f x. f (f (f (x)))$.

Definition 2.1.16 (λ -Definability cf. [4]). A (partial) function $f : \mathbb{N}^k \rightarrow \mathbb{N} \cup \perp$ is λ -definable iff there is a $F \in \Lambda$, with the following properties:

- (i) If $f(n_1, \dots, n_k) = m$, then $Fx_{n_1}, \dots, s_{n_k} =_{\beta} c_m$.
- (ii) If $f(n_1, \dots, n_k) = \perp$, then Fx_{n_1}, \dots, s_{n_k} has no normal form.

Some basic functions in the language of the untyped λ -calculus can be given with the following. Of course, we have to verify for each given input that it reduces to the right term. Normally, we have to prove the base cases of some recursively defined function and the general cases, using induction. We use the definition in Sørensen and Urzyczyn [4] but there are other ways to define some of these functions.

$$\begin{aligned} \text{true} &:= \lambda xy.x \\ \text{false} &:= \lambda xy.y \\ \text{if } p \text{ then } q \text{ else } r &:= p \ q \ r \end{aligned}$$

such that

$$\begin{aligned} \text{if } \text{true} \text{ then } q \text{ else } r &\rightarrow_{\beta} q \\ \text{if } \text{false} \text{ then } q \text{ else } r &\rightarrow_{\beta} r \end{aligned}$$

A pair and the projections can be defined in the untyped λ -calculus as a λ -expression. But they cannot be defined in the simply typed λ -calculus in a similar way, because the corresponding type uses a \wedge . This cannot be defined in the implicational fragment of propositional logic.

$$\begin{aligned} \langle s, t \rangle &:= \lambda x.x \ s \ t \\ \pi_1 &:= \lambda p.p(\lambda xy.x) \\ \pi_2 &:= \lambda p.p(\lambda xy.y) \end{aligned}$$

where

$$\pi_i \langle s_1, s_2 \rangle \rightarrow_{\beta} s_i$$

Here are some functions that we can also define.

$$succ := \lambda nfx.f(nfx)$$

$$add := \lambda mnfx.mf(nfx)$$

$$mult := \lambda mnfx.m(nf)x$$

$$\pi_i^k := \lambda m_1 \dots m_k.m_i$$

These functions are expecting church numerals as an input.

Since we know that λ -definability is equivalent to Turing computability this concept is as well a useful tool in the investigation of computability questions.

2.2 Intuitionistic Propositional Logic

It is important to introduce the notion of intuitionistic logic before we consider typed λ -calculi. The logical reasoning within these λ -calculi is embedded within different fragments and higher order intuitionistic logic.

Why would we like to have intuitionistic logic and not some kind of classical logic? The type system that we would like to use should be constructive and should fit to the nature of our combinatory terms. Given a provably true logical formula, we should be able to construct a term that has this specific type – based on the proof tree of that formula. This is necessary for executing programs, since we need to be able to obtain objects of the types we are dealing with. If we have a rule for a proof by contradiction in our proof system, then what should the corresponding term in the simply typed λ -calculus be? A proof by contradiction is based on assigning truth values and on an absolute notion of truth. Syntactically constructing a term or a program is purely constructive and has no relation to absolute truth or the principle of the excluded middle. If we are able to show that there is no construction for the inverse type of a term, than this does not give us a construction for the type itself nor to a corresponding inverse term of the assumed one and its type. Consequently we need a logic that is purely constructive and does not allow a proof by contradiction. This approach can be found in intuitionistic logic. We give a short introduction, where we follow Sørensen and Urzyczyn [4].

When giving mathematical proofs we use classical logic. Since we are used to classical logic, intuitionistic logic seems very unintuitive to us at first sight. In classical logic $p \vee \neg p$ holds in any case. We do not need to know what p means. Truth is absolute. We can imagine giving a proof like: If p is true, then the first disjunct is true, otherwise the other is. By *tertium non datur* this is a valid proof. But it is not a constructive proof. It might not be decidable if p is true or not. Consider for example the question if a given term has a normal form. This is undecidable. Once we give a finite reduction sequence that ends in a normal form we can say yes. But we will not know if we will ever stop when doing leftmost reductions.

Intuitionistic logic is a constructive approach, where truth is not absolute and the principle of the excluded middle does not apply. A logical judgement is only true if

we can verify its correctness. This approach makes it more attractive to computer scientists. But with a classical background it might seem difficult to understand it intuitively. We will see that the symmetry of classical logic disappears. For example, we can give a counterexample for one direction of one of DeMorgan's laws: $\neg(\varphi \wedge \psi) \rightarrow \neg\varphi \vee \neg\psi$. This is clearly valid in propositional logic, but not in intuitionistic propositional logic. Conversely, there is a constructive proof for $\neg(\varphi \vee \psi) \rightarrow \neg\varphi \wedge \neg\psi$.

After introducing natural deduction systems, we will give this proof. Other classical propositional tautologies like $p \rightarrow \neg\neg p = p \rightarrow ((p \rightarrow \perp) \rightarrow \perp)$ and $\neg\neg p \rightarrow p$ are also not symmetric.

This means judging a logical statement is based on giving an explicit proof for the statement and not on assigning truth values to it.

Language, deductive system, and semantics characterize a logical calculus. Its language is defined by its vocabulary and its grammar. Sometimes language and the deductive system are called its syntax. To characterize a logic it is possible to combine these elements as follows: language & deductive system, language & semantics, language & deductive system & semantics. In terms of semantics there are a view ways to do so in logic. We have to give a definition what truth means in an interpretation. In terms of propositional logic this is a valuation function. Truth in an interpretation may be defined in different ways. There are algebraic approaches to define truth in an valuation. For example, in first order logic there are two standard methods to deal with quantifiers. One is the β -variant method, the other one is Tarski's method. In the latter we define an analogue for truth for formulae with free variables (satisfaction) and then define truth in terms of it.

We will define the language of propositional intuitionistic logic, natural deduction as a deductive system, and give it a Kripke semantics.

The language of intuitionistic propositional logic (or intuitionistic propositional calculus) is the language of propositional logic \mathcal{L}_{PL} . Consequently, it's vocabulary is the one of \mathcal{L}_{PL} , and we have the following formation rules in terms of expressions. This means that not any string that might be composed from the vocabulary of \mathcal{L}_{PL} might be a well-formed sentence. We define these sentences as follows.

Definition 2.2.1 (Formulae). Formulae are recursively defined as follows

- (i) Every propositional variable is a formula.
- (ii) \perp is a formula
- (iii) If φ and ψ are formulae, then $\varphi \rightarrow \psi$, $\varphi \wedge \psi$ and $\varphi \vee \psi$ are formulae.

We use the following abbreviations $\neg\varphi := \varphi \rightarrow \perp$, $\varphi \leftrightarrow \psi := (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$, and $\top := \perp \rightarrow \perp$. Implication is right associative, negation has the highest priority, and implication the lowest.

Since judging a logical statement is based on giving an explicit proof for the statement, the interpretation of compound formulae should be given by their construction and not by truth tables. We will introduce the Brouwer-Heyting-Kolmogorov (BHK) interpretation. This interpretation defines the meaning of compound formulae recursively by their construction. It is an important foundation in the understanding of λ -calculi and hence combinatory logic but also in the understanding of intuitionistic logic. Throughout type theories we find ideas that relate to the BHK interpretation.

Definition 2.2.2. BHK Interpretation

- (i) A construction of $\varphi_0 \wedge \varphi_1$ consists of a construction of φ_0 and a construction of φ_1
- (ii) A construction of $\varphi_0 \vee \varphi_1$ consists of an indicator $i \in \{0, 1\}$ and a construction of φ_i
- (iii) A construction of $\varphi_0 \rightarrow \varphi_1$ is a function (method) that transforms a construction of φ_0 in a construction of φ_1
- (iv) There is no construction of \perp .

According to this definition every construction of $\neg\varphi$ is a method that turns every construction of φ to an object that we cannot construct, i.e. an object that does not exist. If we consider the BHK interpretation, then there is no gap between logic and set theory. We will later talk about this correspondence when explaining the rules of the extended Curry-Howard isomorphism in more detail.

Natural deduction systems can be defined for various logics. For a detailed introduction to the rules of inference of natural deduction in classical logic one may consider for example [6].

Definition 2.2.3 (Deduction). A deduction of a conclusion φ from a set of premises Γ is a finite sequence of sentences in which every member is either a member of Γ , an axiom or a rule of inference, ending in φ .

Definition 2.2.4 (Deducibility). ϕ is deducible from the set of premisses Γ if and only if there exists a deduction of φ from Γ . We denote this with $\Gamma \vdash \varphi$ and call it a *judgement*.

In the case of intuitionistic propositional logic we have one axiom

Definition 2.2.5 (Axiom). $\Gamma, \varphi \vdash \varphi$ (Ax)

Definition 2.2.6 (Proof or derivation of φ). A *proof* or *derivation* of $\Gamma \vdash \varphi$ is a finite sequence (tree) of judgements that satisfy the following conditions:

- (i) The root (last line) is $\Gamma \vdash \varphi$
- (ii) The leaves (lines above) are axioms.
- (iii) The line of the mother node (preceding line) is obtained by one of the following rules of inference by the line of the child node (current line).

If $\Gamma = \emptyset$ and there is a deduction $\emptyset \vdash \varphi$ then we call φ a theorem and denote it with $\vdash \varphi$. If such a proof (derivation) exists, we say the judgement $\Gamma \vdash \varphi$ is provable (derivable) and write $\Gamma \vdash \varphi$.

What we call in the context of natural deduction a proof, is what we initially called a construction.

We introduce two different notations for the rules of inference. Gentzen's notation is the one that is introduced at first. It is space saving and faster to use. The secondly introduced notation is more appropriate in higher type theory and is commonly used within this field. It does not really matter in the simply typed λ -calculus whether we use Gentzen's style or the other. In the calculus of constructions λC for example it is better if we use the second notation. The reason for this is that the order of the hypotheses is important when we are working with types that depend on other types. The reason is this dependency. We denote the set of premises with Γ , by convention. The rules for intuitionistic propositional logic are given by the following

Definition 2.2.7 (Rules of Natural Deduction). Rule of Conjunction Introduction:

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} (\wedge I)$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} (\wedge I)$$

Rule of Conjunction Elimination:

$$\frac{\varphi \wedge \psi}{\varphi} (\wedge E) \quad \frac{\varphi \wedge \psi}{\psi} (\wedge E)$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} (\wedge E) \quad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} (\wedge E)$$

Rule of Conditional Introduction:

$$\frac{\begin{array}{c} [\varphi] \\ \vdots \\ \psi \end{array}}{\varphi \rightarrow \psi} (\rightarrow I)$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi} (\rightarrow I)$$

An assumption in square brackets is discharged by this rule.

Rule of Conditional Elimination:

$$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi} (\rightarrow E)$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \varphi \rightarrow \psi}{\Gamma \vdash \psi} (\rightarrow E)$$

This rule is also known as modus ponens.

Ex Falso Quodlibet:

$$\frac{\perp}{\varphi} (\perp E)$$

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash \varphi} (\perp E)$$

Rules of Disjunction Introduction:

$$\frac{\varphi}{\varphi \vee \psi} (\vee I) \quad \frac{\psi}{\varphi \vee \psi} (\vee I)$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} (\vee I) \quad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} (\vee I)$$

Rule of Disjunction Elimination:

$$\frac{\varphi \vee \psi \quad \begin{array}{c} [\varphi] \\ \vdots \\ \gamma \end{array} \quad \begin{array}{c} [\psi] \\ \vdots \\ \gamma \end{array}}{\gamma} (\vee E)$$

$$\frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \gamma \quad \Gamma, \psi \vdash \gamma}{\Gamma \vdash \gamma} (\vee E)$$

The reader may have recognized that there is no rule for a proof by contradiction. Indeed, if we want to have a proof system for classical logic, then we might like to add the following rule

Reductio ad Absurdum:

$$\frac{\begin{array}{c} [\neg \varphi] \\ \vdots \\ \perp \end{array}}{\varphi} (RAA)$$

Example 2.2.1. We show that $(\varphi \rightarrow \psi) \rightarrow (\neg \psi \rightarrow \neg \varphi)$ is valid in intuitionistic logic. First, let us rewrite the formula, avoiding abbreviations $(\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \perp) \rightarrow (\varphi \rightarrow \perp))$.

$$\frac{\frac{\frac{[\psi \rightarrow \perp]_2 \quad \frac{[\varphi \rightarrow \psi]_1 \quad [\varphi]_3}{\psi} \rightarrow E}{\perp} \rightarrow I_3}{(\psi \rightarrow \perp) \rightarrow (\varphi \rightarrow \perp)} \rightarrow I_2}{(\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \perp) \rightarrow (\varphi \rightarrow \perp))} \rightarrow I_1$$

Let $\Gamma = \{(\varphi \rightarrow \psi), (\psi \rightarrow \perp), \varphi\}$, then in the other style we obtain

$$\frac{\frac{\frac{\Gamma \vdash \psi \rightarrow \perp}{\Gamma \vdash \psi} \rightarrow E \quad \frac{\Gamma \vdash \varphi \rightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \rightarrow E}{\Gamma \vdash \perp} \rightarrow I \quad \frac{(\varphi \rightarrow \psi), (\psi \rightarrow \perp) \vdash \varphi \rightarrow \perp}{(\varphi \rightarrow \psi) \vdash (\psi \rightarrow \perp) \rightarrow (\varphi \rightarrow \perp)} \rightarrow I}{\vdash (\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \perp) \rightarrow (\varphi \rightarrow \perp))} \rightarrow I$$

Example 2.2.2. Above we mentioned a proof of one of the directions of one of De Morgans' Laws. Now, we will give it by natural deduction.

$$\frac{\frac{(\varphi \vee \psi) \rightarrow \perp, \varphi \vdash (\varphi \vee \psi) \rightarrow \perp \quad \frac{(\varphi \vee \psi) \rightarrow \perp, \varphi \vdash \varphi}{\varphi \vdash \varphi \vee \psi} \vee I}{(\varphi \vee \psi) \rightarrow \perp, \varphi \vdash \perp} \rightarrow I \quad \frac{\frac{(\varphi \vee \psi) \rightarrow \perp, \psi \vdash (\varphi \vee \psi) \rightarrow \perp \quad \frac{(\varphi \vee \psi) \rightarrow \perp, \psi \vdash \psi}{\psi \vdash \varphi \vee \psi} \vee I}{(\varphi \vee \psi) \rightarrow \perp, \psi \vdash \perp} \rightarrow I}{((\varphi \vee \psi) \rightarrow \perp) \vdash (\varphi \rightarrow \perp) \wedge (\psi \rightarrow \perp)} \wedge I \quad \frac{((\varphi \vee \psi) \rightarrow \perp) \vdash (\varphi \rightarrow \perp) \wedge (\psi \rightarrow \perp)}{((\varphi \vee \psi) \rightarrow \perp) \rightarrow (\varphi \rightarrow \perp) \wedge (\psi \rightarrow \perp)} \rightarrow I$$

We will define Kripke semantics of intuitionistic propositional logic. This is a another constructive approach.

Definition 2.2.8 (Kripke Model). A *Kripke Model* is a triple $\mathcal{C} = \langle C, \leq, \Vdash \rangle$ where $C \neq \emptyset$, \leq is a partial order on C , and \Vdash is a binary relation between the elements of C and propositional variables.

The elements of C are called states or possible worlds. The forcing relation must satisfy monotonicity.

Definition 2.2.9 (Monotonicity). Assume $c, c' \in C$ and p is a propositional variable. If $c \leq c'$ and $c \Vdash p$ then $c' \Vdash p$.

The generalization of monotonicity for some formula φ can be proven by induction.

Definition 2.2.10. Let $\mathcal{C} = \langle C, \leq, \Vdash \rangle$ be a Kripke model and $c \in C$, then

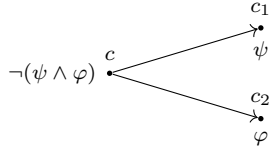
1. $c \Vdash \varphi \wedge \psi$ iff $c \Vdash \varphi$ and $c \Vdash \psi$
2. $c \Vdash \varphi \vee \psi$ iff $c \Vdash \varphi$ or $c \Vdash \psi$

3. $c \Vdash \varphi \rightarrow \psi$ iff for all $c' \in C$ with $c' \geq c$ if $c' \Vdash \varphi$ then $c' \Vdash \psi$
4. $c \Vdash \perp$ never holds

Remark 2.2.3. This implies that $c \Vdash \neg\varphi$ iff $c' \not\Vdash \varphi$, for all $c' \geq c$.

Example 2.2.4. We will now give a counterexample for one of DeMorgan's formulae that does not hold in propositional intuitionistic logic. We show that $\neg(\psi \wedge \varphi) \rightarrow \neg\psi \vee \neg\varphi$ is not valid in intuitionistic propositional logic.

Proof. Let $C = \{c, c_1, c_2\}$ with $c \leq c_1, c_2$ and c_1, c_2 incomparable. We give a Kripke model to show that $c \not\Vdash \neg(\psi \wedge \varphi) \rightarrow \neg\psi \vee \neg\varphi$. This is the case iff $\exists c' \geq c : c' \Vdash \neg(\psi \wedge \varphi)$ and $c' \not\Vdash \neg\psi \vee \neg\varphi$. Consequently, $c' \not\Vdash \neg\psi \vee \neg\varphi$ iff $c' \not\Vdash \neg\psi$ and $c' \not\Vdash \neg\varphi$:



By *Remark 2.2.3* $c \not\Vdash \neg\psi$ iff $\exists c' \geq c$ with $c' \Vdash \psi$. Since $c_1 \Vdash \psi$, $c \not\Vdash \neg\psi$ and $c_2 \Vdash \varphi$ implies $c \not\Vdash \neg\varphi$. Hence, by $c \not\Vdash \neg\psi$ and $c \not\Vdash \neg\varphi$ also $c \not\Vdash \neg\psi \vee \neg\varphi$ implying the existence of such a state c' . Since no state forces $\psi \wedge \varphi$, by the semantics of the implication also no state forces \perp . \square

2.3 The Simply Typed Lambda Calculus λ_{\rightarrow}

The simply typed lambda calculus (λ_{\rightarrow}) is the basis for type theories that are based on the λ -calculus. Type theory is the study of type systems that can be used as a foundation for the development of programming languages. Type theories enrich λ -calculi with types that are not paradoxical. Given the term M , $M M$ cannot be given a type. We end up in a version of Russel's paradox since it is not possible to interpret M both as an argument and as a function.

We are interested in the termination of programs that can be guaranteed by the programming language. Therefore, we are interested in the so called Church-Rosser property (confluence) and (strong) normalization in λ -calculi. Strong

normalization (every reduction sequence is finite) and normalization (there is a reduction sequence ending in a normal form) gives us existence, and together with the Church Rosser property the uniqueness of a normal form (no further reducible form). Finally, we introduce the subject reduction property (a reduced term has the same type as the initial one), and with it we obtain the termination of a typed term in a normal form of the same type.

The Curry-Howard isomorphism shows a correspondence between logic (like the propositional intuitionistic logic that we introduced) and (computational) λ -calculi forming the *propositions as types* or *proofs-as-programs* paradigm. We need the syntactic tradition of proof theory, not the semantic approach of classical logic. In particular, for a rewriting system it is more important to study the proof structure, and give rules how to construct it. This means that the constructive approach of intuitionistic logic where a proof (construction) is the final criterion on validity is of greater importance than the semantics. This implies that the BHK interpretation is also an important foundation in the understanding of λ -systems and their relation to logic. The simply typed λ -calculus basically relates to the implicational fragment of intuitionistic logic but can be extended as we will see. Finally, we obtain a correspondence between logic, λ -calculus and category theory. We follow Sørensen and Urzyczyn [4].

