Computational Type Theory for Philosophers:
A Case Study for the Feasibility of Mechanization

Zur Erlangung des akademischen Grades eines
Bachelor of Arts

vorgelegte Bachelorarbeit
im Fach **Philosophie**
der
Philosophischen Fakultät der Universität des Saarlandes
von

Yannic Muskalla
2570820
Promenadenweg 20
66450 Bexbach
s8yamusk@stud.uni-saarland.de

Erstgutachter: Prof. Dr. Holger Sturm Zweitgutachter: Dominik Kirst

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Ferner habe ich die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, unter Angabe der jeweiligen Quelle als Entlehnung kenntlich gemacht. Dies gilt ebenso für Zeichnungen, Skizzen und Pläne sowie bildliche und grafische Darstellungen, die nicht eigenhändig von mir angefertigt wurden.

Saarbrücken, 10.05.2022, Yannic Muskalla

Acknowledgments

I owe a dept of gratitude to both of my advisors. Prof. Holger Sturm arroused my interest in the philosophy of logic through his teaching and accepted this somewhat unusual project plan as a bachelor's thesis. I am also very grateful for our discussions, for which he always made time throughout the development of this thesis.

My further thanks go to Dominik Kirst who agreed on acting as the second reviewer, making this interdisciplinary project possible in the first place. Additionally, he also kindly agreed on advising me, even though I am very new to the topic of Computational Type Theory. I believe my formal understanding of CTT and logic in general improved vastly through his continuous support.

Abstract

This thesis serves two goals. On the one hand, it introduces the reader to Computational Type Theory as a tool kit for the formalization of reasoning as well as an interesting philosophical theory which is in stark contrast to common conceptions about the nature of logic and mathematical objects. On the other hand, this thesis exemplifies how classical results written in the language of axiomatic set theory can be translated and mechanized in Computational Type Theory, i.e. implemented in a proof assistant. Therefore, we will sketch the benefits of working with a proof assistant and then put our considerations into practice by implementing part of the article *Intuitionistic Modal Logics as Fragments of Classical Bimodal Logics* [1] by Wolter and Zakharyaschev. For our implementation we use Computational Type Theory as the meta logic.

Contents

1 Introduction

Computational Type Theory (CTT) [2] is a relatively modern system for providing a foundation of both mathematics and logic. It unites concepts and positions that could not prevail on their own since axiomatic set theory became the standard. In this chapter I will sketch how these positions historically came about.

Russell's Paradox

In the course of time, mathematics became more and more diverse by including new subdisciplines like topology, group theory etc. Since most subdisciplines also come with their new mathematical objects, the realm of mathematical objects increased rapidly and it became more and more questionable, if the whole mathematical venture was sound and consistent. But to be able to ultimately check the soundness of arguments, one first needs a notion of soundness i.e. a formulation of a series of first principles that justify the further theoretical developments. These first principles then serve as a foundation of the domain. In the late 19th century, mathematicians became especially concerned about the foundations of mathematics and started to formulate them for different domains of mathematics such as arithmetics and geometry. The german mathematician Gottlob Frege [3] famously attempted to give a complete foundation of mathematics as a whole by founding it in logic. His endeavors ultimately failed since his logical axioms contained a paradox which was revealed to Frege in a letter from Bertrand Russell [4]. The paradox in its most famous formulation concerns the class

$$R \coloneqq \{x \mid x \notin x\}$$

of all classes which do not contain themselves. This class cannot be allowed to be a proper set, since both $R \in R$ and $R \notin R$, this would be contradictory. The nonrestrictive set formation of naive set theory, often ascribed to Georg Cantor, did not

prevent this paradox. Frege's system of axioms did allow for a similar contradiction[1] and hence Frege had to resign from founding mathematics in logical axioms. Afterwards, the *foundational crisis of mathematics* broke out and various attempts were made to provide a basis of mathematics that does not fall victim to the paradox.

Type Theory

One of these attempts was published by Russell himself together with his british colleague Alfred North Whitehead in the *Principia Mathematica (PM)* in 1910 [5], a book that was regarded as the benchmark for formal mathematics for years to come. In this act, he did lay down his type theory, i.e. his concept of ordering all mathematical objects into a hierarchy of types. In this theory, a membership relation $a \in A$ is only defined between an element a and an object A whose type is at least one step above a in the type hierarchy.[2] Thus claims of self-membership like $x \in x$ or its negation simply are syntactical errors which are not covered by the definition of the membership relation. Thence the type theory laid down in the PM does not contain the paradox. The class R cannot even be formulated in type theory. So type theory in principle can be seen as a possible way to found mathematics. However, because of some theoretical obscurities[3] and other nonacademic circumstances, Russell's type theory did not prevail on its own.

Intuitionism

Another approach to the foundation of mathematics was given by the intuitionists. The intuitionistic standpoint was mainly represented by its founder L.E.J. Brouwer during the foundational crisis. The main doctrine of intuitionism is that mathematical objects and mathematical proofs are a creation of the (human) mind and thus „the truth of a mathematical statement can only be conceived via a mental construction that proves it to be true"[7]. This is not only controversial from a philosophical point if view, since most philosophers accepted mathematical objects as mind-independent entities

[1] Although Frege didn't use sets in his foundation, his notion of functions runs into the same paradox as the naive notion of sets.

[2] See Section 1.3.1 of [6].

[3] Russell for example wasn't entirely clear about the (in his type theory) ambiguous nature of cardinals. Also it seemed strange that impredicative definitions were first declared illicit to then seemingly get legitimized by his Axiom of Reducibility. See Section 1.3.1 of [6] for details.

in their ontology, as I will briefly explain in 3.1. But it also has far-reaching formal implications like for example the refutation of the law of excluded middle,[4] making intuitionism incompatible with classical mathematics. At first, intuitionism lacked a language for intuitionistic reasoning, due to the fact that Brouwer had concerns about developing such a language. He believed reasoning depends on primal, languageless intuitions and any formal system that tries to represent the mental constructions and the workings of our intuition thus can never truly replace the mental proceedings.[5] Later in the 1930s, Arend Heyting, a student of Brouwer, did provide a formalization of the intuitionistic system. But at this point in history, the quarrel in the mathematical community was already decided against intuitionism and in favor of the well-known axiomatic set theory.

Axiomatic Set Theory

As briefly described before, unrestricted set theory failed, since it did fall vicitim to Russell's paradox. Although the paradox was named after Russell, it was simultaneously but independently known by Ernst Zermelo, a german mathematician. In contrast to Russell, Zermelo did not give up on founding mathematics on sets after learning about the paradox. Instead, he did invent an axiomatization for sets in his acts *Neuer Beweis für die Möglichkeit einer Wohlordnung* [9] and *Untersuchungen über die Grundlagen der Mengenlehre* [10] in 1908. This axiomatization restricts the formation of sets. For example, the separation axiom only provides *subsets* containing the elements that satisfy a first-order property of a given set. Since the class of all classes is not a set in the axiomatization, the separation axiom (as well as the other axioms) does not guarantee the existence of the set R. His axiomatization was supplemented by Abraham Fraenkel and Thoralf Albert Skolem to the *Zermelo-Fraenkel set theory* of today [11]. Set theory claims that all mathematical objects are (reducible to) sets. This implies that a proof of a theorem ultimately traces back to the axioms of set theory, which normally means the principles of Zermelo-Fraenkel set theory plus the axiom of choice (ZFC).

[4] I will come back to this in Section 2.3.4.
[5] See page 23 of [8].

Computational Type Theory

Russell's type theory and Brouwers intuitionism did fade into the background since the universally known Zermelo-Fraenkel set theory displaced them in the second third of the twentieth century. Both theories became relevant again in Computational Type Theory, which unites the concept of types with the constructive logic of intuitionism and is hence also called *Intuitionistic Type Theory*. It was devolped as a formalization of Errett Bishop's constructive mathematics [12] by the Swedish computer scientist Per Martin-Löf in 1972 [2] to provide an alternative foundation of mathematics and logic. In this thesis, I will give a brief introduction to Computational Type Theory and demonstrate its practical capabilities.

Outline

Therefore, I will first explain the technical concepts underlying CTT in Chapter 2. By doing so, we will see the type theoretic roots of CTT at work. In Chapter 3 I will sketch the philosophical background of CTT that mainly originates from intuitionism. CTT provides an alternative approach to understanding the nature of mathematical objects and proofs. Hence, CTT might already be interesting from a genuine philosophical point of view because it can be used to reject both platonism and logicism, two major philosophical theories. Afterwards, I will showcase two major benefits of CTT in Chapter 4 in case one should not be convinced yet that there is any value concerning oneself with it. But even if CTT is convincing from a purely philosophical standpoint or for the pragmatic reasons sketched in Chapter 4, it faces the challenge that most existing work in mathematics and logic is done in axiomatic set theory which is usually combined with classical reasoning. Since it would be completely out of the question to toss all the hitherto results of logic and mathematics out of the window, it is crucial that the recreation of set-theoretic proofs in CTT is smooth. This will be made plausible in Chapter 5 on the basis of a case study about modal logic. The case study also exemplarily shows the feasibility of mechanization, i.e. the possibility of implementing it in software, since most results of Chapter 5 will be implemented in the Coq proof assistant.

2 Basics of Computational Type Theory

In this chapter I will explain the concepts of Computational Type Theory. Because CTT is such a large topic I surely cannot exhaust it completely in this thesis, I will confine my presentation that understanding the outline of the later case study in Chapter 5 should be within grasp of the reader. I recommend [13, 14, 15] for a more sophisticated introduction.

2.1 Terms

Before we can dive into the central topic of types, I will briefly state the way in which the elements of types, which I will call terms,[6] are going to be presented in this thesis.

2.1.1 Lambda Calculus

It is customary to present terms in CTT using the so-called *lambda calculus*, a formal system originally introduced by Alonzo Church [16]. Although it may not be widely spread in philosophy, the lambda calculus is a common tool in computer science to model the computational behavior of functional programming languages.

In the lambda calculus terms are either

Variables $x, y, z, ...$

Abstractions $\lambda x.\, s$ is a term if x is a variable and s is a term.

Applications $s\, t$ is a term if s and t are terms.

An abstraction $\lambda x.\, s$ can be seen as the definition of a function which takes an argument x and returns the body s of the lambda abstraction. For convenience of notation, nested

[6] Note that the distinction between terms and types cannot be stringent in practice since dependent types consist of terms. In practice, many variations of CTT do not make an effort to distinguish between terms and types at all.

abstractions, which are needed to define functions with more than one argument, like $\lambda x.\, \lambda y.\, s$, will be presented as $\lambda xy.\, s$. An application $s\, t$ states that the term t gets applied to the term s.

2.1.2 Reduction via Computation

Because the Type Theory we are dealing with is computational, a key feature is that terms which show the same computational behavior get treated the same. For example, it should not make a difference if $7 + 5$ gets replaced in a term by 12. We would like to state that terms which show identical computational behavior are identical. In this case, we say that two terms are computationally equal. Here, the lambda calculus comes in handy. The lambda calculus provides three so-called reduction rules for terms which correspond to computation steps:

α - **conversion** The computational behavior of a term stays the same if a bound variable gets renamed.

$$\overline{\lambda x.\, s \succ_\alpha \lambda y.\, s[y/x]}^{7}$$

β - **reduction** The computational behavior of an application stays the same if the abstraction to which the argument is applied gets replaced by its body, with the variable substituted by the argument.

$$\overline{(\lambda x.\, s)\, u \succ_\beta s[u/x]}$$

η - **equivalence** A lambda abstraction $\lambda x.\, sx$ reduces to the term s if there are no free instances of x in it.

$$\frac{x \text{ does not occur freely in } s}{\lambda x.\, sx \succ_\eta s}$$

Now we can recursively define reduction ($\succ$) in general as the relation containing the computation steps $\succ_\alpha$, $\succ_\beta$ and $\succ_\eta$ in the complete term as well as in applications and the body of abstractions with the following rules:

$$\frac{s \succ_\alpha s'}{s \succ s'} \quad \frac{s \succ_\beta s'}{s \succ s'} \quad \frac{s \succ_\eta s'}{s \succ s'} \quad \frac{s \succ s'}{s\, t \succ s'\, t} \quad \frac{t \succ t'}{s\, t \succ s\, t'} \quad \frac{s \succ s'}{\lambda x.\, s \succ \lambda x.\, s'},$$

to then define computational equality ($\approx$) a symmetric and transitive relation which contains reduction:

[7] I use the notation $s[y/x]$ to denote the result of substituting every free occurrence of x with y in s.

$$\frac{s \succ t}{s \approx t} \quad \frac{t \approx s}{s \approx t} \quad \frac{s \approx t' \quad t' \approx t}{s \approx t}.$$

This untyped lambda calculus, which I have characterized so far, does not correspond to the possible terms in CTT. In fact, the simply-typed lambda calculus at the core of CTT is more restrictive.[8]

2.2 Types

At the very heart of CTT lies the idea that every term has to be of some type, so for example the number 3 has Type $\mathbb{N}$, the square root-function has type $\mathbb{R} \to \mathbb{R}$ and $\{\{\}\}$ may have type Set. To denote that s is of type T, we write $s : T$. In set-theoretic fashion, it is common to formulate „s is an element of T“ to phrase that $s : T$. I will name types using upper-case letters and terms using lower-case letters to visually distinguish them wherever possible. By giving a type definition, one states how elements of this type can be constructed. I will explain this further with the help of a small example.

2.2.1 Defining Types

To get things going, we need definitions for some types. We normally assume that there are some basic types to which we have access. But let us pretend that there are no types defined yet. We can start by defining the Boolean type $\mathbb{B}$ for example. This type should contain the two values F and T. Because in CTT, every expression has to be of some type, we need to specifiy a type for the Boolean type itself. We put $\mathbb{B}$ in the type universe $\mathbb{T}$, the type of all types. I will not discuss type universes and their workings in this thesis, because it is a sophisticated matter on its own and I don't consider it crucial for the further developments in this thesis. We can now define $\mathbb{B}$ by providing three constructors:

$$\mathbb{B} : \mathbb{T} \tag{2.1}$$

$$\mathsf{F} : \mathbb{B} \tag{2.2}$$

$$\mathsf{T} : \mathbb{B} \tag{2.3}$$

[8] Note for computer scientists: The untyped lambda calculus is Turing-complete; The simply-typed version on the other hand is less expressive and thus not Turing-complete.

We call 2.1 the type constructor of $\mathbb{B}$ since it determines in which type universe the type gets placed. We refer to 2.2 and 2.3 as the value constructors of $\mathbb{B}$ since they state how an element/value of the Boolean type can be constructed. In the Boolean case, the values F and T can be constructed without further requirements.

Function Types

Additional to plain types, there are also function types in CTT. For these, the following rule applies:

If A and B are types, then $\forall a : A.\, B$ is a type as well.

We can classify two versions of function types:

Simple function types where the result type does not depend on the argument variable. In such a case it is convention to write $A \to B$ instead of $\forall a : A.\, B$.

Dependent function types where the result type depends on the argument variable. For example, $\forall a : A.\ p\, a$ would be such a dependent function type, assuming a function $p : A \to \mathbb{T}$.

The elements of a function type $\forall a : A.\, B$ are lambda abstractions $\lambda(a : A).\, b$. Note that the argument of the abstraction now gets assigned a type, unlike described previously in Section 2.1.1. This ensures that functions can only be applied to arguments of the right type. The corresponding concept of type checking will be explained in Section 2.2.3. I will sometimes omit the argument type if it is clear from the context. Furthermore, if a function gets assigned a name F, I will use the well-known function notation $F(a : A) \coloneqq s$ instead of $F \coloneqq \lambda(a : A).\, s$.

2.2.2 Discrimination on Types via Eliminators

Now that we have function types at our hands, we can complete the type definition of our Boolean type by defining an eliminator $\mathtt{E}_{\mathbb{B}}$:

$$\mathtt{E}_{\mathbb{B}} : \forall p : \mathbb{B} \to \mathbb{T}.\, p\,\mathsf{F} \to p\,\mathsf{T} \to \forall b : \mathbb{B}.\, p\, b \tag{2.4}$$

$$\mathtt{E}_{\mathbb{B}}\ p\ p_{\mathsf{F}}\ p_{\mathsf{T}}\ \mathsf{F} \coloneqq p_{\mathsf{F}} \tag{2.5}$$

$$\mathtt{E}_{\mathbb{B}}\ p\ p_{\mathsf{F}}\ p_{\mathsf{T}}\ \mathsf{T} \coloneqq p_{\mathsf{T}} \tag{2.6}$$

This eliminator enables us to define functions that take a Boolean value as an argument and then discriminate between the two Boolean values. It states that a function taking a Boolean as an argument has a return value for all Boolean if it has a return value for the arguments F and T. Its defining equations 2.5 and 2.6 function as additional computation rules, allowing further term reduction. Let us consider Boolean negation as an example for a function definition using the eliminator:

$$Neg_{\mathbb{B}} : \mathbb{B} \to \mathbb{B} \coloneqq \lambda b.\, \mathtt{E}_{\mathbb{B}}\ (\lambda\_ : \mathbb{B}.\, \mathbb{B})\ \mathsf{T}\ \mathsf{F}\ b$$

In this definition we first take an argument b. Then we concern ourselves with adjusting the arguments for the eliminator. We choose as the function p an abstraction taking a Boolean and returning the Boolean type $\mathbb{B}$. To illustrate that we don't make use of the argument we write a blank _ instead of an argument name. Then we set the return value to the opposite of the argument, so $\mathsf{F} : p\,\mathsf{T}$ and $\mathsf{T} : p\,\mathsf{F}$. Finally, we apply the adjusted eliminator to b. By retracing the reduction steps for both possible arguments you can see that the definition produces the expected values:

$$Neg_{\mathbb{B}}\mathsf{F} \succ_{\beta} \mathtt{E}_{\mathbb{B}}(\lambda\_ : \mathbb{B}.\, \mathbb{B})\mathsf{T}\mathsf{F}\ \mathsf{F} \succ_{\text{by 2.5}} \mathsf{T}$$
$$Neg_{\mathbb{B}}\mathsf{T} \succ_{\beta} \mathtt{E}_{\mathbb{B}}(\lambda\_ : \mathbb{B}.\, \mathbb{B})\mathsf{T}\mathsf{F}\ \mathsf{T} \succ_{\text{by 2.6}} \mathsf{F}$$

Since none of the value constructors of $\mathbb{B}$ needs an argument, the corresponding eliminator $\mathtt{E}_{\mathbb{B}}$ is also very basic. Later in Section 5.2.1 we will see an example where the eliminator not only provides a case distinction for the different values of the type, but also supplies us with a hypothesis for recursive function definitions and inductive proofs.

2.2.3 Type Checking

Now that we know about types and type definitions, I will explain how one can check whether a term has the denoted type. This is especially interesting since a motivation for using a typed calculus is to circumvent nonterminating recursion by limiting function applications to a predetermined argument type. Thus, the ability to check the argument type is of utmost importance.

Instead of assuming that no types are assigned yet, we formulate the rules of type checking in a context Γ to be more flexible. This context can be considered as a list of already known type assignments (e.g. $\Gamma = [s_1 : T_1, s_2 : T_2, s_3 : T_3, ...]$). With this in

mind, we can formulate typing judgments $\Gamma \vdash s : T$ using the following three typing rules:

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ \mathbf{\Gamma} \qquad \frac{\Gamma, (x : T) \vdash s : V}{\Gamma \vdash \lambda(x : T).\, s : \forall x : T.\, V}\ \boldsymbol{\lambda} \qquad \frac{\Gamma \vdash s : \forall x : T.\, V \quad \Gamma \vdash t : T}{\Gamma \vdash s\, t : V[t/x]}\ \boldsymbol{App}$$

To make use of the context, we provide the $\mathbf{\Gamma}$-rule which allows us to conclude $x : T$ if the assignment already resides in the context. The $\boldsymbol{\lambda}$-rule states that an abstraction $\lambda(x : T).\, s$ has the function type $\forall x : T.\, V$ if s is of type V in the context extended by the assumption that the argument x has type T. A function application $s\, t$ is, according to the ***App***-rule, of type V if the return type of the function is V and the type of the argument t matches the argument type. For dependent function types we of course need to adjust it by substituting the argument variable with the argument.

As a first example of type checking, let us consider the identity function for Boolean $Id_{\mathbb{B}} \coloneqq \lambda(b : \mathbb{B}).\, b$ and the context $\Gamma \coloneqq [\mathsf{F} : \mathbb{B},\ \mathsf{T} : \mathbb{B},\ \mathbb{B} : \mathbb{T}]$ in which the information about our Boolean constructors is stored. We can check that the application $Id_{\mathbb{B}}\mathsf{T}$ has type $\mathbb{B}$ by using the three typing rules:

$$\dfrac{\dfrac{\dfrac{(b : \mathbb{B}) \in \Gamma, (b : \mathbb{B})}{\Gamma, (b : \mathbb{B}) \vdash b : \mathbb{B}}\ \mathbf{\Gamma}}{\Gamma \vdash Id_{\mathbb{B}} : \forall (b : \mathbb{B}).\, \mathbb{B}}\ \boldsymbol{\lambda} \qquad \dfrac{(\mathsf{T} : \mathbb{B}) \in \Gamma}{\Gamma \vdash \mathsf{T} : \mathbb{B}}\ \mathbf{\Gamma}}{\Gamma \vdash Id_{\mathbb{B}}\mathsf{T} : \mathbb{B}}\ \boldsymbol{App}$$

Such as terms are always considered under computational equality, we also allow type reduction as formalized in the *conversion rule*:

$$\frac{\Gamma \vdash s : V \quad T \approx V}{\Gamma \vdash s : T}$$

Types should be unique, i.e. an expression has exactly one type up to conversion:[9]

$$\Gamma \vdash s : T \quad \text{and} \quad \Gamma \vdash s : V \quad \Rightarrow \quad T \approx V.$$

Furthermore, computation preserves the typing of an expression. Intuitively, if $7+5 : \mathbb{N}$ then $12 : \mathbb{N}$ or more generally:

$$\Gamma \vdash s : T \quad \text{and} \quad s \succ s' \quad \Rightarrow \quad \Gamma \vdash s' : T.$$

[9] This rule does not necessarily apply to types that represent a type universe. As I said before, I decided to not touch on the topic of type universes here.

Now it should be quite clear, that even if we usually don't do type checking explicitly on paper and rather depend on the intuition that the typing is sound, our intuition has this formal notion of type checking as a backup. I will come back to this in Section 2.3.3.

2.3 Logic

Up to this point, I have sketched how types can be defined and how a typed calculus works. One can probably imagine how mathematical objects like numbers, relations etc. can be constructed within this theory even if I did not provide examples except for Boolean.

But we are not reviewing CTT for its mathematical capabilities. The most relevant aspect of CTT for philosophy, namely constructive logic, didn't get touched on yet. In the following section, I will present how logic and logical propositions are usually handled in CTT. Along the way, we will see how all previously sketched aspects are linked.