Definition 2.3.1 (Simple Types). Simple Types are formulae in the implicational fragment of propositional intuitionistic logic (formulae only containing \rightarrow). We define them as follows:

- (i) Every propositional variable is a simple type
- (ii) If σ, τ are simple types, then so is $\sigma \rightarrow \tau$.

Nothing else is a simple type.

Example 2.3.1. $p \rightarrow q \rightarrow r$ is a simple type for the propositional variables p, q, r . By convention implication is right associative. This means $p \rightarrow q \rightarrow r = p \rightarrow (q \rightarrow r)$.

We will introduce the modern Church style simply typed λ -calculus. In the orthodox Church Style all type information is part of the term. In the modern

Church style we use contexts to declare the types of the free variables as we do in a Curry style system. But in contrast the types of bound variables remain in the syntax.

Definition 2.3.2 (Raw Terms). We define the set of raw terms recursively as follows.

- (i) Every individual variable x, y, z, \dots is a raw term.
- (ii) If M, N are raw terms then the application $(M \ N)$ is a raw term.
- (iii) If M is a raw term, x is an individual variable, and σ is a simple type then $(\lambda x : \sigma. M)$ is a raw term. Nothing else is a raw term.

Definition 2.3.3. Let M be a raw term, x an individual variable, and σ be simple type.

We call an expression of the form $M : \sigma$ a *statement*.

A statement of the form $x : \sigma$ is called a *declaration*.

A finite set of declarations is called a *context*. We usually denote it with Γ .

We call an expression of the form $\Gamma \vdash M : \sigma$ a *judgement* and will say „ M is of type σ in context Γ “.

For a given type σ , if there is a term of type M with $M : \sigma$ such that $\Gamma \vdash M : \sigma$ then M is called an „*inhabitant* of σ in context Γ “. If $\vdash M : \sigma$, we say that σ is „non-empty“ and has the *inhabitant* M – in this case M is closed.

Definition 2.3.4 (Free variables). Free variables (FV) of a raw term M are

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x : \sigma. N) &= FV(N) \setminus \{x\} \\ FV(N \ P) &= FV(N) \cup FV(P) \end{aligned}$$

A term is again closed if it does not contain free variables.

Variables are again considered bound if a λ -abstraction binds them. In $\lambda x : \sigma. N$, x is bound.

When we talk about a term we will mean its α -equivalence class. The terms $\lambda x : \sigma. N$ and $\lambda y : \sigma. N[y/x]$ are in the same class.

Definition 2.3.5 (Substitution). For raw terms M, N, P, Q and a variable x we define a substitution $M[N/x]$ as follows

$$\begin{aligned} x[N/x] &= N \\ y[N/x] &= y, \text{ if } x \neq y \\ (P \ Q)[N/x] &= P[N/x] \ Q[N/x] \\ (\lambda y : \sigma.P)[N/x] &= \lambda y : \sigma.P[N/x], \text{ with } x \neq y \text{ and } y \notin FV(N) \end{aligned}$$

Again we will define β -reduction, basically in the same way as we did in the untyped λ -calculus. The rules defining β -reduction are captured by the notion of a compatible relation. Introducing it once, we save space in the definitions.

Definition 2.3.6 (Compatible Relation). A relation \succ on raw terms is compatible iff it satisfies the following

- (i) If $M \succ N$ then $\lambda x.M \succ \lambda x.N$.
- (ii) If $M \succ N$ then $M \ P \succ N \ P$.
- (iii) If $M \succ N$ then $P \ M \succ P \ N$.

for raw terms M, N, P , and some variable x .

Definition 2.3.7 (β -Reduction). We define the relation \rightarrow_β as the smallest compatible relation on raw terms, such that $(\lambda x : \sigma.P) \ Q \rightarrow_\beta P[Q/x]$.

We use the notation \rightarrow_β and $=_\beta$ as in the untyped λ -calculus.

Definition 2.3.8 (η -reduction). η -reduction is the smallest compatible relation on raw terms \rightarrow_η such that $\lambda x : \sigma.M \ x \rightarrow_\eta M$, if $x \notin FV(M)$.

We will consider the simply typed λ -calculus as a deductive system where we can derive judgements with the following rules.

Definition 2.3.9.

$$\begin{array}{c} \Gamma, x : \sigma \vdash x : \sigma \text{ (Var)} \\ \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} \text{ (Abst)} \\ \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \text{ (Appl)} \end{array}$$

We see that these rules directly correspond to the rules *axiom*, *implication introduction* and *implication elimination of propositional intuitionistic logic*.

Definition 2.3.10 (Term). For a raw term M and a context Γ , we say M is a *term* in context Γ , if there is a simple type σ such that $\Gamma \vdash M : \sigma$ is derivable.

From a computer scientific perspective the deductive system that corresponds to the simply typed λ -calculus can be understood as part of a compiler. Conversely β -reduction corresponds to the execution of a program.

Definition 2.3.11 (Typable). A term $M \in \Lambda$ is typable if there is a context Γ and a simple type σ such that $\Gamma \vdash M : \sigma$.

Some terms of the untyped λ -calculus are not typeable. This means that we cannot derive a corresponding type in a proof tree. For example, we cannot find a type for $x x$, $x y x$ or ω .

Indeed by definition of a term (Definition 2.3.10), a term in the modern Church style simply typed λ -calculus is correctly typed and therefore typeable. So a term comes always with a type. A type of a bound variable is embedded in the term and a type of a free variable has to be declared in the context.

β reductions should be innocuous to the type assigned to a term. If a term is well-typed it stays well-typed no matter if we reduce it.

Theorem 2.3.2 (Subject Reduction Theorem cf. [4]). *If $M \rightarrow_\beta N$ and $\Gamma \vdash M : \sigma$ then $\Gamma \vdash N : \sigma$.*

Proof. Induction on the complexity of M with respect to the definition of \rightarrow_β and some lemmas proven in Sørensen and Urzyczyn [4]. \square

Example 2.3.3. Let $M = (\lambda f : p \rightarrow q. \lambda x : p. f x) g$, $g : p \rightarrow q \in \Gamma$ and $\Gamma \vdash (f : p \rightarrow q. \lambda x : p. f x) g : p \rightarrow q$. Particularly, $M \rightarrow_{\beta} N$ with $N = f x [g/f] = g x$. Then $\Gamma, f : p \rightarrow q \vdash \lambda x : p. f x : p \rightarrow q$ and $\Gamma \vdash g : p \rightarrow q$ by the rules. And thus $\Gamma \vdash \lambda x : p. f x [g/f] : p \rightarrow q$, and $\Gamma \vdash \lambda x : p. g x : p \rightarrow q$.

In the modern Church style a term is a term by definition if all its free variables have a declared type in a given context Γ . The rest of the type information is built into the expression by directly assigning a type to the bound variables. This way the type of the bound variables comes together with the syntax. The type of a term is unique.

Proposition 2.3.4 (cf. [4] 3.4.1). *If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$ then $\sigma = \tau$.*

Proof. We proceed via induction on the complexity of a term with respect to the lines in a proof.

If $\Gamma \vdash x : \sigma$ and $\Gamma \vdash x : \tau$, $x : \tau \in (\Gamma)$, and the last line is

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} (Var)$$

, then $\sigma = \tau$, otherwise by the analogue argumentation $\sigma = \tau$.

If $\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau$, and $\Gamma \vdash \lambda x : \delta. M : \delta \rightarrow \gamma$ then the last lines are

$$\frac{\Gamma, x : \delta \vdash M : \gamma}{\Gamma \vdash \lambda x : \delta. M : \delta \rightarrow \gamma} (Abst) \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} (Abst)$$

γ, τ are both derivable. Hence, by IH $\gamma = \tau$, implying $\delta = \sigma$, by the definition of $(Abst)$.

If $\Gamma \vdash M N : \tau$, and $\Gamma \vdash M N : \gamma$ then the last lines are

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M N : \tau} (App) \quad \frac{\Gamma \vdash M : \delta \rightarrow \gamma \quad \Gamma \vdash N : \delta}{\Gamma \vdash M N : \gamma} (App)$$

Since σ and δ are derivable, by IH $\sigma = \delta$. Indeed, $\Gamma \vdash M : \sigma \rightarrow \tau$ and $\Gamma \vdash M : \delta \rightarrow \gamma$, implying, by IH, $\delta \rightarrow \gamma = \sigma \rightarrow \tau$. By definition of $(Abst)$, $\tau = \gamma$. \square

We do not introduce the other convenient style (Curry Style). One essential difference is that we do not assign types to bound variables of a function. This

means instead of $\lambda x : \sigma. N : \sigma \rightarrow \tau$ we have $\lambda x. N : \sigma \rightarrow \tau$. This means that using Church we have deliver more information since we integrate parts of our corresponding logic into the λ -calculus. While the Church Style refers to a programming language where the programmer has to provide types for a function, in the Curry style we can think of an interpreter that does the type inference for the programmer. To some extent the systems can be transferred into each other (see Definition 3.4.3, and Propositions 3.3.4 – 3.4.5 in Sørensen and Urzyczyn [4]).

All simply typed terms have normal forms and are strongly normalizing. This means that it does not matter how we reduce terms, we will reach a normal form. This is a very important result, since a programming language based on the simply typed λ -calculus cannot infinitely loop.

Theorem 2.3.5 (Existence of Normal Forms cf. [4]). *All Church-style λ_{\rightarrow} terms have a normal form.*

Proof. Consider Sørensen and Urzyczyn [4], Thm. 3.5.1. □

Theorem 2.3.6 (Terms are strongly-normalizing cf. [4]). *If $\Gamma \vdash M : \sigma$, then M is strongly normalizing. Terms are strongly normalizing.*

Proof. See Thm. 3.5.5 in Sørensen and Urzyczyn [4]. □

Theorem 2.3.7 (Church-Rosser Property). *The Church style simply typed λ -calculus has the Church Rosser property: If M, N, P are terms with $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, then there is a λ -term Z such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.*

Proof. With Newman's Lemma it suffices to show that λ_{\rightarrow} has WCR, see Sørensen and Urzyczyn [4]. □

Remark 2.3.8. We have strong normalization and through normalization we obtained the existence of a normal form. Together with the Church Rosser property we get the uniqueness of a normal form:

Since every reduction sequence is finite by strong normalization, and there is a reduction sequence ending in a normal form Z by normalization, then every other

reduction sequence ending in some Z' has a common successor. But since Z is in normal form and Z' is not reducible it should be the case that $Z = Z'$.

Example 2.3.9. In the simply typed λ -calculus a la Church we can define a church numeral for a natural number n as $\bar{n} := \lambda f : p \rightarrow p. \lambda x : p. f^n(x)$ and we can show $\vdash \bar{n} : \mathbb{N}$ for $\mathbb{N} := (p \rightarrow p) \rightarrow (p \rightarrow p)$.

Proof.

$$\frac{\frac{\frac{}{f : p \rightarrow p, x : p \vdash x : p} (Var)}{f : p \rightarrow p \vdash \lambda x : p. x : p \rightarrow p} (Abst)}{\vdash \lambda f : p \rightarrow p. \lambda x : p. x : p \rightarrow (p \rightarrow p)} (Abst)$$

Assume for induction on the lines of the proof that $f^{n-1}(x) : p$, then

$$\frac{\frac{\frac{}{f : p \rightarrow p, x : p \vdash f : p \rightarrow p} (Var)}{f : p \rightarrow p, x : p \vdash f(f^{n-1}(x)) : p = f^n(x) : p} (IH)}{\frac{f : p \rightarrow p \vdash \lambda x : p. f^n(x) : p \rightarrow p}{} (Abst)} (App)$$

By induction $\vdash \bar{n} : \mathbb{N}$ follows. □

Definability in the simply typed λ -calculus is weaker than in the untyped λ -calculus since every term in λ_{\rightarrow} is strongly normalizing.

Definition 2.3.12 (λ_{\rightarrow} -Definability cf. [4]). Let $\mathbb{N} := (p \rightarrow p) \rightarrow (p \rightarrow p)$ for some individual type variable p . A function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is λ_{\rightarrow} -definable if there is a term F with $\vdash F : \mathbb{N}^k \rightarrow \mathbb{N}$, such that for all $n_0, \dots, n_k \in \mathbb{N}$.

$$F c_{n_0} \dots c_{n_k} =_{\beta} c_{f(n_0, \dots, n_k)}$$

Definition 2.3.13 (Extended Polynomials cf. [4]). The smallest class of functions over \mathbb{N} that is closed under composition and contains the constant functions 1 and

0, projections, addition, multiplication, and the conditional function

$$\text{cond}(n_1, n_2, n_3) = \begin{cases} n_2 & \text{if } n_1 = 0 \\ n_3 & \text{otherwise} \end{cases}$$

is called the class of *extended polynomials*.

Theorem 2.3.10 (Schwichtenberg cited according to [4]). *The λ_{\rightarrow} -definable functions are exactly the extended polynomials.*

Proof. First, we proof that the extended polynomials are λ_{\rightarrow} -definable.

The constant 0 function is definable by $\lambda x_1 : \mathbb{N}. \dots \lambda x_k : \mathbb{N}. c_0$. We showed in Example 2.3.9 that $c_n : \mathbb{N}$ for some $n \in \mathbb{N}$. Consequently, $\lambda x_1 : \mathbb{N}. \dots \lambda x_k : \mathbb{N}. c_0$ has type $\mathbb{N}^k \rightarrow \mathbb{N}$.

Clearly, $(\lambda x_1 : \mathbb{N}. \dots \lambda x_k : \mathbb{N}. c_0). c_{n_0} \dots c_{n_k} \rightarrow_{\beta} c_0$.

And we can derive

$$\frac{\frac{\text{see Example 2.3.9}}{x_1 : \mathbb{N}, \dots x_k : \mathbb{N}. \vdash c_0 : \mathbb{N}} \quad \vdots \text{ k-2 times (Abst)}}{x_1 : \mathbb{N} \vdash \lambda x_2 : \mathbb{N}. \dots \lambda x_k : \mathbb{N}. c_0 : \mathbb{N} \rightarrow (\mathbb{N}^{k-1} \rightarrow \mathbb{N})} \text{ (Abst)} \\ \vdash \lambda x_1 : \mathbb{N}. \dots \lambda x_k : \mathbb{N}. c_0 : \mathbb{N} \rightarrow (\mathbb{N}^{k-1} \rightarrow \mathbb{N})$$

The following statements have to be proven in similar manner. The constant 1 function is definable with $\lambda x_1 : \mathbb{N}. \dots \lambda x_k : \mathbb{N}. c_1$, the projections are defined by $\lambda x_1 : \mathbb{N}. \dots \lambda x_k : \mathbb{N}. x_i$,

addition by $\lambda m : \mathbb{N}. \lambda n : \mathbb{N}. \lambda f : p \rightarrow p. \lambda x : p. m \ f(nfx)$,

multiplication with $\lambda m : \mathbb{N}. \lambda n : \mathbb{N}. \lambda f : p \rightarrow p. \lambda x : p. m \ (nf)x$.

The conditional function can be defined by

$\lambda m : \mathbb{N}. \lambda n : \mathbb{N}. \lambda k : \mathbb{N}. \lambda f : p \rightarrow p. \lambda x : p. m(\lambda y : p. kfx)(nf x)$. Indeed

$$(\lambda m : \mathbb{N}. \lambda n : \mathbb{N}. \lambda k : \mathbb{N}. \lambda f : p \rightarrow p. \lambda x : p. m(\lambda y : p. kfx)(nf x)) \ c_0 \ c_{n_2} \ c_{n_3} \rightarrow_{\beta} c_{n_2}$$

$$(\lambda m : \mathbb{N}. \lambda n : \mathbb{N}. \lambda k : \mathbb{N}. \lambda f : p \rightarrow p. \lambda x : p. m(\lambda y : p. kfx)(nf x)) \ c_{n_1 \neq 0} \ c_{n_2} \ c_{n_3} \rightarrow_{\beta} c_{n_3}$$

For the composition assume that $g : \mathbb{N} \rightarrow \mathbb{N}$, $h : \mathbb{N}^k \rightarrow \mathbb{N}$ are extended polynomials such that $g \circ h$ is a composable pair and that they are λ_{\rightarrow} -defined by G , H

respectively. It is λ -definable with

$$\begin{aligned} &\vdash \lambda g : \mathbb{N} \rightarrow \mathbb{N}. \lambda h : \mathbb{N}^k \rightarrow \mathbb{N}. \lambda x_1 : \mathbb{N} \dots \lambda x_k : \mathbb{N}. g \ (h \ (x_1 \dots x_k)) : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N}^k \rightarrow \mathbb{N}) \rightarrow \mathbb{N}^k \text{ and} \\ &(\lambda g : \mathbb{N} \rightarrow \mathbb{N}. \lambda h : \mathbb{N}^k \rightarrow \mathbb{N}. \lambda x_1 : \mathbb{N} \dots \lambda x_k : \mathbb{N}. g \ (h \ (x_1 \dots x_k))) \ G \ H \ c_{n_1} \dots c_{n_k} \twoheadrightarrow_{\beta} \\ &G \ (H \ (c_{n_1} \dots c_{n_k})) = g \circ h(n_1 \dots n_k). \end{aligned}$$

Suppose that $g : \mathbb{N}^k \rightarrow \mathbb{N}, h_i : \mathbb{N} \rightarrow \mathbb{N}$ are extended polynomials such that $g \circ h$ is a composable pair and that they are λ_{\rightarrow} -defined by G, H respectively.

The other direction can be found in the solution section (exercises 3.25 and 3.26) of Sørensen and Urzyczyn [4]. \square

Remark 2.3.11. Considering the composite in a more informal way, we get the following. The composite of $\Gamma \vdash \lambda x : \varphi. M : \varphi \rightarrow \psi$ and $\Gamma \vdash \lambda y : \psi. N : \psi \rightarrow \rho$ is $\Gamma \vdash \lambda x : \varphi. N[M/y] : \varphi \rightarrow \rho$ for a fixed variable $y : \psi$, occurring freely in the second postcomposed abstraction.

Considering our proof above, G can be derived from $y : \mathbb{N} \vdash G'(x) : \mathbb{N}$ for some G' and H can be derived from $x_1 : \mathbb{N}, \dots, x_k : \mathbb{N} \vdash H'(x) : \mathbb{N}$ for some H' . This means that the composition is given by $x_1 : \mathbb{N}, \dots, x_k \vdash G'[H'/y] : \mathbb{N}$.

2.3.1 The Curry-Howard Isomorphism

We observe similarities between the natural deduction proof rules $\text{NJ}(\rightarrow)$ for the implicative fragment of intuitionistic propositional logic $\text{IPC}(\rightarrow)$ and the simply typed λ -calculus that are very natural.

λ_{\rightarrow}	$\text{NJ}(\rightarrow)$
$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} (Var)$	$\frac{}{\Gamma, \sigma \vdash \sigma} (Ax)$
$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} (Abst)$	$\frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \rightarrow \tau} (\rightarrow I)$
$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} (Appl)$	$\frac{\Gamma \vdash \sigma \rightarrow \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau} (\rightarrow E)$

In contrast to Curry and Howard's logical interpretation, Lambek interprets $p \rightarrow q$ as a set-theoretic function. Another adequate interpretation is that we interpret a type $p \rightarrow q$ as a programm specification. This means we specify a programm that takes a term of type p and returns a term of type q . The corresponding terms can be interpreted as programs meeting the programm specification.

Definition 2.3.14. For a type context Γ , $rg(\Gamma)$ denotes only the types of the variables of the context, conversely $dom(\Gamma)$ denotes only the variables of the context.

Proposition 2.3.12 (Curry Howard Isomorphism). *It holds that*

- (i) *If $\Gamma \vdash M : \sigma$ in λ_{\rightarrow} then $rg(\Gamma) \vdash \varphi$ in $IPC(\rightarrow)$.*
- (ii) *If $\Delta \vdash \varphi$ in $IPC(\rightarrow)$ then there are a term M and a context Γ such that $\Gamma \vdash M : \varphi$ in λ_{\rightarrow} and $rg(\Gamma) = \Delta$.*

Proof. We proceed via induction on the complexity of a term with respect to the lines of a proof.

Assume $\Gamma \vdash x : \sigma$, then the last line of the proof is

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} (Var)$$

Since $x : \sigma \in \Gamma$, indeed $\sigma \in rg(\Gamma)$. Consequently, by

$$\frac{}{rg(\Gamma), \sigma \vdash \sigma} (Ax)$$

also $rg(\Gamma) \vdash \sigma$.

Assume $\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau$. The last proof lines of the proof are

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} (Abst)$$

Since $x : \sigma \in \Gamma$, indeed $\sigma \in rg(\Gamma)$. Since $\Gamma, x : \sigma \vdash M : \tau$, by IH it follows that

$rg(\Gamma), \sigma \vdash \tau$ is derivable. This implies

$$\frac{rg(\Gamma), \sigma \vdash \tau}{rg(\Gamma) \vdash \sigma \rightarrow \tau} (\rightarrow I)$$

The third case can be proved in a similar way, and is left away for the sake of brevity.

Conversely, assume for the first case that $\Delta \vdash \sigma$ then

$$\frac{}{\Delta, \sigma \vdash \sigma} (Ax)$$

Then there is a term x , since there are denumerably infinitely many variables. We assign the variable x to type σ and assign distinct variables to the rest of Δ , constructing a context Γ with $x : \sigma \in \Gamma$ and $rg(\Gamma) = \Delta$. Consequently, by construction

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} (Var)$$

, implying $\Gamma \vdash x : \sigma$ is derivable.

For the second case assume that $\Delta \vdash \sigma \rightarrow \tau$. The last proof line is

$$\frac{\Delta, \sigma \vdash \tau}{\Delta \vdash \sigma \rightarrow \tau} (\rightarrow I)$$

Since $\Delta, \sigma \vdash \tau$ by IH there is a term $M : \tau$ in a context $\Gamma' = \Gamma, x : \sigma$ such that $\Gamma' \vdash M : \tau$ and $rg(\Gamma') = \Delta, \sigma$. This implies that $rg(\Gamma) = \Delta$ for some $\Gamma = \Gamma' - x : \sigma$. For the type σ we find an inhabitant in a variable x_i , for some variable x_i not occurring in $dom(\Gamma)$. Consequently,

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} (Abst)$$

, and with this $\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau$ is derivable.

Again, the third case can be performed in a similar manner. □

The second implication provides a method to create a term for the type we are proving in $IPC(\rightarrow)$. We decorate our proof tree using the rules for Abstraction

and Application. This can of course be done the other way around. We create a proof tree using λ_{\rightarrow} -rules and begin when finished to assign simple types to the terms.

Example 2.3.13. Given a type $(p \rightarrow q) \rightarrow p \rightarrow q$, we would like to find an inhabitant of this type. This means we have to find a term $M : (p \rightarrow q) \rightarrow p \rightarrow q$ such that $\vdash M : (p \rightarrow q) \rightarrow p \rightarrow q$

$$\frac{\frac{\frac{\overline{p \rightarrow q, p \vdash p \rightarrow q} (Ax) \quad \overline{p \rightarrow q, p \vdash p} (Ax)}{\overline{p \rightarrow q, p \vdash q} (\rightarrow I)} (\rightarrow E)}{\overline{p \rightarrow q \vdash p \rightarrow q} (\rightarrow I)} (\rightarrow I)$$

We start assigning names to the assumptions of (Ax) . This way we create the domains of the contexts.

$$\frac{\frac{\frac{\overline{f : p \rightarrow q, x : p \vdash f : p \rightarrow q} (Ax) \quad \overline{f : p \rightarrow q, x : p \vdash x : p} (Ax)}{\overline{p \rightarrow q, p \vdash q} (\rightarrow I)} (\rightarrow E)}{\overline{p \rightarrow q \vdash p \rightarrow q} (\rightarrow I)} (\rightarrow I)$$

Using our rules we get the rest of the terms.

$$\frac{\frac{\frac{\overline{f : p \rightarrow q, x : p \vdash f : p \rightarrow q} (Var) \quad \overline{f : p \rightarrow q, x : p \vdash x : p} (Var)}{\overline{f : p \rightarrow q, x : p \vdash f(x) : q} (Abst)} (Appl)}{\overline{f : p \rightarrow q \vdash \lambda x : p. f(x) : p \rightarrow q} (Abst)} (\rightarrow I)$$

2.3.2 Extended Curry-Howard Isomorphism

A pair and the projections can be defined in the untyped λ -calculus as a λ -expression. There is a natural correspondence between the BHK interpretation and set theory. Proofs of a product type are pairs. A product type corresponds to the cartesian product in set theory. By BHK, and the set theoretic correspondence it is natural to use an \wedge for the product type. By an analogous reasoning we would like to have \vee for the disjoint sum and so on. Consequently, a product cannot be

defined in the simply typed λ -calculus in a similar way as in the untyped case since the corresponding type uses an \wedge . This cannot be defined in the implicational fragment of propositional logic. In order to define pairs it is unavoidable to extend our definition of raw terms. But first we define the extended simple types.

Definition 2.3.15 (Extended Simple Types). We extend our definition as follows:

- (i) Every propositional variable is a simple type.
- (ii) If σ, τ are simple types, then so is $\sigma \rightarrow \tau, \sigma \wedge \tau, \sigma \vee \tau$
- (iii) \perp is a type
- (iv) 1 is a type

Nothing else is a simple type.

$\sigma \wedge \tau$ denotes a product type. \wedge relates to \times of the (cartesian) product. Hence, we will later denote it by $\sigma \times \tau$. This is a convention and more natural since we relate through BHK to set theory and later to category theory. It consists of pairs of elements $\langle M, N \rangle$. The first Element of a pair is of type σ and the second of type τ . $\sigma \vee \tau$ (also denoted by $\sigma \dot{\cup} \tau$) is a disjoint sum or disjoint union type (coproduct/variant type). Objects of this type consist of data of one of the types together with a flag pointing on the variant. \perp is the empty type. And 1 is a unit type with only one element.

Remark 2.3.14. We included 1 as a special kind of type following Barr and Wells [7]. This type corresponds later to the terminal object (Definition 3.0.17) and is normally not included in the simply typed λ -calculus.

Probably we could also use $\top := \perp \rightarrow \perp$ instead of the unit type 1 in order to let it correspond to the terminal object. For \top we need in general function extensionality to proof that two terms of type \top are indeed the same term of type \top .

This is not very problematic in the simply typed λ -calculus a la modern Church. Where we have

$$\frac{\overline{x : \perp \vdash x : \perp} \text{ (Var)}}{\vdash \lambda x : \perp. x : \perp \rightarrow \perp} \text{ (Abst)}$$

Remember that we put all terms in α -equivalence classes. So this function is unique up to α -equivalence. The non-abbreviated form serves better for the underlying logic but we could also consider the abbreviated \top and rename it in 1. But this might become a problem in other related calculi of the λ -cube where we do not necessarily have function extensionality. To introduce a type 1 explicitly, underlines the duality between the initial and the terminal object and their corresponding types.

Definition 2.3.16 (Raw Terms in the Extended Simply Typed λ -Calculus). We extend the set of raw terms recursively as follows.

- (i) Every individual variable x, y, z, \dots is a raw term.
- (ii) If M, N are raw terms then the application $(M \ N)$ is a raw term.
- (iii) If M is a raw term, x is an individual variable, and φ is a simple type then $(\lambda x : \varphi. M)$ is a raw term.
- (iv) $\pi_i \ M$, $\langle M, N \rangle$, $in_i^{\varphi \wedge \psi}(M)$, *case* M of $[x]P$ or $[y]Q$, $exf^\varphi(M)$, $*$ for some raw terms M, N, P, Q , variables x, y , $i \in \{1, 2\}$ and simple types φ, ψ are raw terms.

Nothing else is a raw term.

In the *case*-expression the by $[x]$ denoted variable x is bound in the corresponding branch (consider the derivation rules below for clarification). This means that P and Q depend on them. Still bound variables come together with a type to ensure type uniqueness. $in_i^{\varphi \vee \psi}$ is shorthand for $in_i : \varphi \vee \psi$. Again a raw term becomes a *term* if we can derive its type. The definition stays the same (Definition 2.3.10).

Definition 2.3.17 (Free Variables in the Extended λ_{\rightarrow} cf. [4]). We extend the definition by the following

$$\begin{aligned}
 FV(\langle M, N \rangle) &= FV(M) \cup FV(N) \\
 FV(\pi_i(M)) &= FV(in_i(M)) = FV(exf^\varphi(M)) = FV(M), \ i \in \{1, 2\} \\
 FV(\text{case } M \text{ of } [x]P \text{ or } [y]Q) &= FV(M) \cup (FV(P) \setminus \{x\}) \cup (FV(Q) \setminus \{y\})
 \end{aligned}$$

Substitution in the *case* construct is defined as follows:

$$(case\ M\ of\ [x]P\ or\ [y]Q)\ [N/z] = case\ M[N/z]\ of\ [x]P[N/z]\ or\ [y]Q[N/z].$$

Definition 2.3.18 (β –Reduction in the Extended λ_{\rightarrow} cf. [4]). We extend the definition of β –reduction by the smallest compatible relation \rightarrow_{β} satisfying

$$\begin{aligned} \pi_i(\langle M_1, M_2 \rangle) &\rightarrow_{\beta} M_i \\ case\ in_i^{\varphi \vee \psi}(M)\ of\ [x_1].P_1\ or\ [x_2].P_2 &\rightarrow_{\beta} P_i[M/x_i] \end{aligned}$$

The second rule corresponds to a detour in a proof (tree).

Definition 2.3.19 (Extended η –Rules). (i) For $M : \varphi \wedge \psi$, $\langle \pi_1 M, \pi_2 M \rangle \rightarrow_{\eta} M$
(ii) For $case\ M\ of\ [x_1].N[in_1 x_1/z]\ or\ [x_2].N[in_2 x_2/z] \rightarrow_{\eta} N[M/z]$
(iii) *Uniqueness of the unit element $*$* : For any $M : 1$, $M \rightarrow_{\eta} *$

The first rules correspond to detours in a proof. The third one is very important, since we will consider terminal object of a category.

Definition 2.3.20 (Extended Church λ_{\rightarrow} –Rules). We extend the implicational rules by the following.

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \pi_1 M : \varphi} (\wedge E_l)$$

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \pi_2 M : \psi} (\wedge E_r)$$

$$\frac{\Gamma \vdash M : \varphi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N \rangle : \varphi \wedge \psi} (\wedge I)$$

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash in_1^{\varphi \vee \psi}(M) : \varphi \vee \psi} (inleft)$$

$$\frac{\Gamma \vdash M : \psi}{\Gamma \vdash in_2^{\varphi \vee \psi}(M) : \varphi \vee \psi} (inright)$$

$$\frac{\Gamma \vdash M : \varphi \vee \psi \quad \Gamma, x : \varphi \vdash P : \gamma \quad \Gamma, y : \psi \vdash Q : \gamma}{\Gamma \vdash case\ M\ of\ [x].P\ or\ [y].Q : \gamma} (cases)$$

$$\frac{\Gamma \vdash M : \perp}{\Gamma \vdash exf^\varphi(M) : \varphi} (exfalse)$$

$$\frac{}{\Gamma \vdash * : 1} (*)$$

Remark 2.3.15. With the BHK interpretation, logic, set theory and category come close. A proof of $\varphi \wedge \psi$ is a pair $\langle M, N \rangle$ that contains a proof of φ and a proof of ψ . This corresponds to an element of the cartesian product and in general this type corresponds to the product in category theory.

$\wedge E_l$ and $\wedge E_r$ correspond to the projections. Conjunction introduction corresponds to the properties, and the unique morphism into the product such that the diagram commutes. We still need to define the projections such that $\pi_0 \langle M, N \rangle \succ M$ and $\pi_1 \langle M, N \rangle \succ N$.

Again, using BHK, a proof of $\varphi \vee \psi$ is a pair (i, M) with $i \in \{0, 1\}$. If $i = 0$, then M is a proof of φ . If $i = 1$, then M is a proof of ψ . This corresponds to an element of the set theoretic disjoint union and the type to the coproduct in category theory. The *(cases)* rule extracts a term from a product type while analyzing the cases. The variables x, y are bound in the corresponding branch. This means that P and Q depend on them. *cases* corresponds to the universal map out of the coproduct.

In the BHK interpretation there is no proof for \perp . \perp corresponds to the empty type, and hence to the initial object of a category. The *ex falso* rule corresponds to the arrow from the initial object to any other object (here φ). *ex f φ* can be regarded as a error message. The unit type 1 corresponds to a set theoretic singleton and to the terminal object. It can be regarded as a void type. We discussed this earlier in more detail.

Proposition 2.3.16 (Extended λ_{\rightarrow} Curry-Howard Isomorphism [4]). *Similarly,*

- (i) *If $\Gamma \vdash M : \varphi$ then $rg(\Gamma) \vdash \varphi$.*
- (ii) *If $\Delta \vdash \varphi$ then there are a term M and a context Γ such that $\Gamma \vdash M : \varphi$ in the extended λ_{\rightarrow} and $rg(\Gamma) = \Delta$.*

Example 2.3.17. If we would like to find an inhabitant of type $p \wedge q \rightarrow q$ in the empty context, we might proceed by the decorating method.

$$\frac{\frac{\frac{}{x : p \wedge q \vdash x : p \wedge q} (Var)}{x : p \wedge q \vdash \pi_2 x : q} (\wedge E_r)}{\vdash \lambda x : p \wedge q. \pi_2 x : p \wedge q \rightarrow q} (Abst)$$

Theorem 2.3.18 (Strong normalisation in the extended λ_{\rightarrow} [4]). *The extended simply typed λ -calculus is strongly normalizing.*

Each data type (product, sum, function space) comes with its own constructors, corresponding to introduction in the rules. These constructors can be seen as operators that create objects of this particular type. Destructors (elimination rules) use this particular objects of a particular type and and decompose them to an object of a component type of the initial type. They use them in their computation.

We observed some similarities between logic and a simple typed λ -calculus. We keep track of them and summarize them in Remark 4.2.5.

3 Categories

Category theory is the study of mathematical structures and their relations that appear throughout mathematics but also in category itself. We can regard it as one of the foundational paradigms of mathematics, similar to, for example, set theory. However, in contrast to set theory, category theory does not focus on the elements of the structure but on the (homo)morphisms between objects. It can be seen as the *morphism composition paradigm*. Homomorphisms are structure preserving maps. In a category there are objects and morphisms between these objects. Classical examples are the category of sets and the category of groups. In the category of sets the objects are sets and functions are the morphisms between them. The category of groups consists of groups as objects and group homomorphisms.

Categories have been introduced by Eilenberg and Mac Lane in order to define natural transformations in the context of algebraic topology in the mid of the 20th century. A natural transformation is a morphism between two functors that preserves the internal structure. A functor is a morphism between two categories that preserves the structure of these categories. A graph homomorphism behaves in a similar way as a functor. We could see it as some kind of graph homomorphism.

Category theory is used within logic, for example in type theory. Other applications are within pure mathematics in algebra and geometry, and in (theoretical) computer science. Type theory has its foundations in typed λ -calculi and connects to theoretical computer science and programming languages. We will study some connections to λ -calculi, but first give a brief introduction to some important notions of category theory. We will follow Leinster [8], and in some parts Barr and Wells [7] in this introduction.

It is possible to define categories from a graph theoretic perspective like Barr and Wells [7] do. An introduction from this point of view can be found in the appendix. Here, we adjust Barr Wells definition of a category in order to define it graph free. This way it comes closer to Grothendieck's and Leinster's definitions. The changes are innocuous.

Definition 3.0.1 (Category). A *category* \mathcal{C} consists of a collection of *objects* \mathcal{C}_0 , and a collection of *morphisms* (also called arrows or maps) \mathcal{C}_1 with the following properties:

- (i) For every two objects A, B there is a collection of morphisms from A to B . We denote it by $Hom_{\mathcal{C}}(A, B)$.
- (ii) Each morphism has a domain and codomain. The arrow $f : A \rightarrow B$ has domain A and codomain B . We denote this by $dom(f) = A$ and $cod(f) = B$. Domain and codomain are also called source and target.
- (iii) For any morphism f, g there is function

$$Hom(B, C) \times Hom(A, B) \rightarrow Hom(A, C)$$

$$(g, f) \mapsto g \circ f$$

with $dom(g \circ f) = dom(f)$ and $cod(g \circ f) = cod(g)$. If $dom(g) = cod(f)$, then (g, f) is called a composable pair.

$$\begin{array}{ccc} \bullet & \xrightarrow{f} & \bullet \\ \text{dom}(f) & & \text{cod}(g) \end{array}$$

- (iv) Composition is associative: For any composable morphisms $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow D$, it holds that $(h \circ g) \circ f = h \circ (g \circ f)$.
- (v) For any object A there is a morphism $id_A : A \rightarrow A$ such that $id_A \circ f = f$ for any morphism $f : B \rightarrow A$, and $g \circ id_A = g$ for any morphism $g : A \rightarrow B$.

The requirement in (v) is equivalent to the following: for any morphism $h : A \rightarrow B$, $h \circ id_A = h = id_B \circ h$.

To consider objects as sets and morphisms as functions may seem natural. This will be one approach to work with categories. There is a category *Sets* of sets and

functions.

From the set theoretic perspective, we might be interested in such size issues. Depending on the category $Hom_{\mathcal{C}}(A, B)$ can be a set.



Definition 3.0.2 (Small and Large Categories). If every $Hom_{\mathcal{C}}(A, B)$ for some objects A, B is a set, we say that the category is locally small. In this case we can call it hom-set. A category is small if its objects and arrows form sets and not proper classes, otherwise we call it large.

Set theoretic considerations play a role in higher category theory where the lastly introduced notions are used. This connection is considered for example in Shulman [9]. In the connection of type theory with category theory they might also play a role when considering composed types, and \square that forms a proper class in the calculus of constructions.

In the following we refer to Barr and Wells [7]. The trivial category $\mathbf{0}$ has no objects and no arrows. It is the smallest possible category. Smallest means that it is a subcategory of every category. If we understand the structure of categories as graphs than this notion refers to a subgraph of a graph.

This category is followed by the next smallest that has one and only one object and one and only one arrow, namely the identity arrow. We denote this category with $\mathbf{1}$.

The following categories are occasionally referred to the prior ones:

$\mathbf{1+1}$ with  and $\mathbf{2}$ with 

The choice of composites is forced. I.e. in $\mathbf{1+1}$ we have the following possibilities: $id_A \circ id_A$, $id_B \circ id_B$, and in $\mathbf{2}$ we obtain $id_A \circ id_A$, $id_B \circ id_B$, $(A \rightarrow B) \circ id_A$, $id_B \circ (A \rightarrow B)$.

Definition 3.0.3 (Category of Sets (*Sets*)). The category *Set* is the category whose objects are sets and whose arrows are functions with the composition of functions given by function composition and the set identity function from a set S to S for id_S .

We denote this category with *Set* or *Sets*.