There are a few ways to present the formal definitions in this section. I have chosen to commit myself to the notation used in [13]. Most of the following type definitions are also provided in [13].

2.3.1 Curry-Howard Isomorphism

The main idea underlying computational logic is to regard logical propositions as types. Objects of such a type act as proofs for propositions. This correspondence between objects and proofs as well as types and propositions is called *Curry-Howard Isomorphism*.

The constructivists perceived propositions as expectations for a proof [8, p. 24]. So uttering a proposition leaves the listener with a demand for a proof. Consequently claiming a proposition P means claiming being able to prove P. Proving P means constructing an object of type P, hence the term *constructive logic*.

2.3.2 BHK Interpretation

To implement the idea of the Curry-Howard Isomorphism, we need to arrange types which correspond to logical propositions. These types get placed in the universe $\mathbb{P}$ of all propositions. The value constructors of these types then function as guidelines

for proving a proposition. The usual way of doing this is called *Brouwer-Heyting-Kolmogorov interpretation (BHK)*:

Table 2.1: Brouwer-Heyting-Kolmogorov interpretation of propositions

Logical structure	**Type**	**Proof term(s)**
Absurdity	$\bot$	$\diagdown$
Implication	$P \to Q$	$\lambda(p : P).\, q$
Conjunction	$P \wedge Q$	a pair consisting of p and q
Disjunction	$P \vee Q$	a decision for p or q
Universal quantification	$\forall(x : X).\, p\, x$	$\lambda(x : X).\, (c : p\, x)$
Existential quantification	$\exists(x : X).\, p\, x$	a pair consisting of x and $(c : p\, x)$

Absurdity (or falsity) gets represented through as an empty type $\bot$ without a value constructor since it should not be provable. However, we provide an eliminator $\mathtt{E}_\bot$ to implement *ex falso quodlibet*:

$$\begin{aligned} \bot &: \mathbb{P} \to \mathbb{P} \\ \mathtt{E}_\bot &: \forall(P : \mathbb{P}).\, \bot \to P \end{aligned}$$

This allows us to get a proof of any proposition if we have a proof of falsity on our hands.

A simple function type $P \to Q$ interprets an implication with an antecedent P and a consequence Q. So functions taking an argument of type P and then returning an object of type Q count as proofs for the implication.

$$\begin{aligned} \wedge &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\ \mathtt{C} &: \forall(P\, Q : \mathbb{P}).\, P \to Q \to P \wedge Q \\ \mathtt{E}_{\wedge l} &: \forall(P\, Q : \mathbb{P}).\, P \wedge Q \to P \\ \mathtt{E}_{\wedge r} &: \forall(P\, Q : \mathbb{P}).\, P \wedge Q \to Q \end{aligned}$$

Conjunctions are interpreted as a conjunction type $P \wedge Q$, whose constructor $\mathtt{C}$ demands for an element of P and an element of Q, returning an element of type $P \wedge Q$. So we can prove a conjunction if we can construct an object for both conjuncts. The eliminators

$\mathtt{E}_{\wedge l}$ and $\mathtt{E}_{\wedge r}$ allow us to infer both P and Q from a conjunction $P \wedge Q$.

$$\begin{aligned}
\vee &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\
\mathtt{L} &: \forall (P\, Q : \mathbb{P}).\, P \to P \vee Q \\
\mathtt{R} &: \forall (P\, Q : \mathbb{P}).\, Q \to P \vee Q \\
\mathtt{E}_{\vee} &: \forall (P\, Q\, R : \mathbb{P}).\, P \vee Q \to (P \to R) \to (Q \to R) \to R
\end{aligned}$$

Disjunctions are provided via a type with type constructors `L` and `R` that allow us to prove a disjunction $P \vee Q$ if we have a proof for P or Q. On the other hand, if we already have an object of type $P \vee Q$ and we have for both types P and Q a function taking such an element and returning an element of type R, the eliminator $\mathtt{E}_{\vee}$ provides us with an element of type R. I.e. speaking in the language of reasoning: if we have our hands on a disjunction $P \vee Q$ and know that P as well as Q imply R, we can infer R.

We have already used the symbol $\forall$ for dependent function types. So it should be no surprise that propositions that start with a universal quantification are interpreted according to BHK as dependent function types and are considered proven when a dependent function of that type is constructed.

$$\begin{aligned}
\exists &: \forall (X : \mathbb{T}).\, (X \to \mathbb{P}) \to \mathbb{P} \\
\mathtt{Ex} &: \forall (X : \mathbb{T})\, (p : X \to \mathbb{P})(x : X).\, p\, x \to \exists x.\, p\, x \\
\mathtt{E}_{\exists} &: \forall (X : \mathbb{T})(p : X \to \mathbb{P})(Q : \mathbb{P}).\, (\exists x.\, p\, x) \to (\forall x.\, p\, x \to Q) \to Q
\end{aligned}$$

The BHK interpretation regards an existential quantification $\exists x.\, p\, x$ as a statement which is only provable if a concrete *witness* x is provided as well as a proof for $p\, x$, the so-called *certificate*. The eliminator allows us to infer a proposition Q from an existential quantification if we have a function that takes a witness and the corresponding certificate and returns an object of type Q.

The core idea of the BHK-interpretation is summarized in Table 2.1. One might wonder how negation is implemented since it was not mentioned yet. Negation is not considered a primitive in CTT. Instead, it is regarded as a notational abbreviation for an implication to absurdity:

$$\neg P \coloneqq P \to \bot.$$

So proving $\neg P$ means providing a proof of falsity in case a proof for P is given.

2.3.3 Proving and Proof Checking

Now that we know how logical propositions are interpreted in CTT, let us exert that knowledge on an example.

Consider *contraposition*, a well-known logical theorem. The BHK interpretation of contraposition for two propositions P and Q results in

$$(P \to Q) \to \neg Q \to \neg P.$$

So we interpret contraposition as a nested function type. Functions of this type take three arguments:

- a function f of type $P \to Q$,
- a function g of type $Q \to \bot$,
- an object p of type P,

and return a proof of falsity. If we want to prove the contraposition, we have to construct such a function. So after we take the arguments, we use them in the body of the lambda abstraction to construct an object of falsity. Since function g returns such an object, we want to feed g with an argument of type Q. An argument of type Q is provided by f, in case we can supply f with an argument of type P. Luckily, we have an object of type P, namely p. If we assemble those steps, we obtain the proof $\lambda f g p.\, g(fp)$. You can see an exemplification of the proof term in Figure 2.1.

$$\begin{array}{llll} (P \to Q) \to & (Q \to \bot) \to & P \to \bot & \\ \lambda f & \lambda g & \lambda p & g(fp) \end{array}$$

Figure 2.1: A proof term for contraposition.

Intuitively, it should be clear that $\lambda f g p.\, g(fp)$ is indeed a proof term for $(P \to Q) \to \neg Q \to \neg P$. Nonetheless, it is quite simple to check whether we obtain a proof of a proposition. Since proofs of propositions are just elements of the proposition, proof checking is type checking and I already explained in Section 2.2.3 how type checking works.

Before I focus on our example again, I want to quickly point out the following similarity. In Figure 2.2, one can see that the ***App***-rule for simple function types looks

like *modus ponens* if you only consider the types.

$$\frac{s : A \to B \quad t : A}{s\,t : B} \qquad\qquad \frac{A \to B \quad A}{B} \text{ without terms}$$

Figure 2.2: ***App***-rule for simple function types

This illustrates that the BHK-interpretation for propositions suits our logical intuition and precognition. Propositions and computational objects get bound together as statements and proofs in a way that makes us intuitively comply with it. You can see that in our example everything works out just as expected:

We have to check the soundness of the typing judgment $[P\,Q : \mathbb{P}] \vdash \lambda(f : P \to Q)(g : Q \to \bot)(p : P).\, g(fp) : (P \to Q) \to \neg Q \to \neg P$.

$$\dfrac{\dfrac{\dfrac{(g : Q \to \bot) \in \Gamma}{\Gamma \vdash g : Q \to \bot}\,\mathbf{\Gamma} \quad \dfrac{\dfrac{(f : P \to Q) \in \Gamma}{\Gamma \vdash f : P \to Q}\,\mathbf{\Gamma} \quad \dfrac{(p : P) \in \Gamma}{\Gamma \vdash p : P}\,\mathbf{\Gamma}}{\Gamma \vdash fp : Q}\,\textbf{\textit{App}}}{\Gamma \vdash g(fp) : \bot}\,\textbf{\textit{App}}}{[P\,Q : \mathbb{P}] \vdash \lambda(f : P \to Q)(g : Q \to \bot)(p : P).\, g(fp) : (P \to Q) \to \neg Q \to \neg P}\,\boldsymbol{\lambda} \times 3$$

Type derivation for $\lambda fgp.\, g(fp)$
$\Gamma \coloneqq [P\,Q : \mathbb{P},\, f : P \to Q,\, g : Q \to \bot,\, p : P]$

As already mentioned in Section 2.2.3, it is very uncommon to build such derivation trees by hand. Normally, we rely on our intuition or make use of the automatic type/proof checking capability of proof assistants as I will explain in Section 4.1.2.

2.3.4 Law of Excluded Middle

In the previous segment, I have characterized computational logic as a constructive logic that regards proofs as computable objects and functions, which can be formulated as lambda terms. A major consequence of this property is that the so-called *law of excluded middle* (LEM), often traded in philosophy under the name *tertium non-datur*, is not provable. LEM states that for every proposition, either the proposition itself or

its negation holds. So its formulation

$$\forall(P : \mathbb{P}).\, P \vee \neg P$$

is considered a tautology in classical logic. But in CTT, this can not be the case, since, according to BHK, LEM gets interpreted as „There is a function which takes any proposition as an argument and returns a proof of this proposition or its negation.“ It is clear that there can't be such a function, since CTT is definitely powerfumathcalBl enough so that the first *incompleteness theorem* of Gödel applies.[10] It says that any formal system equipped with enough power contains a statement which can neither be proved nor disproved. Informally, this is a clear refutation of LEM in CTT. The most common visualization that excluded middle cannot be proven in CTT works by defining a logical subsystem, in which we can easily show equivalence of logical deduction and evaluation to truth-values. We then use a mulitvalent evaluation to disprove LEM in that subsystem.[11][12]

Nonetheless, we can just assume LEM if we require it. Even if there is no proof of LEM in general, it does no harm to assume such a proof. In fact, consistency under the assumption of excluded middle is an important feature of CTT. It allows us a much better understanding of where exactly LEM is needed by making every use of it explicit. Not every proof classically conducted as a *reductio ad absurdum* has to be done in a nonconstructive way.

To close this chapter, I would like to take a look at the reverse direction of contraposition:

$$(\neg Q \to \neg P) \to P \to Q. \tag{2.7}$$

There is no chance to construct a proof term for this proposition without the law of excluded middle.[13] For this reason, we assume a function $\mathcal{D} : \forall(P : \mathbb{P}).\, P \vee \neg P$ (i.e. a proof for LEM). We then continue as usual by taking the arguments $f : \neg Q \to \neg P$ and $p : P$ to construct a proof for Q. The application $\mathcal{D}\, Q$ provides us with the disjunction $Q \vee \neg Q$, which we can convert with $\mathtt{E}_\vee$ to a proof of Q in case we can construct two

[10] Even if I didn't cover it in this thesis, natural numbers as well as equality can be formulated in CTT, which already suffices to fall under the grasp of Gödel's theorem.

[11] The proof can be found in Chapter 28.9 of [13].

[12] Of course there are instances of LEM that can be proven, e.g. for every negated formula $\neg P$ it can be easily shown that $(\neg P) \vee \neg(\neg P)$. Such instances are called *deciders*. Usually, such deciders are formulated in $\mathbb{T}$ instead of $\mathbb{P}$. A great part of research of intuitionistic logic is concerned with finding deciders. Since CTT is computational, a decider is at the same time a proof for the computability of the decision.

[13] or other assumptions

functions $Q \to Q$ and $\neg Q \to Q$. The first one is just the identity function. For the second one, we take an argument $g : \neg Q$ and then construct a proof of falsity by feeding f with the arguments $g : \neg Q$ and $p : P$. Finally, we apply $\mathtt{E}_\bot$ to transform the proof of falsity to a proof of Q. These steps result in the proof term

$$\lambda(f : \neg Q \to \neg P)\,(p : P).\,\mathtt{E}_\vee\,Q\,(\neg Q)\,Q\,(\mathcal{D}\,Q)\,(\lambda(q : Q).\,q)\,(\lambda(g : \neg Q).\,\mathtt{E}_\bot\,Q\,(f\,(g\,p))). \tag{2.8}$$

Although it is convenient that we can assume LEM without breaking our logical system, we should restrain ourselves from unnecessary usage to not give away the crucial benefit of CTT I will sketch in Section 4.2.

2.4 Roundup

We have seen in this chapter that the Computational Type Theory interprets logical propositions as types. The values of types (which we called terms), can be written down in the simply-typed lambda calculus. The Curry-Howard Isomophism considers values of propositions as proofs for the propositions. The Brower-Heyting-Kolmogorov interpretation of propositions in unison with the typing rules allows an intuitive notion of proof checking via type checking.

After this technical introduction to Computational Type Theory, we will now change focus to the philosophical viewpoint in the upcoming chapter.

3 Philosophical Implications

Since it would not fit within the scope of this thesis to cover the philosophical implications of Computational Type Theory to the full extent, I will confine myself to only explain why CTT conflicts with two major philosophical theories in the philosophy of logic and mathematics: Platonism and Logicism.

3.1 Platonism

In the following section, while giving a definition of platonism, I will explain the philosophical thoughts underlying CTT. Because there are many slightly varying definitions of mathematical platonism, we will stick for our purpose with the concise definition from [17]:

Mathematical platonism is the claim that

- mathematical objects exist.
- mathematical objects are abstract.
- mathematical objects are independent.

Now I will discuss how these three different claims spell out and apply to CTT.

Existence

That mathematical objects exist means that there is at least one mathematical object. That this is true for CTT is fairly obvious. Since mathematical objects, like numbers, functions and so on, are no other than types and members of them in CTT, the claim reduces to the proposition that the type universe of mathematical objects is inhabited. For this proposition we can easily construct a proof by quoting a type in that universe. One can argue that the universe of mathematical objects is $\mathbb{T}$, so $\mathbb{B}$ would be a proof for the proposition, since $\mathbb{B} : \mathbb{T}$ by (2.1).

Independence

The most common way to spell out the independence posit of platonism works with counterfactuals. If mathematical objects are independent, then „had there not been any intelligent agents, or had their language, thought, or practices been suitably different, there would still have been mathematical objects.“[14] The platonic view is that the existence of mathematical objects is not bound by human consciousness and reasoning. This position gets often (but not necessarily) supplemented with the thesis that every logical proposition has an independent truth-value. So if we prove a statement for the first time, we add nothing to the statement on an ontological level. We just gain inside about the already fixed truth-value.

Computational Type Theory stands in tradition of the intuitionistic philosophy of Brouwer and thus rejects that posit. Mathematical objects are understood in CTT as constructions inside the human mind and nothing beyond that. So the mathematician does not operate with objects already provided by nature, but rather he creates the needed objects. This creative process of doing mathematics starts with formulating types, which act as blue prints for building mathematical objects, because they contain construction rules (constructors) for objects. Then one can follow the constructors and construct objects of the type. As a conclusion, mathematical objects would never have been created in a world without creatures concerned with mathematics, and hence the existence of mathematical objects is subject dependent.

However, even though mathematical objects are not subject independent, they are still language independent. Brouwer describes his *first act of intuitionism* in his lectures as the process of „completely separating mathematics from mathematical language [...] recognizing that intuitionistic mathematics is an essentially languageless activity of the mind“.[15]

I don't want to conceal that this notion of mathematical objects as mind-dependent entities comes with a bunch of philosophical difficulties. For example, it needs to be taken care of that our mental constructions are intersubjective. If two mathematicians construct, let's say the natural number 3, it is, for obvious reasons, strongly required that the two constructed numbers share the same properties and are thus computationally equal. Also the intuitionist needs to ensure that even if the constructions are subjective, it is not up to the subject to decide whether a construction is mathematically sound. Elsewise this would open the door for psychologizing mathematics.

[14] SEP, Platonism in the Philosophy of Mathematics [17], 4.1.
[15] [18], S. 4

He needs a way to call someone out on erroneous constructions. Brouwer intended to solve these issues by the introduction of the *Creating Subject*, an „idealized mind in which mathematics takes place already abstracts away from inessential aspects of human reasoning such as limitations of space and time and the possibility of faulty arguments.“[16]

Accordingly, by doing mathematics (in a proper way) we partake in the Creating Subject which guarantees for intersubjective and sound constructions. Of course, this attempt by Brouwer to solve the difficulties originating in the dependency of mathematical objects raises further questions. These questions are very similar to the ones raised by platonism. Instead of worrying about the nature of independent abstract mathematical objects and how our epistemic access to them works we now are concerned with the nature of the Creating Subject and the process of partaking in it, in case we follow Brouwer's approach.

Clearly, these problems that I mentioned require further philosophical reflection. Since it is not the topic of this thesis to conduct this discussion, I will move on to the last posit of platonism, namely abstractness.

Abstractness

According to the claim of platonism, mathematical objects are abstract in the sense that they are not spatiotemporally anchored and therefore not causally efficacious. Whether this claim also applies in CTT is debatable in my opinion. At least, the properties of mathematical and logical objects can change with time. Since asserting that a proposition is true means asserting that I have a proof for it, logical propositions (about mathematical objects) become true the moment I construct a proof term for them. So in contrary to a common platonist standpoint, whenever we prove a statement, we cause an ontological change by constructing a proof term. And this term doesn't give us insight about an already fixed truth-value but changes the verity of the proposition. What does this say about abstractness? Mathematical objects have to be constructed at some point and their existence is hence temporally dated. In addition, at least one property of propositions, the verity, can change with time. Whether mathematical objects have a spatial coordinate as mental files is a discussion I would rather not invoke in this thesis. Let us for now conclude that it isn't entirely clear, whether mathematical objects count es abstract entities in CTT.

[16] See [7, Section 2.2]

Conclusion

In summary it is quite clear, that Computational Type Theory disagrees with platonism at least about the independence of mathematical objects. If CTT classifies as a theory within object realism, i.e. as a theory that asserts existence as well as abstractness of mathematical objects, depends on the exact way to lay out abstractness.

3.2 Logicism

In the following, I will give a basic definition of logicism. I will use this section to then point out the relationship of logic and mathematics in CTT.

> Logicism is the claim that all mathematical truths form a species of logical truth. The axioms/first principles of mathematics hence must be derivable from logic.[17]

Sometimes, a weak version of logicism is distinguished from this definition [6], stating that the above definition only applies to all *provable* mathematical truths.

As I already explained in the previous section, in CTT a statement is only considered as true the moment a proof construction is given. So the two versions of logicism do not differ from the viewpoint of CTT because there can not possibly be true propositions that are not provable. But let us come back to the general relation of mathematics and logics.

As one can see in the definition, logicism proposes to ground mathematics in logic. Gottlob Frege, for example, tried to come up with logical axioms that are intuitively sound and strong enough to cover mathematics and mathematical reasoning. This concept of regarding mathematics as a subdiscipline of logic is highly incompatible with CTT. One can argue that CTT is practically a counter-draft to logicism. Logic is embedded in the type universe of mathematical objects. The Curry-Howard-Isomorphism (2.3.1) allows CTT to handle logic the same way as mathematics. Propositions are treated as types and proofs are treated as constructions of types. So the procedure does not differ whether I construct a mathematical object or I prove a proposition. In both cases I apply the constructors of the type definition and use the reduction methods provided by the simply-typed lambda calculus. So the only way of formally differentiating between regular constructions and proofs is to look up the type of the

[17] See [6] for reference.

constructed term. Propositions are placed in the universe $\mathbb{P}$. Whenever a type is an element of $\mathbb{P}$, we speak of it as a proposition. It is just a convention to put certain types in $\mathbb{P}$ and some others not. Theoretically, it would do no harm to put the type of the natural numbers $\mathbb{N}$ in the universe of propositions and hence regard e.g. the number 3 as a proof for $\mathbb{N}$.[18]

CTT furnishes the claim that logic can formally be reduced to mathematics, i.e. types and mathematical constructions. Logic is just another perspective to the universe of mathematics. It adds nothing genuine logical to it but a different way of speaking, using the common logical vocabulary of proofs, propositions, sentential connectives, etc.

Conclusion

In CTT, logic is a subdomain of mathematics. This is expressed by the placement of the universe of logical propositions in the universe of mathematical objects $\mathbb{P} : \mathbb{T}$. To come back to the definition of logicism, one can see that

> CTT claims that all logical propositions form a species of types in the type universe of mathematics. Logic avails itself of the construction mechanism for mathematical objects. The logical truths stated in propositions hence must be constructible in the same way as mathematical entities.

Remarks

Before we move on to the next chapter, where I will explain two benefits of CTT, I have two final remarks concerning the philosophical implications:

In Section 2.3.4, I have argued from a technical point of view that the law of excluded middle is not provable and thus does not hold in CTT. That LEM is not valid in generality can also be apprehended from an ontological standpoint. In CTT, mathematical objects and logical proofs are mental constructions in the human mind. The acceptance of LEM would entail that such mental constructions/objects would have to abruptly begin to exist in my mind the moment I have shown that the negation is contradictory. Since this would be quite strange, it is ontologically stringent that CTT does not contain LEM.

[18] In practice, there often are some restrictions when working in the universe of propositions. The CTT of Coq for example imposes an elimination restriction on propositional types. This prevents certain inconsistencies in connection with assuming excluded middle [13, Section 4.3] which I will not touch on in this thesis.

Also, we have to remind ourselves that this chapter only showed what the main philosophical implications are and that they contradict platonism and logicism. To make a proper argument for the philosophy of CTT, one would need to point out why it is superior to the alternatives like platonism, formalism etc. respectively why the alternatives are lacking in important aspects, but this is beyond the scope of this thesis.

4 Two Major Benefits

4.1 Support by Proof Assistants

As I mentioned earlier, Computational Type Theory combines the constructive approach to logic and mathematics from intuitionism with Type Theory. This combination is well suited for implementing it in a proof assistant which allows for the mechanization of results. In this chapter I will sketch what a proof assistant, like e.g. Coq [19], Agda [20] or Lean [21] is and which aspects of CTT cause this fitness for mechanization.[19]

This section is of utmost importance for the upcoming case study in Chapter 5, since the case study is supposed to demonstrate the practical feasibility of mechanization using the Coq Proof Assistant.

4.1.1 Proximity to Programming Languages

The biggest reason why CTT is well suited for software mechanization is the vast conceptual similarity to programming languages.