Proof. Indeed *Set* is a category. This follows by checking the properties in the definition of a category in 3.0.1. We will proof the associativity of the composite for *Set*. The composite is defined as $g \circ f(x) = g(f(x))$ (see Definition A.1.3). Hence we obtain

$$h \circ (g \circ f)(x) = h((g \circ f)(x)) = h(g(f(x))) = (h \circ g)(f(x)) = (h \circ g) \circ f(x)$$

□

Definition 3.0.4 (Categories of Finite Sets (*Fin*) cf. [7]). The *category of finite sets* is the category whose objects are finite sets and whose arrows are all the functions between finite sets. We denote it with *Fin*.

Another example is the category of *sets and partial functions*. A partial function f is a function with domain S and codomain T for sets S, T . Sometimes it is denoted as a restricted function where we restrict the domain to a $S_0 \subseteq S$. It can be also seen as a function from S to $T \cup \{\perp\}$ with

$$s \mapsto \begin{cases} f(s), & \text{if } s \in S_0, \\ \perp, & \text{otherwise} \end{cases}$$

Definition 3.0.5 (Category of Sets and Partial Functions (*Pfn*) cf. [7]). The *Category of sets and partial functions*, denoted with *Pfn* has all sets as objects and all partial functions as arrows.

Proof. cf. [7] Most of the properties follow directly from *Sets*. We have to show that the composite of two partial functions is also a partial function.

Let $f : S \rightarrow T$ and $g : T \rightarrow V$ be partial functions with f defined on $S_0 \subseteq S$ and g on $T_0 \subseteq T$, then $g \circ f : S \rightarrow V$ is a partial function defined on the subset $\{x \in S_0 \mid f(x) \in T_0\}$.

Next, for associativity let $f : S \rightarrow T$, $g : T \rightarrow V$, $h : V \rightarrow W$ be partial functions with defined domains $S_0 \subseteq S$, $T_0 \subseteq T$, $V_0 \subseteq V$ respectively.

Firstly, we have to show that the $\text{dom}((h \circ g) \circ f) = \text{dom}(h \circ (g \circ f))$. The domain of $(h \circ g) \circ f$ is the set of all $x \in S_0$, such that $f(x)$ is in the domain of definition of $h \circ g$, namely T_0 . This is the set of all $t \in T$ such that $g(t) \in V_0$. Thus the domain of definition of $(h \circ g) \circ f$ is $\{x \in S_0 \mid g(f(x)) \in V_0\}$. With $g \circ f(x) = g(f(x))$,

we obtain that the domain of definition of $h \circ (g \circ f)$ is the same. For an x in the common domain of $(h \circ g) \circ f = h \circ (g \circ f)$ we proceed as in the associativity proof of Definition 3.0.3, obtaining $(h \circ g) \circ f(x) = h \circ (g \circ f)(x)$. \square

Definition 3.0.6 (Composite of Relations). Let $\alpha \subseteq S \times T$ be a relation, denoted with $S\alpha T$. If $\beta \subseteq T \times U$ is a relation then the *composite* $\beta \circ \alpha \subseteq S \times U$ is the relation from S to U defined as follows:

If $s \in S$ and $u \in U$, $(s, u) \in (\beta \circ \alpha)$ if and only if there is an element $t \in T$ such that $(s, t) \in \alpha$ and $(t, u) \in \beta$. We denote it with $S(\beta \circ \alpha)U$,

Example 3.0.1. Let $S := \{1, 2\}$, $T := \{2, 3\}$ and $U := \{1, 2, 3, 4\}$, with the following relations $S < T$, here $\{(1, 2), (1, 3), (2, 3)\}$ and $T = V$, here $\{(2, 2), (3, 3)\}$, respectively. Then the composite $S (= \circ <) T = \{(1, 2), (1, 3), (2, 3), (2, 2)\} = S \leq T$.

Definition 3.0.7 (Category of Sets and Relations - *Rel* cf. [7]). With the definition of composition above (3.0.6) the *category of sets and relations*, denoted with *Rel*, has sets as objects and relations as arrows. The identity arrow for a set S is the diagonal relation $\Delta_S = \{(x, x) \mid x \in S\}$.

A monoid (S, e, \circ) is a set S together with a binary operation $\circ : S \times S \rightarrow S$, that is associative, and an identity element e . A monoid basically is a semigroup with an additional neutral element. But in contrast to a semigroup, $S = \emptyset$ is not allowed. $(\mathbb{N}^+, +)$ is a semigroup, but not a monoid, while $(\mathbb{N}_0^+, 0, +)$ is also a monoid. A monoid is a category with one and only one object and its elements as arrows. This implies that composition denotes the binary operation.

Definition 3.0.8 (Monoid as a category cf. [7]). A monoid (S, e, \circ) determines the category $\mathcal{C}(M)$.

- (i) $\mathcal{C}(M)$ has only one object M .
- (ii) The arrows $M \rightarrow M$ of $\mathcal{C}(M)$ are the elements of S .
- (iii) Composition is the binary operation \circ of the monoid.

If we have an object $*$ in the monoid $(\mathbb{N}, 0, +)$ and morphisms $* \rightarrow *$. Indeed the identity morphism is the unit 0. If we compose a morphism that we call 1 with itself we get a morphism called 2. So we obtained $\circ(1, 1) = 2$ or in other words $1 \circ 1 = 2$. This way composition corresponds to the binary operation.

Of course, there is also a category of monoids.

Definition 3.0.9 (The category of Monoids (*Mon*)). *Mon* is the category with monoids as objects and monoid homomorphisms as arrows.

Functors

Most mathematical structures are defined together with a structure preserving map. In vector spaces they are represented in the form of linear mappings, in groups by group homomorphisms and in graphs by graph homomorphisms. All of these structures can be understood as a category, once we defined what the morphisms are. Taking these examples into account, there are the categories of vector spaces, groups and graphs. A category is a general structure that describes these properties. Since a category itself is also a mathematical structure it comes with a structure preserving map. This structure preserving map is called a functor. A functor is basically what we defined as a graph homomorphism (see Definition A.1.9 for a detailed explanation).

Definition 3.0.10 (Functor cf. [7]). Given two categories \mathcal{C}, \mathcal{D} , a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ is a pair of operations $F_0 : \mathcal{C}_0 \rightarrow \mathcal{D}_0$ and $F_1 : \mathcal{C}_1 \rightarrow \mathcal{D}_1$ that satisfy the following properties:

- (i) For every arrow $f : A \rightarrow B$ in \mathcal{C} there is an arrow $F_1(f) : F_0(A) \rightarrow F_0(B)$ in \mathcal{D} .
- (ii) Given an object $A \in \mathcal{C}_0$, then $F_1(id_A) = id_{F_0(A)}$
- (iii) Whenever the composite $g \circ_{\mathcal{C}} f$ is defined in \mathcal{C} then $F_1(g) \circ_{\mathcal{D}} F_1(f)$ is defined in \mathcal{D} and $F_1(g \circ_{\mathcal{C}} f) = F_1(g) \circ_{\mathcal{D}} F_1(f)$.

The notation with the two operations F_0 and F_1 is an emphasis. This means, for an object A , $F(A) = F_0(A)$ is an object and for an arrow $F(f) = F_1(f)$ is an arrow.

Example 3.0.2. Let S be a set and S^* denote the Kleene closure, i.e. S^* is the set of finite lists of elements of S . Then $(S^*, [], (++))$ is a monoid. The Kleene closure is of interest in the study of formal languages.

We denote strings as lists, the empty string ε as the empty list $[]$, and the operation of concatenation by $(++)$.

We do not need to assume that $S \neq \{\}$, since $\{\}^* = \{\varepsilon\} = \{[]\} \neq \emptyset$. This monoid is also called the *free monoid*, denoted with $F(S)$.

Since $F(S) = (S^*, [], (++))$ is a monoid, it is a category. Since Mon is a category, also free Mon is a category with free monoids as objects and monoid homomorphism as defined below as arrows.

This also induces a functor called the free functor $F' : Set \rightarrow Mon$, that takes a set S in Set to a free monoid $F(S)$. If $f : S \rightarrow T$ is an arrow in Set , then we define $F'(f) : F(S) \rightarrow F(T)$.

In particular, $F'(f)$ is the arrow $f^* : F(S) \rightarrow F(T)$, also denoted as $f^* : S^* \rightarrow T^*$.

We can define the monoid homomorphism f^* in free Mon as follows:

- (i) $f^*([]_{S^*}) = []_{T^*}$
- (ii) $f^*([s]) = [f(s)]$ for all $s \in S$
- (ii) $f^*([s_1, \dots, s_n]) = [f(s_1), \dots, f(s_n)]$ for all $[s_1, \dots, s_n] \in S^*$

Proof. The neutral element is preserved by definition. It remains to show, that it preserves the operation. Let $[s_1, \dots, s_n], [s'_1, \dots, s'_n] \in S^*$ be arbitrary but fixed, then

$$\begin{aligned}
 f^*([s_1, \dots, s_n] (++)_{S^*} [s'_1, \dots, s'_n]) &= f^*([s_1, \dots, s_n, s'_1, \dots, s'_n]) \\
 &= [f(s_1), \dots, f(s_n), f(s'_1), \dots, f(s'_n)] \\
 &= [f(s_1), \dots, f(s_n)] (++)_{T^*} [f(s'_1), \dots, f(s'_n)] \\
 &= f^*([s_1, \dots, s_n]) (++)_{T^*} f^*([s'_1, \dots, s'_n])
 \end{aligned}$$

□

For the functor $F' : Set \rightarrow Mon$ with $F'(S) = F(S)$ for a set S in Set and $F'(f) : F(S) \rightarrow F(T)$ for an arrow $f : S \rightarrow T$ in Mon , it remains to show that the identity and composition is preserved.

Proof. Assume that S is an object in Set . By definition $F'(id_S) : S^* \rightarrow S^*$ is $id_{S^*} : S^* \rightarrow S^*$ with $id_{S^*}([s_1, \dots, s_n]) = [id_S(s_1), \dots, id_S(s_n)] = [s_1, \dots, s_n]$ for all $[s_1, \dots, s_n] \in S^*$. Consequently, $F'(id_S) = id_{S^*} = id_{F'(S)}$.

If $f : S \rightarrow T$ and $g : T \rightarrow U$ are functions in Set , then $F'(f) = f^*$ and $F'(g) = g^*$ are the corresponding arrows in free Mon with $f^* : S^* \rightarrow T^*$ and $g^* : T^* \rightarrow U^*$.

By the definition of a category also $g \circ f : S \rightarrow U$ is in Set and $(g \circ f)^* : S^* \rightarrow U^*$ is in free Mon . Finally, $F'(g \circ_{Set} f) = (g \circ_{Set} f)^* = g^* \circ_{Mon} f^* = F'(g) \circ_{Mon} F'(f)$, where we used

$$\begin{aligned} (g \circ f)^*([s_1, \dots, s_n]) &= [(g \circ f)(s_1), \dots, (g \circ f)(s_n)] \\ &= [g(f(s_1)), \dots, g(f(s_n))] \\ &= g^*([f(s_1), \dots, f(s_n)]) \\ &= g^*(f^*([s_1, \dots, s_n])) \\ &= g^* \circ f^*([s_1, \dots, s_n]) \end{aligned}$$

for all $[s_1, \dots, s_n] \in S^*$. □

If we generalize the principle of a functor in a categorical way, we get the category Cat that has functors as arrows and categories as objects.

Definition 3.0.11 (Category of categories (Cat)). In the category of categories the objects are categories and the morphisms are functors. The identity functors are $ID_C : C \rightarrow C$ with $ID_{C_0}(A) = A$ and $ID_{C_1}(f) = f$ and composition of two functors $F : C \rightarrow D$, $G : D \rightarrow E$ by the composite functor $F \circ G : C \rightarrow E$ that preserves the identities and composition.

Again, we obtain the same kind of paradoxon as in Lemma A.1.1. Of course, the category of categories Cat is also an object of itself. To omit this paradoxon, we might define Cat to have only small categories as objects. This way the large category Cat does not have itself as an object.

In a group a bijective group homomorphism is a group isomorphism. Another way of defining this property is: a group homomorphism $g : G \rightarrow H$ is a group isomorphism, if there is a group homomorphism $h : H \rightarrow G$ such that $h \circ g = id_H$

and $g \circ h = id_G$. In categories, it is more natural to define isomorphisms in a similar way.

Definition 3.0.12 (Isomorphism). A morphism $f : A \rightarrow B$ is an isomorphism if there is an arrow $g : B \rightarrow A$ with $f \circ g = id_B$ and $g \circ f = id_A$.

If an isomorphism $f : A \rightarrow B$ exists, we say $A \cong B$.

Proposition 3.0.3. *Every functor preserves isomorphisms: Given a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ and an isomorphism f in \mathcal{C} , then $F(f)$ is an isomorphism in \mathcal{D} .*

Proof. Let f be arrows in \mathcal{C} and assume $f : A \rightarrow B$ is an isomorphism in \mathcal{C} . Then there is an arrow $g : B \rightarrow A$ in \mathcal{C} such that $f \circ g = id_B$ and $g \circ f = id_A$. We show that, given a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, then $F(f) : F(A) \rightarrow F(B)$ is an isomorphism in \mathcal{D} , i.e. there is an arrow $g' : F(B) \rightarrow F(A)$ in \mathcal{D} that $F(f) \circ g' = id_{F(B)}$ and $g' \circ F(f) = id_{F(A)}$. With the choice of $g' = F(g) : F(B) \rightarrow F(A)$ we find such an arrow:

$$\begin{aligned} F(g) \circ_{\mathcal{D}} F(f) &= F(g \circ_{\mathcal{C}} f) = F(id_A) = id_{F(A)} \\ F(f) \circ_{\mathcal{D}} F(g) &= F(f \circ_{\mathcal{C}} g) = F(id_B) = id_{F(B)} \end{aligned}$$

We obtained the first equality by the definition of a functor, the second by the antecedent and the last by the definition of a functor. \square

We define the product of two objects in Definition 4.1.1. It is also possible to define a product on categories.

Definition 3.0.13 (Product Category). Let \mathcal{C}, \mathcal{D} be categories, then we define the product category $\mathcal{C} \times \mathcal{D}$ as follows:

- (i) The objects are pairs $(C, D) \in \mathcal{C}_0 \times \mathcal{D}_0$.
- (ii) The arrows are pairs $(f, g) : (C, D) \rightarrow (C', D')$ where $f : C \rightarrow C'$ in \mathcal{C} and $g : D \rightarrow D'$ in \mathcal{D} .

Definition 3.0.14 (Binary Product Functor). For a category \mathcal{C} that has binary products the binary product functor $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ sends pairs of objects to their product $(A, B) \mapsto A \times B$ and pairs of morphisms $(f, g) \mapsto f \times g$ with $f : A \rightarrow C$ and $g : B \rightarrow D$ such that the following squares commutes:

$$\begin{array}{ccccc}
 A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\
 \downarrow f & & \exists! f \times g \downarrow & & \downarrow g \\
 C & \xleftarrow{\pi_1} & C \times D & \xrightarrow{\pi_2} & D
 \end{array}$$

In Definition 4.1.1 we introduce the projective arrows of a category. The concept of projections of the product can also be applied to functors where P_1 and P_2 define projective functors.

Definition 3.0.15. Let \mathcal{C}, \mathcal{D} be categories, then $P_1 : \mathcal{C} \times \mathcal{D} \rightarrow \mathcal{C}$ is the first projective functor, and $P_2 : \mathcal{C} \times \mathcal{D} \rightarrow \mathcal{D}$ is the second projective functor. They are defined by $P_1(C, D) = C$, and $P_2(C, D) = D$ for all objects $(C, D) \in \mathcal{C}_0 \times \mathcal{D}_0$, and $P_1(f, g) = f$, and $P_2(f, g) = g$ for all morphisms $(f, g) \in \mathcal{C}_1 \times \mathcal{D}_1$.

The principle of duality is used in some areas of mathematics. If there is an A that is the dual of B , then B is the dual of A . This allows sometimes to apply proved theorems on the dual in the opposite way. In the following example we consider the dual of a category.

Definition 3.0.16 (The Dual of a Category/ The Oposite Category). Given a category \mathcal{C} , then the opposite category \mathcal{C}^{op} has the same objects as \mathcal{C} and the same arrows of \mathcal{C} but reversed, i.e. for every arrow $f : A \rightarrow B$ in \mathcal{C} there is one and only one arrow $\overline{f} : B \rightarrow A$ in \mathcal{C}^{op} .

For composable arrows $f : A \rightarrow B$ and $g : B \rightarrow C$ in \mathcal{C} , the arrow $g \circ f : A \rightarrow C$ is in \mathcal{C} .

Then its opposite composition in \mathcal{C}^{op} is defined by $\overline{g \circ f} = \overline{f} \circ \overline{g}$.

Terminal and Initial Object

Another application of duality can be seen in terminal and initial objects.

Definition 3.0.17 (Terminal object and Initial object cf.[7]). Let T be an object in a category \mathcal{C} . T is called *terminal* if there is exactly one arrow $A \rightarrow T$ for each object A of \mathcal{C} , i.e. $\text{Hom}_{\mathcal{C}}(A, T)$ is a singleton. We denote the terminal object with 1 and the unique arrow $A \rightarrow 1$ with $\langle \rangle_A$.

Conversely, I is an initial object if there is exactly one arrow $I \rightarrow A$ for each object A of \mathcal{C} . We denote the initial object with 0 and the unique arrow $0 \rightarrow A$ with $!_A$.

Remark 3.0.4. For any terminal object 1 by this definition there is one unique morphism $1 \rightarrow 1$. Since the object is terminal there must be an arrow from the object to itself. Since the terminal object itself has to be equipped with an identity arrow by the definition of a category this morphism has to be the identity.

Example 3.0.5. In the category of sets the singletons are the only terminal objects.

Any singleton is a terminal object in *Set*.

Proof. Assume the set A is an arbitrary object in *Set* and let some singleton object be denoted by $\{*\}$, with some element $*$ in *Set*.

Existence of an arrow $f : A \rightarrow \{*\}$ follows by the fact that its possible to map all of A to a constant $*$, i.e. the constant function $f : A \rightarrow \{*\}$, $x \mapsto *$ exists.

Consider the arrows $f : A \rightarrow \{*\}$, $g : A \rightarrow \{*\}$. Their graphs are both $\{(a, *) \mid a \in A\}$. Thus, $f = g$ and thus f is the unique arrow with $A \rightarrow \{*\}$. \square

Any terminal object is a singleton in *Set*.

Proof. Assume that T is a terminal object in *Set*. $T \neq \emptyset$, because there is no function that maps to the empty set since we have to define an output. This means that T has to have at least one element.

Since T is terminal, there has to be an arrow $\{*\} \rightarrow T$. We show that any two elements in a terminal object in the category of sets has to be identical.

Suppose, for reductio, that there are two elements $t, u \in T$ with $t \neq u$. Then, there are two functions $f, g : \{*\} \rightarrow T$ with $f(*) = t$ and $g(*) = u$, by the definition of sets. Since X is terminal, $f = g$. But then by the definition of a function, also $t = f(*) = g(*) = u$. This contradicts the initial assumption that $t \neq u$. By

reductio, it follows that T contains only one element. Since T has been arbitrary, all terminal objects are singletons. \square

Other examples are the category of groups with the trivial group (group with one element) as a terminal object or in the category of categories Cat the terminal object is the category $\mathbf{1}$ (that has only one object and one arrow (the identity arrow)).

Initial objects of a category are unique up to unique isomorphism (for the initial objects see Lemma 2.1.8 in [8]). This holds due to duality also for terminal objects.