On the one hand, the lambda calculus, which represents the terms in CTT (as explained in 2.1), is also a common model for the computational behavior of functional programming languages. One can even regard functional programming languages as an implementation of the lambda calculus. So the possibility of implementing the terms of CTT is obviously given. Furthermore, this allows for an implementation style that we are already accustomed with in functional programming.

On the other hand, typing is a concept also used in programming languages, where upfront static type checking prevents runtime errors. This leads us to the first benefit of using a proof assistant.

[19] There are also a handful of set-theoretic proof assistants like Metamath [22] or [23], but they are way less popular.

4.1.2 Automatic Type Checking

The proof assistant does type checking automatically. Since according to the Curry-Howard isomorphism, type checking is also proof checking, the proof assistant is also able to check whether a given term is a sound proof for a logical proposition. Type checking is automatable because it is algorithmically decidable in every instance. This follows intuitively from two properties of the typing rules (2.2.3): Firstly, the typing rules are unique, i.e. in every case only one rule is applicable. Secondly, the application of a rule always reduces the structure of the term. This ensures termination.

So the most basic way to make use of a proof assistant for theorem proving would be to write down a typing judgment of a proof term and enjoy our exemption from the tedious labor of proof checking. To give an example, we can check whether $\lambda(f : P \to Q)(g : \neg Q)(p : P).\, g(fp)$ is a proof of $(P \to Q) \to \neg Q \to \neg P$ by compiling the Coq program

$$Check(fun(f\, g\, p) => g(f\, p)) : (P \to Q) \to not\ Q \to not\ P.$$

So we don't have to build up the inference tree in 2.3.3 by ourselves or rely on our intuition. This also has the advantage of ensuring correctness of our proof terms, since there is no more room for human error.[20]

4.1.3 Step-wise Term Construction

The benefit of automatic type checking is already remarkable, but a proof assistant has further advantages. Whenever we do a more involved proof on paper, we often are not able to come up with a proof term right away. We rather formulate single steps which intuitively lead to a term of the consequence of a function type from terms of the antecedence. An example for this approach is the proof of the reverse direction of contraposition in 2.7. The problem is that we have to translate these steps into a proof term anyway, at least if we want to type check our proof. This can be quite straining, since the length and complexity of the proof term increases at least linear with the length and complexity of the proposition. You can get a first apprehension of this problem by looking at the proof term 2.8 of 2.7.

Luckily, proof assistants also do help out in these cases. Instead of letting the proof assistant check a term we already constructed, we can describe the different steps to the proof assistant with so-called *tactics*. The proof assistant then automatically interprets

[20] There is no room for it in type checking itself. At most, there is room for human mistakes in writing the type checking routine.

Lemma 1 (Reverse Contraposition). *Let P and Q be some propositions. It holds that*

$$\mathsf{LEM} \to (\neg Q \to \neg P) \to P \to Q.$$

Proof.

Proof steps in natural language	**Coq tactics**
We assume a proof $\mathcal{D} : \forall(P : \mathbb{P}).\, P \vee \neg P$ for LEM.	intros $\mathcal{D}$.
We take $f : \neg Q \to \neg P$ and $p : P$ as arguments to construct a proof for Q.	intros $f\, p$.
The application $\mathcal{D}\, Q$ then provides us with the disjunction $Q \vee \neg Q$, which we can convert with $\mathtt{E}_\vee$ to a proof of Q in case we can construct two functions $Q \to Q$ and $\neg Q \to Q$.	destruct $\mathcal{D}\, Q$ as $[q\|g]$.
The first one is just the identity function.	exact q.
For the second one, we take an argument $g : \neg Q$ and then we construct a proof of falsity...	exfalso.
... by feeding f with the arguments $g : \neg Q$ and $p : P$.	apply $(f\, g\, p)$.

■

Figure 4.1: An example of step-by-step proving with the Coq proof assistant. The description in the left column originates from Section 2.3.4. The right column contains the corresponding Coq tactics.

the tactics and constructs the proof term. So when working with a proof assistant, we don't need to concern ourselves with the lowest formal level of CTT. We just describe the proof step-by-step and let the assistant handle the construction as well as proof checking. Figure 4.1 shows an example of translating proof steps into tactics.

4.1.4 Proof Automation

The two previously described levels of automation by proof assistants, automatic type checking and automatic tactic interpretation, get complemented by a third level of automation that I briefly want to mention. Proof assistants usually provide automa-

tion tactics, which prompt the assistant to automatically construct a proof term. Although these automation tactics currently only work on a limited scale of situations, it is a very convenient tool in practice to quickly deal with trivial parts of proofs.

In summary, the possibility of mechanizing allows for a vastly improved proving practice, because type checking as well as term construction is taken care of by the assistant and because certain proofs can even be done automatically. In addition, mechanization also comes in handy for the presentation of results. When presenting a non-mechanized theorem, you always have to decide: Giving a high-level proof description results in easy readability but possibly leaves the reader in doubt about its soundness, i.e. it raises the question whether the description really reduces to the axioms of the applied logic (e.g. the ZF-axioms); Giving a low-level explanation touching the foundations may increase the credibility of soundness but also stretches the explanation and massively downgrades the readability. This problem does not apply to mechanized results. The proof description can be carried out on a very high level, ensuring a pleasant read without any drawback, because the soundness is guaranteed by the proof assistant and the actual code ideally available to the reader. You can see in Chapter 5 how this is done in practice.

4.2 Proofs as Algorithms

Another benefit of CTT directly stems from the BHK interpretation of propositions. Since an implication is interpreted as a function type $P \to Q$, its proof, the function $\lambda(p : P).\, q$, can be used as an algorithm. It computes for every p of type P a corresponding q. So whenever we seemingly prove an abstract logical property, we give a concrete algorithm. The proof term of an implication takes an object of the antecedent as argument and computes an object of the consequence. A proof for a conjunction computes a pair consisting of an element for each of the conjuncts. The construction of a disjunction is at the same time an algorithm that computes an element of one part of the disjunction. The most interesting case is probably existential quantification. In classical logic, the proof of an existential quantification $\exists x.\, px$ is often non-constructive, i.e. instead of providing an x as well as a construction for px, the inconsistency of $\forall x.\, \neg px$ gets shown. Such a proof only informs us that there is indeed an object with the property p, but it does not tell us exactly which object it is. A constructive proof on the other hand not only ensures that there is such an object, but

it even provides the object for which the property holds. For example, a proof of the proposition that for every pair of natural numbers there is a least common multiple is at the same time an algorithm that computes the least common multiple of two natural numbers.

In Section 2.3.4 I already foreshadowed that there is a downside of assuming LEM in CTT. To explain this, let us suppose we want to prove an implication $P \to Q$ under the assumption of LEM. So we construct an element of $\mathsf{LEM} \to P \to Q$. Sadly, we cannot make use of the proof as an algorithm that computes a $q : Q$ for an argument $p : P$. Because any proof term for the proposition has to be a λ-abstraction $\lambda(f : \mathsf{LEM})(p : P).\, q$ taking as arguments not only p but also a proof of excluded middle, i.e. a function f deciding every proposition, the algorithm needs such a function as input to compute q. We have already argued that the existence of this f would be highly problematic.

However, as long as we do not rely on LEM, our proofs can be exploited as an algorithm. This is especially interesting for computer science. The algorithms that are generated through proofs are already guaranteed to behave according to their specification. In the previous section, we have seen that CTT can be implemented in software. So if there is a way to extract algorithms defined through proofs in CTT as regular program code, it will be most useful as a tool to generate verified programs that behave (at least regarding the output) exactly as expected. This is especially relevant in critical environments like airplanes or medical devices where buggy software can cause deaths in the worst case. Indeed, there are tools for proof assistants like Coq to convert their code to common functional programming languages like OCaml or Haskell.[21]

[21] For Coq, there is the *Extraction Library*, which generates program code in OCaml, Haskell, or Scheme. Consider the documentation [24] in which you can see an example for code extraction on the basis of a proof of Euclidean division.

5 Case Study on Two Systems of Modal Logic

From the previous chapters we know what Computational Type Theory is, how it can be used as an alternative philosophical approach regarding the nature of mathematical objects and logic, and that there are some pragmatic reasons for working with it. Sadly, most results in mathematics and logic are not formulated in CTT but in the classical logic of axiomatic set theory (ZFC). It would be a very big object against CTT if it set us back to zero because we cannot rely on set-theoretic results. So the big remaining question is, whether it is simply possible to achieve these results in CTT as well. I believe this to be the case for most scenarios, even if this question can hardly be answered uniformly. The following chapter confirms this answer exemplarily. We will use the article *Intuitionistic Modal Logics as Fragments of Classical Bimodal Logics* by Wolter and Zakharyaschev published in January 1998 [1], written in classical set theory, as a case study to demonstrate that classical proofs can be translated in CTT quite smoothly. Additionally, we will implement most results of this chapter in the Coq proof assistant [19] to showcase the benefits of CTT, which I sketched in Chapter 4.

5.1 Syntax and Deduction Systems

To display modality we define formulas with a modal operator. In the following we consider two different modal systems. For the intuitionstic monomodal system we define IM-Formulas as an inductive type ($\mathcal{F}_{\mathsf{IM}}$) as follows:

$$
\begin{aligned}
\mathcal{F}_{\mathsf{IM}} &: \mathbb{T} \\
\mathit{var} &: \mathbb{N} \to \mathcal{F}_{\mathsf{IM}} \\
\supset &: \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{IM}} \\
\vee &: \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{IM}} \\
\wedge &: \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{IM}} \\
\Box &: \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{IM}}
\end{aligned}
$$

In the following, we will utilize the common infix notation for sentential connectives, e.g. we use $\varphi \supset \psi$ instead of $\supset\ \varphi\ \psi$ to describe the the application of the $\supset$-value constructor to the formulas φ and ψ. Note that $\varphi \supset \psi$ is an implication in our object logic and has to be distinguished from $\varphi \to \psi$, an implication of the constructive logic of CTT, which we use as a metalogic to prove properties of the object logic.

Additionally, we omit the name of the *var*-constructor. This allows for a much more concise notation of the type definition as a Backus–Naur form:

$$\varphi, \psi : \mathcal{F}_{\mathsf{IM}} ::= \bot \mid x : \mathbb{N} \mid \varphi \supset \psi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \Box\varphi$$

For the classical bimodal system we define BM-Formulas ($\mathcal{F}_{\mathsf{BM}}$):

$$\varphi, \psi : \mathcal{F}_{\mathsf{BM}} ::= \bot \mid x : \mathbb{N} \mid \varphi \supset \psi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \Box_i\varphi \mid \Box_m\varphi$$

All $\Box$-operators symbolize a separate form of modality. We consider negation of formulas $\neg\varphi$ as an abbreviation for the implication $\varphi \supset \bot$.

With this syntax we now are able to describe two modal logics. The first logic we are interested in is named IntK and consists of the deduction rules for standard intuitionistic logic plus necessitation (**Nec**) plus the **K** rule which distributes the modal operator over the implication. Classically, the context of a deduction is realized as a set of formulas. Since sets are no base type in CTT and not very likeable in terms of computation, we opt for lists of formulas. This way, the context stays finite, a topic which we have to touch on in a later part. We define the deduction predicate $\vdash$: $\mathcal{L}(\mathcal{F}_{\mathsf{IM}}) \to \mathcal{F}_{\mathsf{IM}} \to \mathbb{T}$ for IntK with the following natural deduction system:

$$\frac{\varphi \in \Gamma}{\Gamma \vdash \varphi}\,\mathbf{A} \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \supset \psi}\,\mathbf{I}_{\supset} \qquad \frac{\Gamma \vdash \varphi \supset \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\,\mathbf{E}_{\supset}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\,\mathbf{I}^{l}_{\vee} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\,\mathbf{I}^{r}_{\vee} \qquad \frac{\Gamma \vdash \psi \vee \psi \quad \Gamma, \varphi \vdash \chi \quad \Gamma, \psi \vdash \chi}{\Gamma \vdash \chi}\,\mathbf{E}_{\vee}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\,\mathbf{I}_{\wedge} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\,\mathbf{E}^{l}_{\wedge} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\,\mathbf{E}^{r}_{\wedge}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\,\mathbf{E}_{\bot} \qquad \frac{\vdash \varphi}{\forall \Gamma.\, \Gamma \vdash \Box\varphi}\,\mathbf{Nec} \qquad \frac{}{\Gamma \vdash \Box(\varphi \supset \psi) \supset \Box\varphi \supset \Box\psi}\,\mathbf{K}$$

We also want to consider the classical bimodal logic $\mathsf{S4} \otimes \mathsf{K}$ which consists of the classical deduction rules plus necessitation and **K** for both modal operators, as well as the additional rules **T** and **4** for the $\Box_i$-modality. So the deduction predicate $\vdash_b$: $\mathcal{L}(\mathcal{F}_{\mathsf{BM}}) \to \mathcal{F}_{\mathsf{BM}} \to \mathbb{T}$ is defined by deduction rules analogous to the first nine rules of $\vdash$ and these additional seven rules:

$$\frac{\vdash_b \varphi}{\forall \Gamma.\, \Gamma \vdash_b \Box_m\varphi}\,\mathbf{Nec}_m \qquad \frac{\vdash_b \varphi}{\forall \Gamma.\, \Gamma \vdash_b \Box_i\varphi}\,\mathbf{Nec}_i \qquad \frac{}{\Gamma \vdash_b \Box_i(\varphi \supset \psi) \supset \Box_i\varphi \supset \Box_i\psi}\,\mathbf{K}_i$$

$$\frac{}{\Gamma \vdash_b \Box_i\varphi \supset \varphi}\,\mathbf{T} \qquad \frac{}{\Gamma \vdash_b \Box_i\varphi \supset \Box_i(\Box_i\varphi)}\,\mathbf{4} \qquad \frac{}{\Gamma \vdash_b \Box_m(\varphi \supset \psi) \supset \Box_m\varphi \supset \Box_m\psi}\,\mathbf{K}_m$$

$$\frac{\Gamma, \neg\varphi \vdash_b \bot}{\Gamma \vdash_b \varphi}\,\mathbf{E}_{\bot}$$

5.2 Kripke Frames

Consider
`https://yamusk.github.io/BA_case_study_implementation/kripke_evaluation.html`
for the Coq implementation of this section.

5.2.1 IM-Frames

An intuitionistic modal Kripke frame (IM-Frame) is a 4-tuple $M = \langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$ where $\mathcal{W}$ is the type of all Kripke worlds belonging to the frame. The binary relation $\leq$ between worlds, used to model intuitionistic implication, is assumed as a reflexive and transitive order, i.e. as a preorder. To handle the modality at the semantic level we use the binary relation $\mathcal{R}$, implemented as a predicate $\mathcal{W} \to \mathcal{W} \to \mathbb{P}$. The following restriction applies:

$$\forall w_1 w_2 w_3.\, w_1 \leq w_2 \to w_2 \mathcal{R} w_3 \to w_1 \mathcal{R} w_3 \tag{5.1}$$

The binary valuation predicate $\mathcal{V} : \mathcal{W} \to \mathbb{N} \to \mathbb{P}$ takes a world and a natural number as input and returns a proposition. It replaces the set of algebraic cones used in the models of [1]. Additionally, Kripke-Frames ensure persistence of the valuation:

$$\forall w_1 w_2 : \mathcal{W},\, x : \mathbb{N}.\, w_1 \leq w_2 \to \mathcal{V} w_1 x \to \mathcal{V} w_2 x \tag{5.2}$$

Now an evaluation predicate $\Vdash : \mathcal{W} \to \mathcal{F}_{\mathsf{IM}} \to \mathbb{P}$ can be formulated as follows:

$$\begin{aligned}
w \Vdash \bot &:= \bot \\
w \Vdash x &:= \mathcal{V} w x \\
w \Vdash \varphi \supset \psi &:= \forall w'.\, w \leq w' \to w' \Vdash \varphi \to w' \Vdash \psi \\
w \Vdash \varphi \vee \psi &:= w \Vdash \varphi \vee w \Vdash \psi \\
w \Vdash \varphi \wedge \psi &:= w \Vdash \varphi \wedge w \Vdash \psi \\
w \Vdash \Box \varphi &:= \forall w'.\, w \mathcal{R} w' \to w' \Vdash \varphi
\end{aligned}$$

Given this predicate we define semantic inference/the validity-relation with lists of formulas as premises.

$$\Gamma : \mathcal{L}(\mathcal{F}_{\mathsf{IM}}) \Vdash \varphi := \forall M w.\, (\forall \gamma \in \Gamma.\, w \Vdash \gamma) \to w \Vdash \varphi.$$

Like above, we usually omit noting down the corresponding IM-Frame. If it is not clear from the context which IM-Frame corresponds to the evaluation we place the Frame at the beginning, for example $M, w \Vdash \varphi$.

Now we would like to prove soundness for IntK, i.e. to show that if something is deducible in IntK it is also semantically derivable. To succeed in doing so, we first have

to achieve a proof of monotonicity regarding $\leq$.

Lemma 2 (Monotonicity). *Given an IM-Frame $\langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$, a formula φ and two worlds w, v it can be proven that*

$$w \leq v \to w \Vdash \varphi \to v \Vdash \varphi.$$

Proof. By induction on φ. Since this is the first inductive proof in this thesis, we will explain the line of action in detail. First, we can define an eliminator for $\mathcal{F}_{\mathsf{IM}}$ of the following type

$$\begin{aligned}
\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} :& \forall (p : \mathcal{F}_{\mathsf{IM}} \to \mathbb{T}).\, p\bot \to \\
& (\forall (x : \mathbb{N}).\, p\, x) \to \\
& (\forall \varphi\psi.\, p\, \varphi \to p\, \psi \to p\, (\varphi \supset \psi)) \to \\
& (\forall \varphi\psi.\, p\, \varphi \to p\, \psi \to p\, (\varphi \vee \psi)) \to \\
& (\forall \varphi\psi.\, p\, \varphi \to p\, \psi \to p\, (\varphi \wedge \psi)) \to \\
& (\forall \varphi.\, p\, \varphi \to p(\Box\varphi)) \to \\
& \forall \varphi.\, p\, \varphi
\end{aligned}$$

with these defining equations

$$\begin{aligned}
&\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} p\ p_\bot\ f_x\ f_\supset\ f_\supset\ f_\vee\ f_\wedge\ f_\Box\ \bot && \coloneqq p_\bot \\
&\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} p\ p_\bot\ f_x\ f_\supset\ f_\supset\ f_\vee\ f_\wedge\ f_\Box\ x && \coloneqq f_x x \\
&\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} p\ p_\bot\ f_x\ f_\supset\ f_\supset\ f_\vee\ f_\wedge\ f_\Box\ (\varphi \supset \psi) && \coloneqq f_\supset\, \varphi\, \psi\, (\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} \varphi)(\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} \psi) \\
&\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} p\ p_\bot\ f_x\ f_\supset\ f_\supset\ f_\vee\ f_\wedge\ f_\Box\ (\varphi \vee \psi) && \coloneqq f_\vee\, \varphi\, \psi\, (\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} \varphi)(\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} \psi) \\
&\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} p\ p_\bot\ f_x\ f_\supset\ f_\supset\ f_\vee\ f_\wedge\ f_\Box\ (\varphi \wedge \psi) && \coloneqq f_\wedge\, \varphi\, \psi\, (\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} \varphi)(\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} \psi) \\
&\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} p\ p_\bot\ f_x\ f_\supset\ f_\supset\ f_\vee\ f_\wedge\ f_\Box\ (\Box\varphi) && \coloneqq f_\Box\, \varphi\, (\mathtt{E}_{\mathcal{F}_{\mathsf{IM}}} \varphi).
\end{aligned}$$

As already explained in Section 2.2.2, the eliminator allows us to prove a property for all elements of a type in case we provide a proof for all value constructors of the type. Additionally, the eliminator for formulas not only provides us with such a case distinction but also with induction hypotheses in the cases for implication, disjunction, conjunction, and necessitation. For example, in the case for implication, we construct an object of type $p\,(\varphi \supset \psi)$ and the eliminator provides us with two objects of type $p\,\varphi$ and $p\,\psi$ as induction hypotheses.

With this eliminator, we can proceed proving inductively our lemma $\forall\varphi.\,\forall wv.\,w \leq v \to w \Vdash \varphi \to v \Vdash \varphi$. To apply the eliminator, we choose as our predicate $p \coloneqq \lambda\varphi.\,\forall wv.\,w \leq v \to w \Vdash \varphi \to v \Vdash \varphi$. As you can see in the type of the eliminator, we now have to construct a function for each of the six cases.

($\varphi = \bot$) In the case for $\bot$, we need to show $[p\bot =]\,\forall wv.\,w \leq v \to w \Vdash \bot \to v \Vdash \bot$. Since $w \Vdash \bot = \bot$ by definition, the case follows by ex falso quodlibet.

($\varphi = x$) In the variable-case, we show $\forall x.\,\forall wv.\,w \leq v \to w \Vdash x \to v \Vdash x$. By the definition of $\Vdash$, this equals $\forall wv.\,w \leq v \to \mathcal{V}wx \to \mathcal{V}vx$, which is ensured by the persistence property (5.2).

($\varphi = \varphi \supset \psi$) If the formula is an implication, then it suffices to prove for all formulas φ, ψ that $[p\,(\varphi \supset \psi) =]\forall wv.\,w \leq v \to w \Vdash \varphi \supset \psi \to v \Vdash \varphi \supset \psi$, given the induction hypotheses $[p\,\varphi =]\forall wv.\,w \leq v \to w \Vdash \varphi \to v \Vdash \varphi$ and $[p\,\psi =]$. In this case we don't even need the induction hypotheses. The statement simply follows by the definition of $\Vdash$ and the transitivity of $\leq$.

($\varphi = \varphi \vee \psi$) In case the formula is a disjunction, we have to show that $\forall wv.\,w \leq v \to w \Vdash \varphi \vee \psi \to v \Vdash \varphi \vee \psi$, given the same hypotheses $p\,\varphi$ and $p\,\psi$ as in the implication case. We assume $w \leq v$ and $w \Vdash \varphi \vee \psi$ to show $v \Vdash \varphi \vee \psi$. Since $w \Vdash \varphi \vee \psi = w \Vdash \varphi \vee w \Vdash \psi$, we get two cases. Each case is straightforward after applying the appropriate induction hypothesis.

($\varphi = \varphi \wedge \psi$) For conjunctions, we show that $\forall wv.\,w \leq v \to w \Vdash \varphi \wedge \psi \to v \Vdash \varphi \wedge \psi$, given the same hypotheses as in the previous cases. Analogous to the disjunction case, we assume $w \leq v$ and $w \Vdash \varphi \wedge \psi$ to show $v \Vdash \varphi \wedge \psi$. Since $w \Vdash \varphi \wedge \psi = w \Vdash \varphi \wedge w \Vdash \psi$, we show both conjuncts by the induction hypotheses.