Lemma 3.0.6. *For any two terminal objects $1, 1'$ of a category there are unique isomorphisms $1 \rightarrow 1'$ and $1' \rightarrow 1$. In particular 1 and $1'$ are uniquely isomorphic.*

Proof. Since $1, 1'$ are terminal there are unique morphisms $f : 1 \rightarrow 1'$ and $g : 1' \rightarrow 1$, respectively. Any terminal object 1^i is equipped with exactly one arrow $1^i \rightarrow 1^i$ (see remark above) - the identity morphism. Hence, we conclude that $f \circ g = id_{1'}$ and $g \circ f = id_1$, showing both are the unique isomorphisms. We conclude that terminal objects are unique up to unique isomorphism. \square

4 Cartesian Closed Categories and a Typed Lambda Calculus

4.1 Cartesian Closed Categories

Products can be defined in arbitrary categories having certain properties. It is possible to define products of arbitrary arity. We concentrate on binary products. In the category *Set* the product is the cartesian product. Indeed a cartesian product $S \times T$ is equipped with two projections (coordinate functions) $\pi_1 : S \times T \rightarrow S$ and $\pi_2 : S \times T \rightarrow T$, defined by $\pi_1(s, t) = s$ and $\pi_2(s, t) = t$.

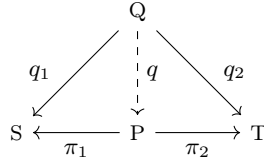
We will first analyse the properties of an element of the cartesian product of *Set* and its behaviour in a functional setting. Firstly, an element of the cartesian product $S \times T$ of two sets can be seen as an element in the product set or as a pair of two elements of different sets. Given two arbitrary elements of S and T , there is an element of $S \times T$ that contains the given element of S as first coordinate and the given element of T as second coordinate. Lastly, the notion of a function $f : S \times T \rightarrow V$ can be seen as a function in two variables or as a function in one variable, i.e. the first refers to $f(s, t)$ while the latter refers to an ordered pair as a variable $f(x)$ where $x := (s, t)$ and $s, t \in S \times T$. The latter is motivated by the following categorical context.

4.1.1 Products in Arbitrary Categories

Definition 4.1.1 (Binary Product cf. [7]). Let S, T be objects in a category \mathcal{C} , then a product of S and T is an object P together with the arrows $\pi_1 : P \rightarrow S$ and $\pi_2 : P \rightarrow T$ that satisfy the following condition:

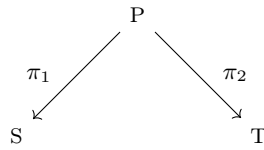
For any object Q and arrows $q_1 : Q \rightarrow S$ and $q_2 : Q \rightarrow T$, there is a unique arrow

$q : Q \rightarrow P$ such that $\pi_1 \circ q = q_1$ and $\pi_2 \circ q = q_2$:



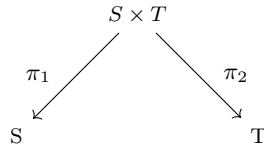
Projective arrows bear the notion of a product diagram or a product cone.

Definition 4.1.2 (Product Cone cf.[7]). Let S, T, P be objects of a category and π_1, π_2 the projections as defined above. The diagram



is called a *product diagram* or *product cone*.

It is customary to denote the product of the objects S, T by $S \times T$ as in



The index set $\{1, 2\}$ relates to these projections. We can also connect such an index set to a discrete graph \mathcal{G} with the vertices 1 and 2 and no arrows and connect it to the objects S, T of the product diagram. The *base of the product diagram* is the diagram $D : \mathcal{G} \rightarrow \mathcal{C}$, where $D(1) = S$ and $D(2) = T$. This is what we can also define as a category of $\text{cone}(D)$. This is a category that has cones as objects and the arrows are the unique morphisms between the cones. Its base is the functor that projects the shape of a shape category.

The base of the product diagram is the ordered pair (A, B) . Other shaped directed graphs can also be used for the same purpose. For a more detailed explanation consider Barr and Wells [7].

Example 4.1.1. In computer science tuple types are commonly used. For a stock list we could imagine a data base like $\{\text{ITEM}, \text{AMOUNT}\}$ as a base, constraining the projections π_{ITEM} and π_{AMOUNT} .

Example 4.1.2. The cartesian product

$$\begin{array}{ccccc}
 & & Q & & \\
 & \swarrow q_1 & \downarrow q & \searrow q_2 & \\
 S & \xleftarrow{\pi_1} & S \times T & \xrightarrow{\pi_2} & T
 \end{array}$$

is indeed a product of S and T in Set .

Proof. The cartesian product is equipped with two projections π_1, π_2 . Assume that there is an object Q and two morphisms $q_1 : Q \rightarrow S$ and $q_2 : Q \rightarrow T$. Assume that a morphism $q : Q \rightarrow S \times T$ that makes the diagram commute exists, then it satisfies $\pi_i \circ q(x) = \pi_i(q(x)) = q_i(x)$, for the cartesian projective functions $\pi_1(s, t) = s$ and $\pi_2(s, t) = t$, for any $(s, t) \in S \times T$. It exists with $q(x) = (q_1(x), q_2(x))$ for functions $q_1 : Q \rightarrow S$ and $q_2 : Q \rightarrow T$, since $\pi_i(q_1(x), q_2(x)) = q_i(x)$. And it is unique since for every other morphism $q' : Q \rightarrow S \times T$ with the properties above, it also holds that $\pi_i \circ q = q_i(x) = \pi_i \circ q'$, and then $q(x) = (q_1(x), q_2(x)) = q'(x)$ for all $x \in Q$. \square

The product in Cat is the product category of two categories (see Definition 3.0.13).

4.1.2 Exponential Objects

A binary operation is usually denoted by $\circ : S \times S \rightarrow S$. This notation is usually abbreviated with the infix notation $s \circ t$ or even st where $s, t \in S$. This notion can be transferred to a category. If the category has a product $S \times S$ satisfying Definition 4.1.1, then an arrow $S \times S \rightarrow S$ is a binary operation. More generally, a function $f : S \times T \rightarrow U$ of two variables can be thought of an arrow $f : S \times T \rightarrow U$ in a category where $S \times T$ is a product. In particular, this means that from the perspective of category theory a function of two variables can be regarded as a function in one variable, since there is only one arrow from the product to U . In this Section we introduce an approach to actually convert such a function - if possible - of two variables into a function of one variable. This is done via putting its values in a function object. An exponential object is an object of functions from A to B .

Definition 4.1.3 (Exponential Object). Let \mathcal{C} be a category where for each pair of objects A, B the binary product $A \times B$ exists. Then the exponential object B^A is an object in \mathcal{C} together with an arrow $eval : B^A \times A \rightarrow B$, such that for any object C and arrow $f : C \times A \rightarrow B$ there exists a unique morphism $\lambda f : C \rightarrow B^A$ such that the diagram

$$\begin{array}{ccc} & C \times A & \\ f \swarrow & & \searrow \lambda f \times id_A \\ B & \xleftarrow{eval} & B^A \times A \end{array}$$

commutes, i.e. the composite

$$C \times A \xrightarrow{\lambda f \times id_A} B^A \times A \xrightarrow{eval} B = C \times A \xrightarrow{f} B$$

B^A , is denoted also by $[A \rightarrow B]$, and $eval : B^A \times A \rightarrow B$ by $eval : [A \rightarrow B] \times A \rightarrow B$.

Example 4.1.3. In *Set* the exponential $B^A = [A \rightarrow B] = \{g \mid g : A \rightarrow B\}$ the set of all functions with domain A and codomain B .

Proof. The product is the cartesian product and it exists for each pair of objects as argued above. Consequently, the function $eval$ is defined as follows $eval(g, a) = g(a)$ where $g : A \rightarrow B$ and $a \in A$. This corresponds to an application in the λ -calculus.

Let $f : C \times A \rightarrow B$ be a morphism. We have to find a morphism λf such that $(\lambda f(c))(a) = f(c, a)$. $\lambda f(c)$ returns a function that maps from A to B . This together with a is the input for $eval$.

Let $a \in A$ and $c \in C$. We define $\lambda f(c)$ as the function $h \in B^A$ that maps every a to $f(c, a)$. That is $h(a) = f(c, a)$.

This function h is for every c unique and well-defined. This implies that the function $\lambda f : C \rightarrow B^A$ is well-defined. Then $\lambda f \times id_A : C \times A \rightarrow B^A \times A$ is given by $\lambda f \times id_A(c, a) = (\lambda f(c), id_A(a))$, where $\lambda f(c)$ is indeed this unique map such that $\lambda f(c)(a) = f(c, a)$, and hence also $eval \circ (\lambda f \times id_A)(c, a) = eval(\lambda f(c), id_A(a)) = eval(h, a) = h(a) = f(c, a)$, where $\lambda f(c) = h$ as defined above for some fixed $c \in C$ and $a \in A$. \square

We obtained that $\lambda f(c)(a) \equiv (\lambda y : C. \lambda x : A. f(y, x))(c)(a) \succ (\lambda x : A. f(c, x))(a)$. $\lambda f(c)$ is defined as the function that maps an $a \in A$ to $f(c, a)$. This means that $\lambda f : C \rightarrow B^A$ is the curried version of $\lambda f : C \times A \rightarrow B$ (for currying see Definition A.1.4). Consequently, since the diagram commutes, it is the same to take f or λf . From f we get λf with the universal property through the diagram. Conversely, $f(c, a) = \lambda f(c)(a)$. Currying or uncurrying a function does no harm to the function.

Remark 4.1.4. In a category with products the product of two categories defines a functor (see Definition 3.0.14). This means that the morphism $\lambda f \times id_A$ is the unique morphism making the following squares commute:

$$\begin{array}{ccccc}
 C & \xleftarrow{\pi_1} & C \times A & \xrightarrow{\pi_2} & A \\
 \lambda f \downarrow & & \exists! \lambda f \times id_A \downarrow & & id_A \downarrow \\
 B^A & \xleftarrow{\pi_1} & B^A \times A & \xrightarrow{\pi_2} & A
 \end{array}$$

4.1.3 Cartesian Closed Category

Definition 4.1.4 (Cartesian Closed Category cf.[7]). A category \mathcal{C} is called a cartesian closed category, if it satisfies the following properties:

- (i) There is a terminal object 1.
- (ii) For each pair of objects A, B the product $A \times B$ exists. More generally, we can add the requirement that \mathcal{C} has finite products instead.
- (iii) For all pairs of objects A, B , the exponential object exists, i.e.

$$\begin{array}{ccc}
 & C \times A & \\
 f \swarrow & & \searrow \lambda f \times id_A \\
 B & \xleftarrow{eval} & B^A \times A
 \end{array}$$

commutes for any object C .

Example 4.1.5. *Set* is cartesian closed.

Proof. The first condition on terminal objects is satisfied with the existence of a singleton set. This serves as a terminal object. In Example 3.0.5 we have showed that the terminal objects in Set are exactly the singletons. For each pair of objects, their product exists with the cartesian product. We showed this in Example 4.1.2. The last condition is satisfied by the existence of an exponential object $B^A = \{g \mid g : A \rightarrow B\}$ as the set of all functions with domain A and codomain B for all pairs of objects, see Example 4.1.3. We conclude that Set is cartesian closed. \square

4.2 Typed λ Calculus and Categories

In Barr and Wells [7] typed λ -calculi are considered in general without introducing them. Since there are a lot of them, we only considered the simply type λ -calculus as a basis. Higher order λ -calculi, systems for arithmetic and dependent type theories can be studied in Sørensen and Urzyczyn [4] and might relate to a cartesian closed category as well. We will not study this correspondences.

We introduced some relations on terms, Barr and Wells [7] adds another relation to their definition on typed λ -calculi that comes close to some kind of definitional equality. Definitional equality is an equivalence relation on the syntactic elements as terms, types, contexts, etc. For example propositional equality, judgmental equality, and typal equality as internal equalities can be used in type theory. Martin L  f defines some of this notions, and his work can be seen as a basic source on the discussion.

We will follow 6.3.3 in Barr and Wells [7] who describe that a λ -calculus has to have at least the following equations. We have to adjust the definition a little bit in order to relate it to our definition of the simply typed λ -calculus. We will see that a lot of the axioms relate to the formerly defined relations, type formation rules and reduction rules. For a term F the notion of $F(x)$ means that the variable x occurs freely in F .

Definition 4.2.1 (Equational property). (i) An equation of the form $M =_X N$ is a relation on terms where $M : \sigma$ and $N : \sigma$ for some type σ such that $\Gamma \vdash M : \sigma, \Gamma' \vdash N : \sigma$ for contexts Γ, Γ' and a finite set of variables

among which are all variables occurring freely in M or N . This means $\text{dom}(\Gamma) \cup \text{dom}(\Gamma') \subseteq X$.

- (ii) The relation $=_X$ is reflexive, symmetric and transitive.
- (iii) If $M : 1$, then $M =_{\{\}} *$
- (iv) If $X \subseteq Y$, then $M =_X N$ implies that $M =_Y N$.
- (v) If $M =_X N$, then $F M =_X F N$
- (vi) If $F =_X G$, then $F M =_X G M$
- (vii) If $F(x) =_{X \cup \{x\}} G(x)$ and $x : \sigma \in \Gamma, \Gamma'$, then $\lambda x : \sigma. F(x) =_X \lambda x : \sigma. G(x)$
- (viii) $\pi_1 \langle M, N \rangle =_X M$, $\pi_2 \langle M, N \rangle =_X N$, for $\langle M, N \rangle : \varphi \wedge \psi$, $M : \varphi$ and $N : \psi$ in context Γ
- (ix) For $M : \varphi \wedge \psi$, $M =_X \langle \pi_1 M, \pi_2 M \rangle$
- (x) If $\lambda x : \sigma. M(x)$ and $N : \sigma$ and x is substitutable in $M(x)$, then $(\lambda x : \sigma. M(x))N =_X M[N/x]$
- (xi) For $x \notin X$, $\lambda x : \sigma. M x =_X M$
- (xii) If y is substitutable for x in $M(x)$ and y is not free in $M(x)$ and vice versa, then $\lambda x : \sigma. M(x) =_X \lambda y : \sigma. M(y)$

Point (xii) is not necessary in our definition of λ_{\rightarrow} . We already put our terms in α -equivalence classes. We define two syntactical elements as equivalent if they have the same meaning. This means on the contrary that two terms are only equal if they are identical up to α -equivalence, not if we write $M =_X N$. The free variables of the set X that occur in M or N , has to be declared in the context otherwise we do not speak of a term, and of course a term comes with a context Γ such that $\Gamma \vdash a : \sigma$ is derivable (see Definition 2.3.10).

λ -calculi and cartesian closed categories are both (abstract) concepts to study functions in many variables. Both concepts are equivalent. This equivalence is shown by Lambek and Scott.

4.2.1 (Simply) Typed λ -Calculus to Cartesian Closed Category

We will now relate to the logic and the introduced simply typed λ -calculus. Barr and Wells [7] is not quite explicit on the definition of a λ -category, and the correspondence to logic, we will adjust the definition a bit and relate it primarily to the simply typed λ -calculus.

Definition 4.2.2 (Definition of the category $\mathcal{C}(\mathcal{L})$). For a (simply) typed λ -calculus \mathcal{L} we define the corresponding category $\mathcal{C}(\mathcal{L})$ as follows:

- (i) The objects of the category are the types.
- (ii) A morphism from an object φ to an object ψ is an equivalence class of terms of type ψ with one free variable of type φ , that is declared in context Γ . The equivalence relation is the least reflexive, symmetric, transitive relation induced by saying that two terms $M(x), N(y)$ (x occurs free in M , and y occurs free in N) are equal if they are definitionally equal (see Definition 4.2.1), and if they satisfy the following properties
 - (xiii) $M : \psi$ and $N : \psi$ for some type ψ
 - (xiv) $x : \varphi$ and $y : \varphi$ for some type φ
 - (xv) x is substitutable for y in N
 - (xiv) $M(x) =_{\{x\}} N(x)$, where we obtain $N(x)$ by $N[x/y]$

In particular, this is the equivalence class of all terms $M_j(x_i) : \psi$ and variables $x_i : \varphi$ that satisfy $x_i : \varphi \vdash M_j(x_i) : \psi$ such that

$$\frac{x_i : \varphi \vdash M_j(x_i) : \psi}{\vdash \lambda x_i : \varphi. M_j(x_i) : \varphi \rightarrow \psi} \text{ (Abst)}$$

This implies that two morphisms $\Gamma \vdash M(x) : \varphi$ and $\Gamma \vdash N(x) : \varphi$ are equal if $M(x) \equiv N(x)$.

- (iii) The composite of two morphisms $x : \varphi \vdash M : \psi$ and $y : \psi \vdash N : \rho$ is $x : \varphi \vdash N[M/y] : \rho$
- (iv) We define the identity morphism as $x : \varphi \vdash x : \varphi$.

Remark 4.2.1. If we postcompose $y : \varphi \vdash M : \psi$ with $x : \varphi \vdash x : \varphi$, we obtain $x : \varphi \vdash M[x/y] : \psi$. Since $M(x) \equiv M(y)$ this morphism is in the same equivalence class, and hence the morphisms $x : \varphi \vdash M[x/y] : \psi$, and $x : \varphi \vdash x : \varphi$ are equal. Conversely, if we precompose $y : \psi \vdash M : \varphi$ with $x : \varphi \vdash x : \varphi$, we obtain the composite $y : \psi \vdash x[M/x] : \varphi$ where indeed $x[M/x] \equiv M$ and hence the arrow is in the same equivalence class as $y : \psi \vdash M : \varphi$.

Proposition 4.2.2. *The category $\mathcal{C}(\mathcal{L})$ is cartesian closed.*

Proof. The type 1 is indeed a terminal object.

For every context Γ the arrow $\Gamma \vdash M : 1$ exists, since there is an equivalence class defined by the axiom

$$\overline{\vdash * : 1} \quad (*)$$

Since for every $M : 1$, M is equivalent to $*$, that is $M \equiv *$, any morphisms $\vdash M : *$ and $\vdash N : *$ are equal.

Any other terminal object is uniquely isomorphic (3.0.6).

The object that corresponds to the product type $\varphi \times \psi$ is indeed the binary product $\varphi \times \psi$.

The projections of a product correspond to the projections of a pair induced by the rules

$$\frac{\vdash M : \varphi \times \psi}{\vdash \pi_1 M : \varphi} (\wedge E_l) \quad \frac{\vdash M : \varphi \times \psi}{\vdash \pi_2 M : \psi} (\wedge E_r)$$

where the projections are defined as we did it in the (extended) simply typed λ -calculus. Consequently, the projections are defined as $\pi_i \langle M_1, M_2 \rangle \equiv M_i$ as in the simply typed λ -calculus implying existence. This means for any pair morphism $\vdash M : \varphi_1 \times \varphi_2$ the projection morphisms are given by $m : \varphi_1 \times \varphi_2 \vdash \pi_i m : \varphi_i$.

Existence and Uniqueness of the pair morphism:

Assume that there is an object ρ equipped with two morphisms $x_i : \rho \vdash M_i : \varphi_i$

for $i \in \{1, 2\}$, and assume that a morphism $z : \rho \vdash M : \varphi_1 \times \varphi_2$ that makes the product diagram commute exists, then it satisfies $(m : \varphi_1 \times \varphi_2 \vdash \pi_i m : \varphi_i) \circ (x_i : \rho \vdash M : \varphi_1 \times \varphi_2) = (x_i : \rho \vdash \pi_i m[M/m]) = (x_i : \rho \vdash M_i : \varphi_i)$.