($\varphi = \Box\varphi$) In the modal case, we get $p\,\varphi$ as an induction hypothesis but we do not require it for the construction. We have to show $p(\Box\varphi)$, i.e. $\forall wv.\,w \leq v \to w \Vdash \Box\varphi \to v \Vdash \Box\varphi$. We take $w \leq v$ and $w \Vdash \Box\varphi$ as arguments to prove $v \Vdash \Box\varphi$. By the definition of $\Vdash$, we have $\forall w'.\,w\mathcal{R}w' \to w' \Vdash \varphi$ and prove $\forall v'.\,v\mathcal{R}v' \to v' \Vdash \varphi$, which follows from the restriction 5.1. ■

Now it should be clear how an inductive proof works in CTT. Since its always the same procedure, I will keep the eliminators implicit in all the upcoming inductive proofs and only provide very high-level remarks on the proof ideas. For more detail simply consider the Coq implementation.

At the basis of monotonicity, we can establish a proof of soundness:

Lemma 3 (Soundness of IntK). *Let $\langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$ be an IM-Frame. For all lists of IM-formulas Γ and all formulas φ it holds that*

$$\Gamma \vdash \varphi \to \Gamma \Vdash \varphi.$$

Proof. By induction on $\Gamma \vdash \varphi$ using 5.1, reflexivity and transitivity of $\leq$, and monotonicity (Lemma 2) in the $I_{\supset}$-case. ■

5.2.2 BM-Frames

A classical bimoldal Kripke-Frame (BM-Frame) is a 4-tuple $M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$. R_i is a preorder for which persistence applies.

$$\forall w_1 w_2 : \mathcal{W},\ x : \mathbb{N}.\ w_1 \mathcal{R}_i w_2 \to \mathcal{V} w_1 x \to \mathcal{V} w_2 x \tag{5.3}$$

The binary relation $\mathcal{R}_m$ is unrestricted. The evaluation-predicate $\Vdash_b$ is defined for a BM-Frame below:

$$\begin{aligned}
w \Vdash_b \bot &:= \bot \\
w \Vdash_b x &:= \mathcal{V} w x \\
w \Vdash_b \varphi \supset \psi &:= w \Vdash_b \varphi \to w \Vdash_b \psi \\
w \Vdash_b \varphi \vee \psi &:= w \Vdash_b \varphi \vee w \Vdash_b \psi \\
w \Vdash_b \varphi \wedge \psi &:= w \Vdash_b \varphi \wedge w \Vdash_b \psi \\
w \Vdash_b \Box_i \varphi &:= \forall w'.\ w \mathcal{R}_i w' \to w' \Vdash_b \varphi \\
w \Vdash_b \Box_m \varphi &:= \forall w'.\ w \mathcal{R}_m w' \to w' \Vdash_b \varphi
\end{aligned}$$

With $\Vdash_b$ is now an analogous satisfaction relation $\Gamma \Vdash_b \varphi$ definable.

Obviously, we also want a proof for soundness of the $\mathsf{S4} \otimes \mathsf{K}$-logic corresponding to the BM-Frames. Because the BM-deduction system consists of a classical core, one needs to assume excluded middle to prove the soundness property.

Lemma 4 (Soundness of $\mathsf{S4} \otimes \mathsf{K}$). *Let $\langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ be a BM-Frame. Assuming* LEM*, for all lists of BM-formulas Γ and all formulas φ it holds that*

$$\Gamma \vdash_b \varphi \to \Gamma \Vdash_b \varphi.$$

Proof. By induction on $\Gamma \Vdash_b \varphi$ using reflexivity and transitivity of $\mathcal{R}_i$ and LEM in the case for $E_\perp$. ■

5.3 Translating the Semantics

5.3.1 Translating the Syntax

We can translate IM-formulas into BM-formulas with the following translation function $t : \mathcal{F}_{\mathsf{IM}} \to \mathcal{F}_{\mathsf{BM}}$:

$$\begin{aligned} tx &:= \Box_i x \\ t\perp &:= \Box_i \perp \\ t(\varphi \supset \psi) &:= \Box_i(t\varphi \supset t\psi) \\ t(\varphi \vee \psi) &:= \Box_i(t\varphi \vee t\psi) \\ t(\varphi \wedge \psi) &:= \Box_i(t\varphi \wedge t\psi) \\ t(\Box\varphi) &:= \Box_i\Box_m(t\varphi) \end{aligned}$$

5.3.2 Translating IM-Frames to BM-Frames

The translation from IM-Frames to BM-Frames is quite simple because the relations of BM-Frames are less restricted. So given an IM-Frame, we can create a BM-Frame without any difficulties.

Therefore we consider the transformation σ, which takes an IM-Frame $M = \langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$ and constructs a BM-Frame $\sigma M = \langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$.

Lemma 5 (σ). *If $\langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$ is an IM-Frame then $\sigma\langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$ is a BM-Frame.*

Proof. It needs to be shown that $\leq$ is a preorder and persistence applies. Both is trivial via the definition of IM-Frames. ■

Lemma 6 (σ preserves evaluation). *Suppose $M = \langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$ is an arbitrary IM-Frame. For all formulas φ we have*

$$\forall w.\, M, w \Vdash \varphi \leftrightarrow \sigma M, w \Vdash_b t\varphi.$$

Proof. By induction on φ with w quantified, using monotonicity for $\leq$ (Lemma 2). ■

Note that in lemma 6 neither direction can be proven separately. The induction hypothesis must contain the biconditional.

5.3.3 Translating BM-Frames to IM-Frames

Now we would like to have a transformation from BM-Frames to IM-Frames. It is clear that this direction will be a little more difficult because IM-Frames must satisfy condition 5.1 while the corresponding relation in BM-Frames is unrestricted in general.

For a first step we define the $*$-transformation. If $M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ is a BM-Frame we set $M^* \coloneqq \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m^*, \mathcal{V} \rangle$, where $\mathcal{R}_m^*$ is defined as

$$\forall wv.\, w\mathcal{R}_m^* v \leftrightarrow \exists ab.\, w\mathcal{R}_i a \wedge a\mathcal{R}_m b \wedge b\mathcal{R}_i v. \tag{5.4}$$

Lemma 7 $(*)$. *If $\langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ is a BM-Frame then $\langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m^*, \mathcal{V} \rangle$ is also a BM-Frame.*

Proof. Trivial because the third component of BM-Frames is unrestricted. ■

Lemma 8 (Subset). *For all BM-Frames $\langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ it holds that $\mathcal{R}_m$ is a subset of $\mathcal{R}_m^*$, i.e:*

$$\forall wv.\, w\mathcal{R}_m v \rightarrow w\mathcal{R}_m^* v.$$

Proof. Given $w\mathcal{R}_m v$ we need to show that $\exists ab.\, w\mathcal{R}_i a \wedge a\mathcal{R}_m b \wedge b\mathcal{R}_i v$ by (eq. (5.4)). For $a \coloneqq w$ and $b \coloneqq v$ this claim holds by reflexivity of $\mathcal{R}_i$. ■

We are now able to prove the following lemmas:

Lemma 9 (Mix). *For all BM-Frames $M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ we have*

$$\forall w\varphi.\, M^*, w \Vdash_b (\Box_i \Box_m \varphi \leftrightarrow \Box_m \varphi) \wedge (\Box_m \Box_i \varphi \leftrightarrow \Box_m \varphi)$$

respectively

$$\forall wv.\, (w\mathcal{R}_m^* v \leftrightarrow \exists a.\, w\mathcal{R}_i a \wedge a\mathcal{R}_m^* v) \wedge (w\mathcal{R}_m^* v \leftrightarrow \exists a.\, w\mathcal{R}_m^* a \wedge a\mathcal{R}_i v).$$

Proof. Straightforward. ■

Lemma 10 ($*$ preserves evaluation). *Suppose $M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ is an arbitrary BM-Frame. For all formulas φ it holds that*

$$\forall w.\, M, w \Vdash_b t\varphi \leftrightarrow M^*, w \Vdash_b t\varphi.$$

Proof. By induction on φ with w quantified, exploiting that $t\varphi$ begins with $\Box_i$ by the definition of t and using Lemma 8 in the modal case. ■

Lemma 11. *Suppose $M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ is an arbitrary BM-Frame. For all worlds w, v, u holds that*

$$w\mathcal{R}_i v \to v\mathcal{R}_m^* u \to w\mathcal{R}_m^* u.$$

Proof. Straightforward. ■

Consider now the transformation ρ which takes a BM-Frame $M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ and transforms it into the IM-Frame $\rho M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m^*, \mathcal{V} \rangle$.

Lemma 12. *If $\langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ is a BM-Frame then $\rho\langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ is an IM-Frame.*

Proof. If suffices to prove that $\forall w_1 w_2 w_3.\, w_1 \mathcal{R}_i w_2 \to w_2 \mathcal{R}_m^* w_3 \to w_1 \mathcal{R}_m^* w_3$ which is exactly Lemma 11. ■

Lemma 13 (ρ preserves evaluation). *Suppose $M = \langle \mathcal{W}, \mathcal{R}_i, \mathcal{R}_m, \mathcal{V} \rangle$ is an arbitrary BM-Frame. For all formulas φ it holds that*

$$\forall w.\, M, w \Vdash_b t\varphi \leftrightarrow \rho M, w \Vdash \varphi.$$

Proof. By induction on φ with w quantified, relying on monontonicity for $\mathcal{R}_i$ (Lemma 2). ■

5.3.4 Main Semantic Results

To connect the results we can show that even the concatenation of σ and ρ preserves evaluation:

Theorem 1 ($\rho \circ \sigma$ preserves evaluation). *Suppose $M = \langle \mathcal{W}, \leq, \mathcal{R}, \mathcal{V} \rangle$ is an arbitrary IM-Frame. For all formulas φ it holds that*

$$\forall w.\, M, w \Vdash \varphi \leftrightarrow \rho(\sigma M), w \Vdash \varphi.$$

Proof. Both directions are straightforward by combining Lemma 6 and Lemma 13. ■

Because both transformations preserve evaluation, we are now able to prove the following more general equivalence, hiding both transformations in the proof.

Theorem 2. *For all formulas φ it holds that*

$$(\forall Mw.\, M, w \Vdash \varphi) \leftrightarrow \forall Mw.\, M, w \Vdash_b t\varphi.$$

Proof. The $\leftarrow$ direction follows by Lemma 6. The $\rightarrow$ direction follows by Lemma 13. ■

The same result can be shown in a context:

Theorem 3 (semantic equivalence). *For all formulas φ and all lists of IM-formulas Γ it holds that*

$$\Gamma \Vdash \varphi \leftrightarrow t\Gamma \Vdash_b t\varphi$$

Proof. Analogous proof to Theorem 2, additionally using that $s \in \Gamma \leftrightarrow ts \in t\Gamma$. ■

Theorem 4 (decidability preservation). *For all list of IM-Formulas Γ it holds that*

$$(\forall \varphi.\, \mathcal{D}(t\Gamma \Vdash_b \varphi)) \rightarrow \forall \varphi.\, \mathcal{D}(\Gamma \Vdash \varphi).$$

Proof. We assume $\varphi : \mathcal{F}_{\mathsf{IM}}$ and discriminate on $\mathcal{D}(t\Gamma \Vdash_b t\varphi)$. In the case for $t\Gamma \Vdash_b t\varphi$ we show $\Gamma \Vdash \varphi$ by Lemma 6. In the case for $t\Gamma \nVdash_b t\varphi$ we show $\Gamma \nVdash \varphi$ by Lemma 13. ■

5.4 Transposing the Deduction Systems

In this section the goal is to demonstrate that a translation $t\varphi$ is deducible in $\mathsf{S4} \otimes \mathsf{K}$ if φ is deducible in IntK. Consider
`https://yamusk.github.io/BA_case_study_implementation/deduction.html`
for the Coq implementation of this section.