By the \wedge introduction rule existence follows. Given two morphisms $\vdash M_1 : \varphi_1$ and $\vdash M_2 : \varphi_2$ there is a morphism $\vdash \langle M_1, M_2 \rangle : \varphi_1 \wedge \varphi_2$.

$$\frac{z : \rho \vdash M_1 : \varphi_1 \quad z : \rho \vdash M_2 : \varphi_2}{z : \rho \vdash \langle M_1, M_2 \rangle : \varphi_1 \times \varphi_2} (\wedge I)$$

By the definition $\pi_i \langle M_1, M_2 \rangle \equiv M_i$ and let $\langle M_1, M_2 \rangle = M$ for some M any other pair morphism $\vdash M' : \varphi_1 \times \varphi_2$ is in the same equivalence class since $\langle M_1, M_2 \rangle \equiv \langle \pi_1 M', \pi_2 M' \rangle \equiv M'$. We used what we formerly considered as a η -rule in 2.3.19 or as (xi) in Definition 4.2.1.

For any two types α and β the product exists. By the type formation also $\alpha \rightarrow \beta$ exists. We have an exponential object $\beta^\alpha := \alpha \rightarrow \beta$ together with the evaluation morphism that we define as follows $e : (\alpha \rightarrow \beta) \times \alpha \vdash (\pi_1 e) (\pi_2 e) : \beta$. For any type ζ and any morphism $f : \zeta \times \alpha \vdash P : \beta$ there is a morphism $l : \zeta \vdash \lambda x : \alpha. P[\langle l, x \rangle / f] : \alpha \rightarrow \beta$. This means that the morphism $f : \zeta \times \alpha \vdash \langle (\lambda x : \alpha. P[\langle l, x \rangle / f]), (x) \rangle : (\alpha \rightarrow \beta) \times \alpha$ precomposed with the morphism $\text{eval } e : (\alpha \rightarrow \beta) \times \alpha \vdash (\pi_1 e) (\pi_2 e) : \beta$ makes the diagram commute, since the composite

$$\begin{aligned} & f : \zeta \times \alpha \vdash ((\pi_1 e) (\pi_2 e))[\langle (\lambda x : \alpha. P[\langle l, x \rangle / f]), (x) \rangle / e] : \beta \\ & \equiv f : \zeta \times \alpha \vdash ((\pi_1 \langle (\lambda x : \alpha. P[\langle l, x \rangle / f]), (x) \rangle) (\pi_2 \langle (\lambda x : \alpha. P[\langle l, x \rangle / f]), (x) \rangle)) : \beta \\ & \equiv f : \zeta \times \alpha \vdash (\lambda x : \alpha. P[\langle l, x \rangle / f]) (x) : \beta \\ & \equiv f : \zeta \times \alpha \vdash P[\langle l, x \rangle / f] : \beta \\ & \equiv f : \zeta \times \alpha \vdash P[f / f] : \beta \\ & \equiv f : \zeta \times \alpha \vdash P : \beta \end{aligned}$$

Any other morphism $l' : \zeta \vdash \lambda x : \alpha. P[\langle l', x \rangle / f] : \alpha \rightarrow \beta$ satisfies the same equations and is therefore in the same equivalence class implying it is the same morphism. \square

4.2.2 Cartesian Closed Category to (Simply) Typed λ -Calculus

The internal language $\mathcal{L}_{\mathcal{C}}$ of a cartesian closed category is a (simply) typed λ -calculus. We will again restrict ourselves to the simply typed λ -calculus, and we will only give an intuition on how the backwards direction might be performed. Lambek [10] gives a detailed explanation.

Definition 4.2.3. Let \mathcal{C} a cartesian closed category equipped with finite products. The corresponding internal language $\mathcal{L}_{\mathcal{C}}$ is a typed λ -calculus defined as follows: The simple types of $\mathcal{L}_{\mathcal{C}}$ are defined by the objects of \mathcal{C} .

- (i) An object A corresponds to a simple type α .
- (ii) For given objects A, B and their corresponding types α, β the product object $A \times B$ corresponds to $\alpha \times \beta$, the exponential object B^A corresponds to $\alpha \rightarrow \beta$ and the terminal object 1 to the type 1 .

And we define the terms as follows:

- (i) Terms of to an object A corresponding type α are polynomial expressions $\mathcal{C}(x_1, \dots, x_n) : 1 \rightarrow \alpha$, generated by arrows $x_i : 1 \rightarrow \alpha$. This implies that there are countably many variables x_i of type α .
- (ii) Having a product means that there is a product cone with projections π_i in \mathcal{C} . Consequently, for morphisms $f : 1 \rightarrow \beta$ and $g : 1 \rightarrow \zeta$ the morphism $\langle f, g \rangle : 1 \rightarrow \alpha \rightarrow \beta \times \zeta$ induces for terms of corresponding types α and β , that there is an equivalence class of terms of type $\beta \times \zeta$ by the existence of $\langle f, g \rangle$. This directly corresponds to the following rule

$$\frac{\vdash M : \varphi \quad \vdash N : \psi}{\vdash \langle M, N \rangle : \varphi \times \psi} (\wedge I)$$

Note that the terminal 1 implies an empty context.

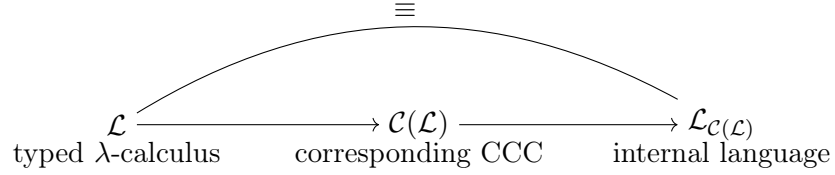
Other definitions for the terms and their equivalence classes are found in Lambek [10].

- (iii) It can be shown that proceeding like above induces an equivalence relation on terms satisfying Definition 4.2.1, defining the definitional equality in the simply typed λ -calculus.

Remark 4.2.3. Property (i) on terms is immediate by the properties of a terminal object we are able to generate any object A with $1 \rightarrow A$.

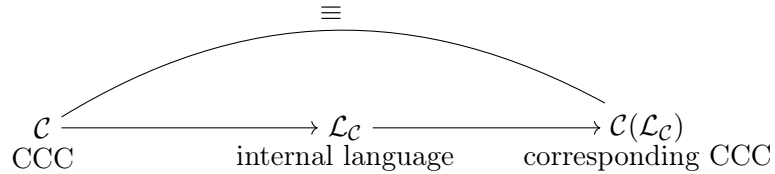
In the context of *Set* it is clear that the arrows $\{*\} \rightarrow A$ describe the elements of the set A . In particular, each map maps to precisely one element of A , and to no other by definition of a function.

Lambek and Scott (1986) show that if we construct from a typed λ -calculus a corresponding category then the internal language and the original typed λ -calculus are equivalent. In particular, the transformation from a typed λ -calculus into a cartesian closed category is innocuous to the typed λ -calculus.



This shows that we can regard cartesian closed categories as models for a typed λ -calculus, making it possible to regard categories as models for a programming language.

Theorem 4.2.4 (cf. [7]). *Given a cartesian closed category \mathcal{C} with internal language \mathcal{L} then the category $\mathcal{C}(\mathcal{L})$ is equivalent to \mathcal{C} .*



This means that if we reconstruct a category from this internal language, then the obtained category is equivalent to the initial cartesian closed category. Consequently, the transformation from a cartesian closed category to a typed λ -calculus is innocuous to the initial category.

These results imply that these two concepts are extensionally equivalent, which allows us to transfer results from one to the other.

4.2.3 Similarities in Logic, Typed λ -Calculus, and Category Theory

At the end of Chapter 4 we compared logic with a typed λ -calculus. Now, we can see more patterns induced by the correspondence to category theory.

Remark 4.2.5. We obtain the following correspondence between logic, simply typed λ -calculus, and category theory.

logic	typed λ -calculus	category theory
formula	type	object
propositional variable	type variable	object name
connective	type constructor	arrows together with commuting diagrams
implication	function space	exponential object
conjunction	product	product
disjunction	disjoint sum	coproduct
absurdity	empty type	initial object
proof	term	morphism
assumption	object variable	variable of a type together with an arrow
introduction	constructor	morphism into a combined object
elimination	destructor	morphism out of a combined object into its parts
proof detour	reducible expression	-
provability	inhabitation	-

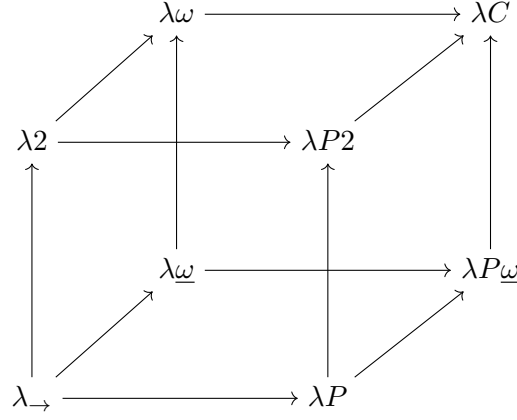
5 Conclusion

For mathematicians, category theory is an abstract field of study that leads to further insight on the particular structures they study, but also a research field by itself. It is not only useful for mathematicians but also useful for a logician or theoretical computer scientist. For a computer scientist working on foundational concepts like formal languages, computability, and logic it might help to get another perspective on the particular structure of study. Especially, the relation of type theory with category theory is interesting from a computer scientific perspective, since this connects the computer scientific research areas of applied logic and typed λ -calculus with a mathematical research area.

We have shown that intuitionistic logic naturally corresponds to the simply typed λ -calculus. We further showed that simply typed λ -calculus induces a cartesian closed category. Indeed, the transformation of a λ -calculus into a category does not preserve syntactical difference between the terms of an equivalence class, implying preservation up to definitional equivalence. The morphisms of a cartesian closed category that we constructed from a typed λ -calculus come closer to an extensional approach. Conversely, a typed λ -calculus is the internal (logical) language of a cartesian closed category.

If we reconstruct a category from this internal language, then the obtained category is equivalent to the initial cartesian closed category. Consequently, the transformation from a cartesian closed category to a typed λ -calculus is innocuous to the the initial category.

Lambek and Scott [1] show that if we construct a corresponding category from a typed λ -calculus, then the internal language and the original typed λ -calculus are equivalent. This means that the transformation from a typed λ -calculus into a cartesian closed category is innocuous to the typed λ -calculus. This shows that



- Terms that depend on Terms
- (\uparrow) Terms that depend on Types (type polymorphism)
 - (\nearrow) Types that depend on Types (type constructors)
 - (\rightarrow) Types that depend on Terms (dependent types)

Figure 5.0.1: λ -cube

we can regard cartesian closed categories as models for a typed λ -calculus. This makes it possible to regard categories as models for a programming language.

These results imply that these two concepts are extensionally equivalent, which allows us to transfer result from one to the other.

Other systems of the λ -cube (see Figure 5) are also part of the Curry-Howard Isomorphism, and relate to some kinds of intuitionistic logic as well. It would be interesting to investigate their correspondance to category theory, since the simply typed λ -calculus is not very expressive. Indeed, the simply typed λ -calculus is not a very good model for most of the programming languages. There are more powerful systems that allow for terms to depend on types (type polymorphism), types to depend on terms (dependent types), and types to depend on other types (type constructors). Gödel's system \mathcal{T} is an example where a combinatory calculus corresponds to Heyting arithmetic (intuitionistic axiomatization of arithmetic

HA^ω) via Kreisel's modified realizability. Examples for other such systems are: Polymorphic type theory is found in, for example, System F ($\lambda 2$) where $\lambda \rightarrow$ is embedded. In System F we have a new kind of λ -abstraction corresponding to a type in a special kind of second order intuitionistic logic (or in other words propositional second order logic since we quantify over a nullary predicate): $\Lambda p.M : \forall p.\sigma$, for M of type σ . With this, for example, a natural number defined by $\Lambda p.\lambda f : p \rightarrow p.x : p.f^n x$ depends on a type p and has type $\forall p.(p \rightarrow p) \rightarrow p \rightarrow p$. In $\lambda\omega$ we add types that depend on types to the polymorphic $\lambda 2$. This means that we introduce type constructors (kinds). This is, for example, interesting, if we have a type *ship* and would like to define a type *sailing ship* in context *ship*. This is what we also call a generic. Finally, in the Calculus of Constructions, we have also types that depend on terms. We can use this, for example, when we would like to define a type *isOdd(x)*, but only in the context in which the input x is of type \mathbb{N} . Indeed, we would like to have such an expressive power in a model for a programming language. System \mathcal{T} can be regarded as an extension on simple types, while other systems like System F, for example, adds polymorphism and relates to a special kind of second order logic, where we only quantify over nullary predicates. This cannot be understood as part of an extended simple typed λ -calculus. However, it is still a system where the simply typed λ -calculus is embedded.

It would be also helpful to study the correspondence of Lambek and Scott in more detail. The correspondance also works with finite products and the concept of an adjoint which are higher concepts that could be studied.

In the following years it would be interesting to study category theory in more detail and its expressive power relating to type theory, as well as the relation to topology, topoi and homotopy. This field of mathematical logic is still a relevant and active field of study where logic, algebra and computation intersect.

A Appendices

A.1 Basic Definitions

A.1.1 Sets

A set is a mathematical structure that is determined by the elements it includes. With the binary predicate \in we denote whether a specific element is a member of a set S . For every x , the statement $x \in S$ is either *true* or *false*. We can also understand sets as being defined via a predicate $P(x)$ that the elements of the set satisfy. For this purpose, we use a specific notation, namely the *setbuilder notation*. Let x be a variable that ranges over a specific collection and suppose that $P(x)$ is defined over the same type of data.

Thus, $\{x \mid P(x)\}$ (the *extension of P* [7]) is the set of all entities of the same type as x that also satisfy $P(x)$. But as we see in the following example this definition of a set allows us to construct a contradiction.

Lemma A.1.1 (Russel’s Paradox [7]). *It is not possible to define the extension of P when ranging over sets S where $P(S) := S \notin S$ for a set S , namely $\{S \mid S \text{ is a set and } S \notin S\}$ without causing a paradox.*

Proof. Suppose for a contradiction that $T := \{S \mid S \text{ is a set and } S \notin S\}$ is the set of all sets that are not members of themselves. Assume now, $T \in T$. This implies that $T \notin T$ by the predicates condition. But on the other hand if now $T \notin T$ it also follows by the predicate that $T \in T$. Since it holds that both $T \in T$ and $T \notin T$ we derived a logical contradiction by the argumentation. \square

This was one of the starting points for Zermelo’s publication on the foundation of set theory [11]. He came up with the approach of the “Axiom der Aussonderung“

(axiom of separation) is: If $P(x)$ is a property that is defined for the elements of a set S then S has a subset S_P , S_P consists of the elements x of S for which $P(x)$ holds. We denote it $S_P := \{x \in S : P(x)\}$. To conclude if we want to build a new set using a specific property then we can only do this when we start from a given set. In this way we are able to define sets without causing contradictions.

Example A.1.2. $M_{X \notin X} = \{X \in 2^{\mathbb{N}} \mid X \text{ is a set and } X \notin X\} = 2^{\mathbb{N}}$ These are all subsets of \mathbb{N} . For any subset in this set it holds by definition that it is not a subset of itself. If we would choose \mathbb{N} instead of $2^{\mathbb{N}}$ we would get \emptyset .

A.1.2 Functions

Definition A.1.1 (Graph of a Function). The *graph of a function* or function graph $f : S \rightarrow T$ is the set of ordered pairs: $\{(x, f(x)) \mid x \in S\}$.

The graph of a function from S to T is a subset of $S \times T$. We see it is also a relation. Conversely, a relation is not always the graph of a function. It has to satisfy the *functional property*, i.e. for all $s \in S$ there is one and only one $t \in T$ such that (s, t) is in the graph.

Example A.1.3. We take the sets $\{1, 2, 3, 4\}$, $\{0, 1\}$ but we consider now the relation $\{(1, 1), (1, 0), (3, 1), (3, 0), (2, 0), (4, 0)\}$. This cannot be the graph of a function as there is more than one $f(x) \in \{0, 1\}$ for $x \in \{1, 3\}$

Defining a function as a relation $\{(x, f(x)) \mid x \in S\}$ with the functional property does not determine the codomain. It is possible that we do not cover all elements of the codomain by the second coordinate of all tuples in the relation. Thus, we have to define the codomain as well.

Definition A.1.2 (Cartesian Product Of Two Functions cf. [7]). Let X, Y, S, T be sets and $f : X \rightarrow S$ and $g : Y \rightarrow T$ be functions. Then $f \times g : X \times Y \rightarrow S \times T$ is the function defined by $(f \times g)(x, y) = (f(x), g(y))$.

Definition A.1.3 (Composite Function cf. [7]). If $f : S \rightarrow T$ and $g : T \rightarrow U$, then the composite function $g \circ f : S \rightarrow U$ is the unique function with domain S

and codomain U for which $g \circ f(x) = g(f(x))$ for all $x \in S$. It is also notated with $f;g$ in computer science (we apply f first and then g).

In categories the domain of g is the set T and not only $im(f)$.

A function that takes more than one argument can be transferred to a sequence of functions, each of them taking one argument. With this concept, a function on two variables can be changed to a function taking one argument and returning a function. This is called currying.

Definition A.1.4 (Currying a Function). Given a function $f : A \times B \rightarrow C$ constructing the new function $\lambda f : A \rightarrow [B \rightarrow C]$ such that $\lambda f(a)$ for some $a \in A$ is the function that satisfies $\lambda f(a)(b) = f(a, b)$ for some $b \in B$ is called currying. $[B \rightarrow C]$ denotes the set of functions from B to C .

Currying can be understood as an unary operator that is applied on a function f and returns the curried equivalent λf , i.e. $curry(f) = \lambda f$. Function application is used to uncurry a function. This means to reverse the performed transformation when currying.

Definition A.1.5 (Uncurry a function). If $\lambda f : A \rightarrow [B \rightarrow C]$ is a function, then reconstructing the function $f : A \times B \rightarrow C$ is called uncurrying.

Consider the function $\lambda f : A \rightarrow [B \rightarrow C]$, then the uncurried version of this function is $f : A \times B \rightarrow C$. f is defined by $f(a, b) = (\lambda f(a))(b)$ with $a \in A$ and $b \in B$. This operation defines the inverse of the curry operator, i.e. $curry^{-1}(\lambda f) = f$.

A.1.3 Graphs

We are going to adapt graphs in a way that it is closer to a functional/ compositional interpretation. For this purpose we consider directed multigraphs, i.e. we allow parallel edges between two nodes as well as edges from a node to itself (self-loops). We can use graphs in the basic definitions of categories and in so called *commutative diagrams*. The latter is usefull in expressing compositions. That is we will compose paths that we will later recognize as maps and study their interaction.

We will now give a short definition of graphs and how we adapt it. Generally a *graph* \mathcal{G} is a tuple whose coordinates are sets, namely $(V(\mathcal{G}), E(\mathcal{G}))$. $V(\mathcal{G})$ is called the vertex set (or node set) of \mathcal{G} . Its elements are the vertices of \mathcal{G} . $E(\mathcal{G})$ is defined as the set of edges. The set of edges in a simple graph is a subset of $\binom{V(\mathcal{G})}{2} = \{F \subseteq V(\mathcal{G}) \mid |F| = 2\}$. We will specify now how we adapt graphs to be defined over collections.

Definition A.1.6 (Graph cf. [7]). Let \mathcal{G} be a graph. We will call its nodes also *objects* and for its edges we will use the term *arrows*. Thus, to specify a graph we have to specify both sets.

Each *arrow* must have a specific *source node* (or *domain node*) and a *target node* (or *codomain node*).

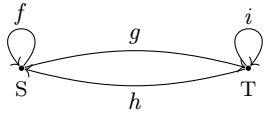
The notation $f : a \rightarrow b$ means that f is an arrow, the node a is its source, and the node b is its target.

As \mathcal{G} is a multigraph with allowed self-loops it is clearly possible that we can have parallel directed edges, i.e. we can have one or more arrows between a given source and target. Secondly it is also feasible that the source and the target of a given arrow is not necessarily distinct (self loop). This is what we call an *endoarrow* or *endomorphism*.

We will denote the collection of nodes of a graph \mathcal{G} by G_0 and the collection of arrows by G_1 . (Similarly, we do it with other letters: If \mathcal{H} is a graph we will denote the object set with H_0 and the arrow set with H_1).

Example A.1.4. Let \mathcal{G} be a graph, the set of objects be $\{S, T\}$ and the set of arrows with $\{f, g, h, i\}$ with $source(f) = target(f) = source(g) = target(h) = S$ and $source(i) = target(i) = source(h) = target(g) = T$.

The representation of \mathcal{G} is as follows



We can interpret the objects as sets and the arrows as functions. There is one and only one graph that has all sets as nodes and all functions between sets as arrows is called the *graph of sets and functions* (cf. [7]).

This graph is a large graph as defined in A.1.7. We interpret the source as domain and the target as codomain.

Definition A.1.7 (Small/Large Graph cf. [7]). A graph that has a *set* of nodes and arrows is called a *small graph*, otherwise it is a *large graph*.

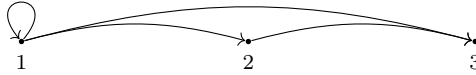
If the nodes and arrows of the graph form sets and not proper classes it is small, otherwise it is large.

The nodes (set of all sets) of the *graph of sets and functions* form a proper class. This is the class of all objects. The arrows (set of all functions between sets) of the *graph of sets and functions* also form a proper class.

Note that if G is a small graph then $source : G_1 \rightarrow G_0$ and $target : G_1 \rightarrow G_0$ are functions ([7]).

Definition A.1.8 (Finite Graph [7]). A graph is *finite* if the number of nodes and arrows is finite.

The graph in A.1.4 was finite. If we adjust the graph of A.1.4 it also depicts a relation between $\{1, 2\}$ and $\{1, 2, 3\}$, namely $\{(1, 1), (1, 2), (2, 3), (1, 3)\}$.



The graph of a function is a special case of a relation and can therefore be depicted in a similar way. When depicting relations the notion of a multigraph does not match our purpose. What fits is a directed and adjusted simple graph (no parallel edges with the same source and the same target).

A.1.4 Homomorphisms Of Graphs

Homomorphisms of graphs preserve the structure (shape) of a graph. A graph homomorphism can be understood as an arrow having graphs as nodes. In the chapter on categories we will introduce the notion of a functor. It is a quite similar concept.

Definition A.1.9 (Graph Homomorphism cf. [7]). A *homomorphism* ϕ from a graph \mathcal{G} to a graph \mathcal{H} , denoted with $\phi : \mathcal{G} \rightarrow \mathcal{H}$, is a pair of functions $\phi_0 : G_0 \rightarrow H_0$ and $\phi_1 : G_1 \rightarrow H_1$ with the property that if $u : m \rightarrow n$ is an arrow of \mathcal{G} , then $\phi_1(u) : \phi_0(m) \rightarrow \phi_0(n)$ in \mathcal{H} .

We will denote all three with ϕ omitting the subscripts. It will be clear if we are talking about arrows or nodes.

Remark A.1.5. cf. [7] We restate the definition above using source and target.

Let $source_{\mathcal{G}} : G_1 \rightarrow G_0$ be a source map that takes an arrow and gives its source value, $target_{\mathcal{G}} : G_1 \rightarrow G_0$ be a target function that returns the target value of the arrow, and similarly we define $source_{\mathcal{H}}$, $target_{\mathcal{H}}$.

Then the pair of maps $\phi_0 : G_0 \rightarrow H_0$ and $\phi_1 : G_1 \rightarrow H_1$ is a graph homomorphism if and only if

$$\begin{aligned} source_{\mathcal{H}} \circ \phi_1 &= \phi_0 \circ source_{\mathcal{G}} \text{ and} \\ target_{\mathcal{H}} \circ \phi_1 &= \phi_0 \circ target_{\mathcal{G}} \end{aligned}$$

Example A.1.6. Let \mathcal{G} be the graph



and \mathcal{H} the graph



Then there is a homomorphism $\phi : \mathcal{G} \rightarrow \mathcal{H}$ with $\phi_0(1) = \phi_0(2) = \phi_0(4) = A$ and $\phi_0(3) = B$.

We will have a look on the arrows and how they are preserved:

$$\begin{aligned} u_1(2) = 1 & \text{ then } (\phi_1(u_1) : \phi_0(2) = A) \rightarrow (\phi_0(1) = A) \\ u_2(2) = 2 & \text{ then } (\phi_1(u_2) : \phi_0(2) = A) \rightarrow (\phi_0(2) = A) \\ u_3(2) = 3 & \text{ then } (\phi_1(u_3) : \phi_0(2) = A) \rightarrow (\phi_0(3) = B) \\ u_4(3) = 4 & \text{ then } (\phi_1(u_4) : \phi_0(3) = B) \rightarrow (\phi_0(4) = A) \\ u_5(2) = 4 & \text{ then } (\phi_1(u_5) : \phi_0(2) = A) \rightarrow (\phi_0(4) = A) \end{aligned}$$

ϕ_1 can use 3 loops in \mathcal{H} , i.e. it has 27 possibilities to use the loop arrows with the fixed ϕ_0 from above.

Example A.1.7. cf. [7] The *identity homomorphism* of a graph \mathcal{G} , namely $id_{\mathcal{G}} : \mathcal{G} \rightarrow \mathcal{G}$ is defined by $(id_{\mathcal{G}})_0 = id_{G_0}$ and $(id_{\mathcal{G}})_1 = id_{G_1}$.

If \mathcal{H} is a graph with only one node n that has a self-loop, then there is a homomorphism from any graph \mathcal{G} to \mathcal{H} using $\phi_0(v_i) = n$ for all v_i in the nodes of that graph.

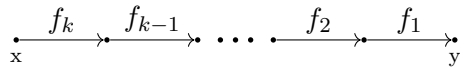
Having this mindset we might be interested to define a category in a graph theoretic approach.

A.1.5 Categories

Definition A.1.10 (Path [7]). Let $k > 0$ and \mathcal{G} a graph. A *path* from a node x to a node y is a sequence (f_1, f_2, \dots, f_k) of (not necessarily distinct) arrows for which

- (i) $\text{source}(f_k) = x$
- (ii) $\text{target}(f_i) = \text{source}(f_{i-1})$ for $i = 2, \dots, k$
- (iii) $\text{target}(f_1) = y$

An empty path is a path of length 0 from x to x . We denote it by $()$. We are drawing the graph as follows in order to be consistent with the composition as in 3.0.1.



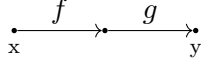
For any arrow f , (f) is a path of length 1.



Example A.1.8. If the diagram is $\begin{array}{c} g \\ \circlearrowleft \\ u \end{array}$ there is one *path of each length k* from u to u with $(g), (g, g), (g, g, g), \dots$ for $k = 1, 2, 3, \dots$

Definition A.1.11 (Composable Pairs cf. [7]). The set of paths of length k in a graph \mathcal{G} is denoted G_k .

We denoted the set of vertices with G_0 . In particular G_2 is the set of pairs of arrows (g, f) for which the target of f is the source of g . We will use G_2 for the definition of a category.



We call the members of the set *composable pairs of arrows*.

Remark A.1.9. cf. [7] We see that we have overloaded the notation of G_0 and G_1 but without causing conflicts. These collections are naturally having a one to one correspondence. G_1 refers to a collection of *arrows* as well as it does to the collection of *paths* of length 1. Equally, we use G_0 to represent either the collection of *nodes* of \mathcal{G} or the collection of *empty paths* (every empty path represents an empty node).

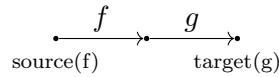
Definition A.1.12 (Category cf. [7]). A *category* is a graph \mathcal{C} together with two functions, $c : C_2 \rightarrow C_1$ and $u : C_0 \rightarrow C_1$ with the properties (c1) through (c4) that we define in the subsequent lines.

C_2 is the set of *paths of length 2*, C_1 is the collection of *arrows* and C_0 is the collection of *objects*.

The function c is called *composition*. If (g, f) is a composable pair, $c(g, f)$ is written $g \circ f$ and is called the *composite of g and f*.

If A is an object of \mathcal{C} , there is an arrow $u(A)$, denoted with id_A . We call it the identity arrow of the object A .

(c1) The source of $g \circ f$ is the source of f and the target of $g \circ f$ is the target g :



(c2) $(h \circ g) \circ f = h \circ (g \circ f)$ whenever either side is defined.

(c3) The source and the target of id_A are both A .

(c4) If $f : A \rightarrow B$, then $f \circ id_A = id_B \circ f = f$

Remark A.1.10. cf. [7] The composite c is defined on C_2 . This is important as now $g \circ f$ is defined if and only if the source of g is the target of f . That is the composition is a function whose domain is an subset of $C_1 \times C_1$ with the property

that the source of g is the target of f . Thus, and also with (c1) we get in (c2) that one side of the equation is defined if and only if the other side is defined.

We recognize that we are able to construct new arrows with the function c from composites. Thus a category can be seen as transitively closed.

A.2 Fixed Points in Untyped λ -Calculus and Cartesian Closed Categories

This section describes the fixed point property of the untyped lambda calculus and constructs a cartesian closed categories that has fixed points as well. We will introduce the concept of ω -CPOs that is a special kind of poset. This concept can be defined in the context of a cartesian closed category. This way cartesian closed categories can be used as semantic models for functional programming languages. Nevertheless, the assumption that any object in a cartesian closed category has fixed points is inconsistent with other assumptions on a category that would be desirable (cf. Barr and Wells [7]). We follow Barr and Wells [7].

Definition A.2.1 (The Category of Posets *Pos* cf.[7]). The category of posets has monotone functions as arrows. Let $(S, \leq_S), (T, \leq_T)$ be posets and $f : S \rightarrow T$ a monotone function. f is monotone if and only if if $x \leq_S y$ in S then $f(x) \leq_T f(y)$ in T .

We prove that the Category of Posets is a category.

Proof. Firstly, we prove that the identity function is monotone. Consider $id_x : X \rightarrow X$ where (X, \leq_X) is a poset and $x \mapsto x$. Assume $x \leq_X y$ in X then $id_X(x) = x \leq_X y = id_X(y)$ in X .

Associativity of the composition and identity properties follow from Sets.

It remains to show that the composition of monotone functions is again monotone. Let $f : S \rightarrow T$ and $g : T \rightarrow U$ be monotone with $(S, \leq_S), (T, \leq_T)$, and (U, \leq_U) being posets. Let $x \leq_S y$ in S then $f(x) \leq_T f(y)$ in T by the definition of a monotone function. By the fact that g is a composable function and the monotony

of g it follows that $g(f(x)) \leq_U g(f(y))$ in U where $g(f(x)) = g \circ f(x)$ and $g(f(y)) = g \circ f(y)$. That means $g \circ f : S \rightarrow U$ is monotone. \square

A more general related result is that sets with relations and homomorphisms as arrows also form a category. The proof proceeds in a similar way. Note that usually the poset relation on a set S is denoted with \leq_S , being placeholder for \subseteq and other partial order relations. We introduce the notion of ω -complete partial orders. These are a special kind of posets. Firstly, we define a surpreum on a subset of a partial order and an ω -chain.

Definition A.2.2 (Surpreum cf.[7]). Let (S, \leq_S) be a poset and $R \subseteq S$. Then the *supremum* is an element $s \in S$ such that for all $r \in R$, $r \leq_S s$ and for all $s' \in S$ with $r \leq_S s'$ for all $r \in R$, also $s \leq s'$ (meaning s is the smallest upper bound).

Definition A.2.3 (ω -Chain cf.[7]). Let (S, \leq_S) be a poset. We call the infinite sequence s_0, s_1, s_2, \dots of elements of S with $s_i \leq_S s_{i+1}$, $i \in \mathbb{N}$ an ω -chain.

For arbitrary s_i, s_{i+1} with $s_i \leq_S s_{i+1}$, we can construct an infinite ω -chain like $s_i \leq_S s_i \cdots \leq_S s_i \leq_S s_{i+1}$ or $s_i \leq_S s_{i+1} \leq_S s_{i+1} \leq_S s_{i+1} \leq_S \dots$. With these two definitions we give a definition of an ω -complete partial order.

Definition A.2.4 (ω -Complete Partial Order (ω -CPO) cf.[7]). A poset (S, \leq_S) is a ω -complete partial order, also ω -CPO, if every ω -chain has a supremum. If the poset S additionally has a minimum element \perp , s.t. for all $s \in S$, $\perp \leq_S s$, then the poset is called a *strict ω -CPO*.

Example A.2.1. cf.[7] An example for a strict ω -CPO is the set \mathfrak{P} of partial functions from S to S .

If we understand a one-parameter function as a set of tuples, then a partial function f on S is a set of ordered pairs with the following property: if $(s, t) \in f$ and also $(s, t') \in f$ then $t = t'$, by definition of a (partial) function. Note that since f is partial we do not require it to have a $t = f(s)$ for each $s \in S$.

By defining $\leq_S := \subseteq$ we obtain the poset $(\mathfrak{P}, \subseteq)$. In particular, we can understand the order relation as follows: Let f, g be partial functions on S . $f \subseteq g$ if and only if $\text{dom}(f) \subseteq \text{dom}(g)$ and for every $x \in \text{dom}(f)$, $f(x) = g(x)$.

Proposition A.2.2 (cf.[7]). \mathfrak{P} is a strict ω -CPO.

Proof. cf.[7] Let \mathfrak{C} be a ω -chain in \mathfrak{P} . Since for every $f_i \in \mathfrak{C}$ it holds that $f_i \subseteq f_{i+1}$, $\text{dom}(f_i) \subseteq \text{dom}(f_{i+1})$ and for every $x \in \text{dom}(f_i)$, $f_i(x) = f_{i+1}(x)$. Then the union $s = \bigcup_{i \in \mathbb{N}} f_i$ is the supremum since it is maximal with respect to inclusion. It remains to show that the union s is a partial function. Assume that $(x, y), (x, z) \in s$ then there are partial functions $f_{i'}, f_{i''} \in \mathfrak{C}$ such that $(x, y) \in f_{i'}$ and $(x, z) \in f_{i''}$. By definition of a chain and the ordering relation \subseteq , there is a partial function $f_j \in \mathfrak{C}$ such that $f_{i'} \subseteq f_j$ and $f_{i''} \subseteq f_j$, i.e. $(x, y), (x, z) \in f_j$. Since f_j is a partial function it follows that $z = y$. We showed for arbitrary $(x, y), (x, z) \in s$, also $z = y$. Thus, s is a partial function. We obtained that \mathfrak{P} is a ω -CPO.

The empty function $\langle \rangle : S|_{\emptyset} \rightarrow S$, or $\langle \rangle = \emptyset$ is the minimum of each ω -chain. Consequently, $\langle \rangle = \perp$. \square

Definition A.2.5 (Continuous Function between ω -CPOs cf.[7]). Let $f : S \rightarrow T$ with S, T being ω -CPOs. f is continuous if and only if if s is the supremum of a chain $\mathfrak{C} = \{s_0, s_1, \dots\}$ in S then $f(s)$ is the supremum of the image $f(\mathfrak{C}) = \{f(s_i) \mid i \in \mathbb{N}\}$ in T . A continuous function between strict ω -CPOs that preserves the \perp element is strict, i.e. $f(\perp_S) = \perp_T$.

Proposition A.2.3 (ω CPO category). (Strict) ω -complete partial orders (as objects) with (strict) continuous maps (as arrows) form a category.

Proof. Firstly, we show that the identity functions are continuous. Let $\text{id}_S : S \rightarrow S$ with (S, \leq_S) be a (strict) ω -CPO and $s \mapsto s$. Consider an arbitrary ω -chain $\mathfrak{C} = \{s_1, s_2, \dots\}$ in S . Assume $s \in S$ is its supremum. By the mapping definition of the identity, the image of the chain is the set $\text{id}_S(\mathfrak{C}) = \{\text{id}_S(s_1), \text{id}_S(s_2), \dots\} = \{s_1, s_2, \dots\}$. By definition of the identity the set contains the same elements as the chain \mathfrak{C} . For sure, s is also the supremum of this set. Since $s = \text{id}_S(s)$, $\text{id}_S(s)$ is the supremum of $\text{id}_S(\mathfrak{C})$. s and \mathfrak{C} has been arbitrary. We conclude that id_S is continuous.

Again associativity of the composition and identity properties follow from Sets.

It remains to show that the composition of (strict) continuous functions is again (strict) continuous. Let $f : S \rightarrow T$ and $g : T \rightarrow U$ be (strict) continuous and let $(S, \leq_S), (T, \leq_T)$ and (U, \leq_U) be (strict) ω -CPOs. Let $s \in S$ be the supremum of an ω -chain $\mathfrak{C} = \{s_1, s_2, \dots\}$ in S . By continuity of f , $f(s)$ is the supremum of the image $f(\mathfrak{C})$. We order the elements of $f(\mathfrak{C})$ according to the order \leq_T and obtain an ω -chain $f(\mathfrak{C})_T$ with supremum $f(s)$. By continuity of g , $g(f(s))$ is the supremum of the image $g(f(\mathfrak{C})_T)$. Since $g(f(s)) = f \circ g(s)$, $g \circ f(s)$ is the supremum of $g(f(\mathfrak{C})_T) = g \circ f(\mathfrak{C})$. It follows that $g \circ f$ is continuous. If f, g are strict continuous functions on a strict ω -CPO then $f(\perp) = \perp$ and $g(\perp) = \perp$. Thus the composition $g \circ f(\perp) = g(f(\perp)) = \perp$ is strict continuous. \square

Fixed points occur in different structures. Recursion in the untyped lambda calculus is done via a fixed point combinator. We will consider this in thm. A.2.6.

Definition A.2.6 (Fixed Point of a Function). A fixed point of a function $f : S \rightarrow S$ is an element $s \in S$ with the property that $f(s) = s$.

Consider the function $f : \mathbb{R} \rightarrow \mathbb{R}$ with $x \mapsto x^2$ then $0, 1$ are fixed points since $f(1) = 1, f(0) = 0$. The successor function on the natural numbers has no fixed points at all.

Example A.2.4. cf.[7] Let now \mathfrak{P} be defined as in A.2.1 with $S = \mathbb{N}$, i.e. it is the set of partial functions with domain and codomain \mathbb{N} .

Then there is a function $\phi : \mathfrak{P} \rightarrow \mathfrak{P}$ that maps a partial function f of \mathfrak{P} with $f : \mathbb{N} \rightarrow \mathbb{N}$ to a partial function $g : \mathbb{N} \rightarrow \mathbb{N}$ of \mathfrak{P} . g is defined as follows

$$\begin{aligned} g(0) &= 1 \\ \text{and for } n > 0 \\ g(n) &= \begin{cases} nf(n-1) & , \text{iff } f(n-1) \text{ is defined} \\ \text{undefined} & , \text{otherwise} \end{cases} \end{aligned}$$

For example for $f(n) = n^2$

$$\phi(f)(n) = \begin{cases} 1 & , \text{if } n = 0 \\ nf(n-1) = n(n-1)^2 & , \text{if } n > 0 \text{ since } f(n-1) \text{ is def. for } n > 0 \end{cases}$$

ϕ is a continuous function.

Proof. (cf. [7]) ϕ is a function between ω -CPOs. Let $\mathfrak{F} = \{f_0, f_1, \dots\}$ be an arbitrary ω -chain with $f_i : \mathbb{N} \rightarrow \mathbb{N}$ and let f_s be its supremum. As argued before f_s is the union of all partial functions f_i . To show that ϕ is continuous, we have to show that $\phi(f_s)$ is the supremum of $\phi(\mathfrak{F}) = \{\phi(f_i) \mid i \in \mathbb{N}\}$, particularly that for every n , $\phi(f_s)(n)$ is defined if and only if $\phi(f_i)(n)$ is defined for some i and for this i $\phi(f_s)(n) = \phi(f_i)(n)$. Again the supremum will be the union of $\phi(f_i)$. We begin with the backward direction. Suppose that $\phi(f_i)(n)$ is defined. If $n = 0$ then $\phi(f_i)(0) = \phi(f_s)(0) = 1$. Otherwise, since $\phi(f_i)(n)$ is defined, by definition $\phi(f_i)(n) = nf_i(n-1)$ is. Since f_s is the union of the f_i it holds that $f_s(n-1) = f_i(n-1)$. Thus, we obtain that $\phi(f_s)(n) = nf_s(n-1) = nf_i(n-1) = \phi(f_i)(n)$. It remains to show the forward direction. Assume that $\phi(f_s)(n)$ is defined. Then $f_s(n-1)$ is defined, $\phi(f_s)(n) = nf_s(n-1)$. Since f_s is the union of all f_i there is some i for which $f_i(n-1)$ is defined and $f_s(n-1) = f_i(n-1)$, by definition of a function. By definition of ϕ , $\phi(f_i)(n) = nf_i(n-1) = nf_s(n-1) = \phi(f_s)(n)$. This completes the proof. \square

The unique fixed point of ϕ is $f(n) = n!$

Proof. (cf. [7]) f is defined on all natural numbers. Also $f(0) = 1 = \phi(f)(0)$ and $f(n) = nf(n-1)$. By definition of ϕ and totality of f we have that $\phi(f)(n) = nf(n-1) = f(n)$ for all $n > 0$. Hence, $\phi(f) = f$. Assume that there is another fixed point g . So $\phi(g) = g$, then $g(0) = 1$ and for an arbitrary $n \in \mathbb{N}$ $g(n) = \phi(g)(n) = ng(n-1)$ if $g(n-1)$ was defined. But if $g(n) = ng(n-1)$ by mathematical induction on n we obtain that $g(n) = n!$. Hence, $g = f$. We conclude the factorial function is the unique fixed point of ϕ . \square

This example shows that is possible to define a morphism between the objects of a ω CPO category that have fixed points.

Definition A.2.7 (Least Fixed Point). A least fixed point $\mu \in S$ is a fixed point of a function $f : S \rightarrow S$ with (S, \leq_S) being a poset, and for all fixed points $s \in S$ with $s \neq \mu$ it holds that $\mu \leq_S s$.

This concept can also be applied to the ω -CPO \mathfrak{P} from above.

Proposition A.2.5 (Least Fixed Point of a Continuous Function on a ω -CPO cf.[7]). *A continuous function $f : S \rightarrow S$ on a strict ω -CPO (S, \leq_S) has a least fixed point.*

Proof. cf.[7] By the strictness of the ω -CPO (S, \leq_S) , we have a minimum element \perp in S . Then $\perp \leq_S s$ for all $s \in S$. In particular, since $f(\perp) \in S$, also $\perp \leq_S f(\perp)$ holds. Assume that for an arbitrary but fixed $n \in \mathbb{N}$, $f^n(\perp) \leq_S f^{n+1}$. Consider the case $f^{n+1}(\perp) \leq_S f^{n+2}$. By hypothesis $f^n(\perp) \leq_S f^{n+1}$ holds. Because f is continuous it preserves the bottom element and we obtain $f^n(\perp) \leq_S f^{n+1}(\perp) \leq_S f^{n+2}$. This way we obtain the ω -chain $\mathfrak{C} = (\perp, f(\perp), f \circ f(\perp), \dots, f^n(\perp), f^{n+1}(\perp), \dots)$.

Let $s \in S$ be the supremum of the chain \mathfrak{C} . Since f is continuous, then $f(s)$ in S is the supremum of the the image $f(c) = \{f(c_i) \mid c_i \text{ occurs in } c\} = (f(\perp), f(f(\perp)), \dots, f^n(\perp), f^{n+1}(\perp), \dots)$. But by construction of $f(c)$, s is also an upper bound for $f(c)$. Consequently, $f(s) \leq_S s$. But the minimum element $\perp \in c$ and $\perp \notin f(c)$, so $f(s)$ is also an upper bound on \mathfrak{C} . Hence, $s \leq f(s)$. It follows that $f(s) = s$. Hence s is a fixed point. It remains to show that s is the least fixed point. Consider now an arbitrary fixed point $t \in S$ with $t = f(t)$, then $\perp \leq_S t, f(\perp) \leq_S f(t) = t, \dots, f^n(\perp) \leq_S f^n(t) = f(\dots(f(t))) = t$. Thus, t is an upper bound of \mathfrak{C} . Since s is its supremum, we get that $s \leq t$. It follows that s is the least fixed point. \square

A.2.1 Fixed Points in the Untyped Lambda Calculus

In the λ -calculus it is not possible that functions can call themselves since all functions are anonymous. If we would like to model recursive functions we shall express a recursive function in a non-recursive approach. For this purpose we use fixed points in the λ -calculus. One of the reasons why we introduced the untyped λ -calculus is that only in the untyped λ -calculus every term has a fixed point. In the typed λ -calculus this is not possible. Imagine the successor function on the natural numbers. This function has no fixed point.

We consider the factorial function from above that is most commonly defined by recursion.

<pre> 1 fac 0 = 1 2 fac n = mult n fac n-1 </pre>

In a functional programming language we would name the faculty function somehow and define the function by self-application. In the chapter on the λ -calculus we considered the λ -calculus as a programming language. In λ -calculus something like

$$fac = \lambda n. ifelse (isZero n) 1 (mult n (fac (pred n)))$$

where fac occurs on the left and right side is not possible. If we use function application we could imagine something like

$$fac = \lambda f. \lambda n. ifelse (isZero n) 1 (mult n (f (pred n))) fac$$

particularly, we would like to have something like this

$$\begin{aligned} fac' &= \lambda n. ifelse (isZero n) 1 (mult n (fac' (pred n))) \\ \beta &\leftarrow \lambda f. \lambda n. ifelse (isZero n) 1 (mult n (f (pred n))) fac' \end{aligned}$$

If we substitute $\lambda f. \lambda n. ifelse (isZero n) 1 (mult n (f (pred n)))$ with F we obtain

$$fac' =_{\beta} F fac'$$

This reminds us of a fixed point.

Definition A.2.8 (Fixed points in the λ -calculus). Let f, n be λ -terms. n is a fixed point of f if and only if $f n =_{\beta} n$ or as we will use it $n =_{\beta} f n$.

Theorem A.2.6 (Fixed points in the λ -calculus). *In the untyped λ -calculus every term f has a fixed point.*

Proof. Let $Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$ be the Curry fixed point combinator

and let $f \in \Lambda$. Let $n = Y f$. We show that n is a fixed point of f , i.e. $f n =_\beta n$.

$$\begin{aligned}
 n &= Y f \\
 &= (\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) f \\
 &\rightarrow_\beta (\lambda x.f (x x)) (\lambda x.f (x x)) \\
 &\rightarrow_\beta f ((\lambda x.f (x x)) (\lambda x.f (x x))) \\
 &\leftarrow_\beta f ((\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) f) \\
 &= f (Y f) \\
 &= f n
 \end{aligned}$$

We obtained that $n =_\beta f n$. Thus, by $n = Y f$, $Y f$ is the fixed point of f . Since f was arbitrary, this holds for every term f . \square

We consider the factorial function example again. If we use the Y combinator that we introduced in the proof again then

$$\begin{aligned}
 fac &= Y f \\
 &=_\beta f (Y f) \\
 &=_\beta f (f (Y f)) \dots
 \end{aligned}$$

where we substituted $\lambda f.\lambda n.ifelse (isZero n) 1 (mult n (f (pred n)))$ with f .

To get an idea of how an application of a such defined recursive function looks like, we apply the number 2, i.e. we calculate $fac(2)$. Note that it is important to reduce leftmost, outermost, if possible and have a base case. Otherwise the Y combinator extends the term, and we do not terminate nor reach the normal form.

We take care that f is only an abbreviation for the bigger term

$\lambda f.\lambda n.ifelse (isZero n) 1 (mult n (f (pred n)))$ within the reduction.

Example A.2.7.

$$\begin{aligned}
fac\ 2 &:= (Y\ f)\ 2 \\
&\rightarrow_{\beta} f\ (Y\ f)\ 2 \\
&\rightarrow_{\beta} (\lambda f.\lambda n.ifelse\ (isZero\ n)\ 1\ (mult\ n\ (f\ (pred\ n))))\ (Y\ f)\ 2 \\
&\rightarrow_{\beta} (\lambda n.ifelse\ (isZero\ n)\ 1\ (mult\ n\ ((Y\ f)\ (pred\ n))))\ 2 \\
&\rightarrow_{\beta} ifelse\ (isZero\ 2)\ 1\ (mult\ 2\ ((Y\ f)\ (pred\ 2))) \\
&\rightarrow_{\beta} (mult\ 2\ ((Y\ f)\ (pred\ 2))) \\
&\rightarrow_{\beta} (\lambda m\ n\ f.m\ (n\ f))\ 2\ ((Y\ f)\ (pred\ 2)) \\
&\rightarrow_{\beta} (\lambda m\ n\ f.2\ (((Y\ f)\ (pred\ 2))\ f)) \\
&\rightarrow_{\beta} (\lambda f.2\ (((Y\ f)\ (pred\ 2))\ f)) \\
&=_{\beta} (\lambda f.2\ (f\ (Y\ f)\ (pred\ 2))\ f)) \\
&\rightarrow_{\beta} (\lambda f.2\ ((\lambda f.\lambda n.ifelse\ (isZero\ n)\ 1\ (mult\ n\ (f\ (pred\ n))))\ (Y\ f)\ (pred\ 2))\ f)) \\
&\rightarrow_{\beta} (\lambda f.2\ ((\lambda n.ifelse\ (isZero\ n)\ 1\ (mult\ n\ ((Y\ f)\ (pred\ n))))\ (pred\ 2))\ f)) \\
&\rightarrow_{\beta} (\lambda f.2\ ((ifelse\ (isZero\ (pred\ 2))\ 1\ (mult\ (pred\ 2)\ ((Y\ f)\ (pred\ (pred\ 2)))))\ f)) \\
&\rightarrow_{\beta} (\lambda f.2\ ((mult\ (pred\ 2)\ ((Y\ f)\ (pred\ (pred\ 2)))))\ f)) \\
&\rightarrow_{\beta} \dots \\
&\rightarrow_{\beta} (\lambda f.2\ ((\lambda f.1\ (1\ f))\ f)) \\
&\rightarrow_{\beta} (\lambda f.2\ (1\ (1\ f))) \\
&= \lambda f.(\lambda f.x.f\ (f\ x))\ (1\ (1\ f)) \\
&\rightarrow_{\beta} \lambda f.(\lambda x.(1\ (1\ f))\ ((1\ (1\ f))\ x)) \\
&\rightarrow_{\beta} \lambda f.(\lambda x.(\lambda x.f\ x)\ (f\ x)) \\
&\rightarrow_{\beta} \lambda f.(\lambda x.(f\ (f\ x))) \\
&= 2
\end{aligned}$$

Definition A.2.9. A fixed point combinator Y is β -reducing if and only if for all $f \in \Lambda$ it holds that $Y\ f \rightarrow_{\beta} f\ (Y\ f)$.

In particular Curry's Y combinator is not β -reducing. As we have seen in the proof of A.2.6 $Y\ f \rightarrow_{\beta} f\ (Y\ f)$ does not hold for Curry's Y combinator since we need one \leftarrow_{β} , i.e. we obtained that $Y\ f$ and $f\ (Y\ f)$ reduce to a common term. Apparently, there is a fixed point combinator by Alan Turing that is β -reducing.

Theorem A.2.8 (Turing's fixed point combinator). *The fixed point combinator $\Theta = (\lambda x.\lambda y.y\ (x\ x\ y))\ (\lambda x.\lambda y.y\ (x\ x\ y))$ by Turing is reducing.*

Proof. We have to show that for all $f \in \Lambda$ it holds that $\Theta\ f \rightarrow_{\beta} f\ (\Theta\ f)$.

Let $n = \Theta f$.

$$\begin{aligned}
 n &= \Theta f \\
 &= (\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) f \\
 &\rightarrow_{\beta} (\lambda y. y ((\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) y)) f \\
 &\rightarrow_{\beta} (f ((\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) f)) \\
 &= f (\Theta f) \\
 &= f n
 \end{aligned}$$

We obtained $\Theta f \rightarrow_{\beta} f (\Theta f)$ as desired. □

With the Θ -operator recursion can be done with \rightarrow_{β} . Consider the faculty function example from above.

$$\begin{aligned}
 fac\ 2 &:= (\Theta f)\ 2 \\
 &\rightarrow_{\beta} f (\Theta f) 2 \\
 &\rightarrow_{\beta} (\lambda f. \lambda n. ifelse (isZero n) 1 (mult n (f (pred n)))) (\Theta f) 2 \\
 &\rightarrow_{\beta} \dots \\
 &\rightarrow_{\beta} \lambda f. (\lambda x. (f (f x))) = 2
 \end{aligned}$$

A fixed point of the identity function $\lambda x. x$ is

$$\begin{aligned}
 &(\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) (\lambda x. x) \\
 &\rightarrow_{\beta} (\lambda x. (\lambda x. x) (x x)) (\lambda x. (\lambda x. x) (x x)) \\
 &\rightarrow_{\beta} (\lambda x. x) (\lambda x. (\lambda x. x) (x x)) (\lambda x. (\lambda x. x) (x x)) \\
 &\rightarrow_{\beta} (\lambda x. (\lambda x. x) (x x)) (\lambda x. (\lambda x. x) (x x)) \\
 &\rightarrow_{\beta} (\lambda x. x) (\lambda x. (\lambda x. x) (x x)) (\lambda x. (\lambda x. x) (x x)) \\
 &\rightarrow_{\beta} \dots
 \end{aligned}$$

or with Θ we obtain

$$\begin{aligned}
 & (\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) (\lambda x. x) \\
 \rightarrow_{\beta} & (\lambda y. y ((\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) y)) (\lambda x. x) \\
 \rightarrow_{\beta} & (\lambda x. x) ((\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) (\lambda x. x)) \\
 \rightarrow_{\beta} & ((\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) (\lambda x. x))
 \end{aligned}$$

Finally, we obtained in both cases an infinite loop. We see in this example that the fixed point combinator does not necessarily return a terminating function.

Generally spoken if a term or a program self-applies itself it can be the case that it will never terminate. Of course not all functions need a fixed point.

A.2.2 Fixed points in cartesian closed categories

Since the untyped λ -calculus has no types we can not directly construct a category corresponding to it having a fixed point property. For this the concept of a category with ω -CPO objects can be useful. At the beginning of the section in A.2.4 we introduced the concept of ω -CPOs. This concept can be defined in the context of a cartesian closed category. This way cartesian closed categories can be used as semantic models for functional programming languages. Nevertheless, the assumption that any object in a cartesian closed category has fixed points is inconsistent with other assumptions on a category that we would like to have (cf. [7]).

Definition A.2.10 (Partially Ordered Object in a Cartesian Closed Category cf.[7]). D is a partially ordered object in a cartesian closed category if and only if for every set $Hom(A, D)$, there is a partial order relation such that for any $f : B \rightarrow A$ and any $g, h : A \rightarrow D$ it holds that if $g \leq h$ in $Hom(A, D)$, then also $g \circ f \leq h \circ f$ in $Hom(B, D)$

Definition A.2.11 (ω -CPO Object in a Cartesian Closed Category cf.[7]). D is an ω -CPO object in a cartesian closed category if and only if D is a partially ordered object and each hom set $Hom(A, D)$ is an ω -CPO.

The following definition can be regarded as a next step after the definition of a continuous function between ω -CPOs in A.2.5 and the example of \mathfrak{P} in A.2.1.

Definition A.2.12 (ω -Continuous Function between ω -CPO Objects cf.[7]). If D and D' are ω -CPO objects, then an arrow $h : D \rightarrow D'$ is ω -continuous, if for any object A and any sequence $g_0 \leq g_1 \leq \dots$ of $A \rightarrow D$ arrows with supremum g , the arrow $h \circ g$ is the supremum of the sequence $h \circ g_0, h \circ g_1, \dots$.

Since D is an ω -CPO object it follows by A.2.10 that if $g_i \leq g_j$ in $Hom(A, D)$ then also $h \circ g_i \leq h \circ g_j$ in $Hom(A, D')$.

Definition A.2.13 (Strict ω -CPO Object cf.[7]). An ω -CPO object D is strict if and only if there is an arrow $\perp : 1 \rightarrow D$ s.t. for every object A and every arrow $f : A \rightarrow D$ it holds that $\perp \circ \langle \rangle_1 \leq f$. 1 denotes the terminal object as in 3.0.17.

Since every object has to have an identity morphism, there is exactly one arrow from any object into the terminal object we conclude that the morphism from the terminal object into the terminal object has to be the identity implying $\perp \circ \langle \rangle_1 = \perp \circ id_1 = \perp \leq f$.

Proposition A.2.9. (cf.[7]) If D is a strict ω -CPO object and $f : D \rightarrow D$ is an ω -continuous arrow, then there is an element $fix(f) : 1 \rightarrow D$ such that $f \circ fix(f) = fix(f)$.

The element $fix(f)$ is the least element of D with this property.

Proof. (cf.[7]) The arrow $\perp : 1 \rightarrow D$ is the least element of D by definition, then $\perp \leq f \circ \perp$. Since f is monotone, $f(\perp) \leq f \circ f(\perp)$. Consequently, $f^n(\perp) \leq f^{n+1}(\perp)$ leading to $\perp \leq f \circ \perp \leq \dots \leq f^n(\perp) \leq f^{n+1}(\perp) \leq \dots$. Then we define $fix(f)$ as the least upper bound of the sequence (supremum), since f is a continuous function ω -CPO objects it preserves the supremum. $fix(f)$ is the least fixed point since for any other fixed point $fix'(f) : 1 \rightarrow D$ also $1 \leq d$ and inductively $f^n(\perp) \leq d$. \square

Bibliography

- [1] J. Lambek, P. Scott, *Introduction to higher order categorical logic*. Cambridge University Press, 1986.
- [2] H. P. Barendregt, *The Lambda Calculus its syntax and semantics*. Elsevier Science Publishers B.V, 1984, Vol. 103.
- [3] P. Selinger, *Lecture notes on the lambda calculus*, 2013. arXiv: 0804.3434 [cs.LG].
- [4] M. H. Sørensen, P. Urzyczyn, *Lectures on the Curry-Howard Isomorphism*. Elsevier Science, 2006.
- [5] A. M. Turing, “Computability and lambda definability,” *The Journal of Symbolic Logic*, Vol 2, Nr. 4, 153—163, Dec. 1937.
- [6] D. Van Dalen, *Logic and Structure*, 4. edition. Springer, 2008.
- [7] M. Barr, C. Wells, *Category theory for computing science*. Prentice Hall New York, 1990, Vol. 1.
- [8] T. Leinster, *Basic Category Theory*. Cambridge University Press, 2014, Vol. 143.
- [9] M. A. Shulman, *Set theory for category theory*, 2008. arXiv: 0810.1279 [math.CT].
- [10] J. Lambek, “Cartesian closed categories and typed λ -calculi,” *Combinators and Functional Programming Languages*, 136–175, 1986.
- [11] E. Zermelo, “Untersuchungen über die Grundlagen der Mengenlehre. I.,” *Mathematische Annalen*, Vol 65, 261–281, 1908.