5.4.1 Weakening

The first step is to prove a property called weakening. This property allows us to enrich the context of a deduction. Intuitively, it should be clear that logic deduction should be monotone, i.e. that it should not be possible that a sound argument becomes unsound if further premises get added. We will show this property holds for the object language $\mathsf{S4} \otimes \mathsf{K}$.

Lemma 14 (Weakening for $\mathsf{S4} \otimes \mathsf{K}$). *For all lists of BM-formulas* Γ_1 *and all BM formulas* φ*, it holds that*

$$\Gamma_1 \vdash_b \varphi \to \forall\, \Gamma_2.\, \Gamma_1 \subseteq \Gamma_2 \to \Gamma_2 \vdash_b \varphi.$$

Proof. By induction on $\Gamma_1 \vdash_b \varphi$ with Γ_2 quantified. All cases are straightforward using the case specific deduction rule. ■

Now we can prove that if a formula is provable in the empty context, then the formula is provable in every context:

Lemma 15. *Let* φ *be a BM-formula. Then*

$$\vdash_b \varphi \to \forall\, \Gamma.\, \Gamma \vdash_b \varphi.$$

Proof. Straightforward by the application of Lemma 14. ■

Necessitation often is described as a modal inference rule in the empty context. Such a formulation wouldn't allow us to prove weakening. But now we can demonstrate that our necessitation rules for $\mathsf{S4} \otimes \mathsf{K}$ imply the standard formulation:

Lemma 16 (Necessitation).

$$\frac{\vdash_b \varphi}{\vdash_b \Box_i \varphi} \qquad\qquad \frac{\vdash_b \varphi}{\vdash_b \Box_m \varphi}$$

Proof. By applying $\mathbf{Nec}_i$/ $\mathbf{Nec}_m$ succeeded by Lemma 15. ■

5.4.2 Shifting

Lemma 17 (Reverse $\mathbf{I}_\supset$).

$$\frac{\Gamma \vdash_b \varphi \supset \psi}{\Gamma, \varphi \vdash_b \psi}\ \boldsymbol{I}_\supset^{\leftarrow}$$

Proof. By $\mathbf{E}_\supset$ and weakening. ■

With $\mathbf{I}_\supset$and $\mathbf{I}_\supset^{\leftarrow}$ we now are able to shift the context of a deduction into the formula as an antecedence and back in the context again. Therefore, we define a recursive

shifting function $\gg$ returning the shifted formula:

$$\varnothing \gg \varphi := \varphi$$
$$\gamma :: \Gamma \gg \varphi := \Gamma \gg (\gamma \supset \varphi)$$

According to this definition: $[\gamma_1, \gamma_2, ..., \gamma_n] \gg \varphi = \gamma_n \supset ... \supset \gamma_2 \supset \gamma_1 \supset \varphi$.

Lemma 18 (Shifting). *For all lists of formulas* Γ *and all formulas* φ *we have*

$$\Gamma \vdash_b \varphi \leftrightarrow \vdash_b (\Gamma \gg \varphi).$$

Proof. By induction on Γ with φ quantified. ■

Lemma 19 (Modal Shifting[22]). *Let* Γ *be a list of translated formulas* $t\gamma_1, ..., t\gamma_n$. *For all BM-formulas* φ, *it holds that*

$$\Gamma \vdash_b \varphi \rightarrow \Gamma \vdash_b \Box_i \varphi.$$

Proof. We take $\Gamma \vdash_b \varphi$ and shift the context with $\mathbf{I}_{\supset}$completely in the formula, such that $\vdash_b t\gamma_n \supset ... \supset t\gamma_1 \supset \varphi$. After applying necessitation, we get $\vdash_b \Box_i(t\gamma_n \supset ... \supset t\gamma_1 \supset \varphi)$. Now $\mathbf{K}_i$ allows us to distribute the modal operator into the implication: $\vdash_b \Box_i(t\gamma_n) \supset ... \supset \Box_i(t\gamma_1) \supset \Box_i\varphi$. Now we exploit again that all translated formulas begin with $\Box_i$_ and infer $\vdash_b t\gamma_n \supset ... \supset t\gamma_1 \supset \Box_i\varphi$ by $\mathbf{T}$. Finally, we complete the proof by reversing the context shifting with $\mathbf{I}^{\leftarrow}_{\supset}$. ■

5.4.3 Transposition

Theorem 5 (Transpose). *Let* Γ *be a list of IM-formulas. For all IM-formulas* φ *we have*

$$\Gamma \vdash \varphi \rightarrow t\Gamma \vdash_b t\varphi.$$

Proof. By induction on $\Gamma \vdash \varphi$. All cases are relatively straightforward with the use of the modal shifting lemma, except the case for $\mathbf{K}$.

In this case, it needs to be shown that $t\Gamma \vdash_b t(\Box(\varphi \supset \psi) \supset \Box\varphi \supset \Box\psi)$.[23] Since we have no useful information about the context, we just show the statement in the

[22] This proof is not implemented in the proof assistant.

[23] It might be hard to follow the outline of this proof on paper. Please consider checking either the Coq code or the derivation trees of the proof in the Appendix for more detail.

empty context (Lemma 15) and eliminate the modality in front with $\mathbf{Nec}_i$, which leads to $\vdash_b t(\Box(\varphi \supset \psi)) \supset t(\Box\varphi \supset \Box\psi)$. We then shift the antecedence in the context and apply the modal shifting lemma to get $[t(\Box(\varphi \supset \psi))] \vdash_b t(\Box\varphi) \supset t(\Box\psi)$. This gets implied by $[t(\Box(\varphi \supset \psi))] \vdash_b \Box_m(t\varphi) \supset \Box_m(t\psi)$ by shifting back and forth and the **T**-rule. Applying **T** again results in $\vdash_b \Box_m\Box_i(t\varphi \supset t\psi) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)$. Since $\vdash_b \Box_m\Box_i(t\varphi \supset t\psi) \supset \Box_m(\Box_i(t\varphi) \supset \Box_i(t\psi))$, we can distribute $\Box_i$ in the implication. So we need to show $\vdash_b (\Box_m(\Box_i(t\varphi) \supset \Box_i(t\psi))) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)$. Now $\mathbf{K}_m$ allows us to also distribute $\Box_m$ in the implication, producing the goal $\vdash_b (\Box_m\Box_i(t\varphi) \supset \Box_m\Box_i(t\psi)) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)$. We now shift the antecedence in the context, such that $[\Box_m\Box_i(t\varphi) \supset \Box_m\Box_i(t\psi), \Box_m(t\varphi)] \vdash_b \Box_m(t\psi)$. Since the implications $\Box_m\Box_i(t\psi) \supset \Box_m(t\psi)$ and $\Box_m(t\varphi) \supset \Box_m\Box_i(t\varphi)$ are both provable in the empty context, the proof is closed.

■

5.5 Completeness of IntK

The goal of the upcoming section is to show completeness of IntK, i.e. to prove $\Gamma \Vdash \varphi \to \Gamma \vdash \varphi$. Note that we already proved the opposite direction called soundness in Lemma 3. So if we achieve our goal, we can freely switch back and forth between the deduction and semantics level of the intuitionistic system.

The non-constructive proof for IntK that will described in the following is leaned on the completeness proof for IEL in the appendix of [25]. This proof could probably also be done in a constructive way, i.e. without assuming LEM. The constructive version will not be covered here since it is way more sophisticated and thus beyond the scope of this thesis. You can look at Section 6 of [26] to get the idea.

Consider
`https://yamusk.github.io/BA_case_study_implementation/IntK_completeness.html`
for the Coq implementation of this section.

5.5.1 The Canonical Frame

The first step of proving completeness is the definition of the canonical IM-Frame. Therefore, we need the following wordings and relations:

We speak of a context Γ as a theory if it is *deductively closed*, i.e. if for all formulas φ

$$\Gamma \vdash \varphi \to \varphi \in \Gamma.$$

Since a finite context can never be deductively closed ($\Gamma \vdash \varphi \supset \varphi, \Gamma \vdash \varphi \supset \varphi \supset \varphi \dots$), we have to switch our context implementation from (finite) lists to predicates $\mathcal{F}_{\mathsf{IM}} \to \mathbb{P}$. To still make use of our deduction predicate $\vdash$: $\mathcal{L}(\mathcal{F}_{\mathsf{IM}}) \to \mathcal{F}_{\mathsf{IM}} \to \mathbb{T}$ which operates with lists as input, we use the following underlying definition translating the two context implementations:

Definition 1. *We define for a predicate* $\Gamma : \mathcal{F}_{\mathsf{IM}} \to \mathbb{P}$ *and a formula* $\varphi : \mathcal{F}_{\mathsf{IM}}$ *that*

$$\Gamma \vdash \varphi \coloneqq \exists \mathcal{L}(\mathcal{F}_{\mathsf{IM}}).\, \mathcal{L}(\mathcal{F}_{\mathsf{IM}}) \subseteq \Gamma \wedge \mathcal{L}(\mathcal{F}_{\mathsf{IM}}) \vdash \varphi,$$

where the $\subseteq$*-relation between lists and predicates is defined as*

$$\mathcal{L}(\mathcal{F}_{\mathsf{IM}}) \subseteq \Gamma \coloneqq \forall \varphi.\, \varphi \in \mathcal{L}(\mathcal{F}_{\mathsf{IM}}) \to \Gamma\varphi.$$

To conventionalize notation, we still use the element symbol for predicate applications, i.e. $\varphi \in \Gamma \coloneqq \Gamma\varphi$. It can be shown quite easily that the deduction rules for IntK are derivable for theories.[24] This allows us to treat theories with an infinite context just like we treated finite contexts and the corresponding deductions before.

A theory is called *prime*, if

$$\forall \varphi\psi.\, \varphi \vee \psi \in \Gamma \to \varphi \in \Gamma \vee \psi \in \Gamma.$$

In addition, we consider the following relation $\mathcal{M}$ on theories Γ_1, Γ_2:

$$\Gamma_1 \,\mathcal{M}\, \Gamma_2 \coloneqq \forall \varphi.\, \Box\varphi \in \Gamma_1 \to \varphi \in \Gamma_2$$

Finally, we need a valuation predicate $\mathcal{V}_T$:

$$\mathcal{V}_T\, \Gamma\, (x : \mathbb{N}) \coloneqq x \in \Gamma$$

Lemma 20 (Canonical Frame). *Let* $\mathcal{T}$ *by the type of all consistent prime theories. Then* $M_C \coloneqq \langle \mathcal{T}, \subseteq, \mathcal{M}, \mathcal{V}_T \rangle$ *is an IM-Frame.*

[24] I won't cover the mostly trivial proofs for the validity of the deduction rules with predicates on paper to avoid unnecessarily stretching out this thesis. Please consider the implementation for details.

Proof. The definition of IM-Frames (Section 5.2.1) demands for three points. It needs to be shown

1. that $\subseteq$ is a preorder.

2. that $\mathcal{V}_T$ is persistent (5.2).

3. that $\forall \Gamma_1, \Gamma_2, \Gamma_3.\ \Gamma_1 \subseteq \Gamma_2 \to \Gamma_2\, \mathcal{M}\, \Gamma_3 \to \Gamma_1\, \mathcal{M}\, \Gamma_3$ (5.1).

1. Preorders are reflexive and transitive relations. That the non-strict subset is such a relation is obvious.

2. To show persistence, we need to show here

$$\forall\, \Gamma_1 \Gamma_2 : \mathcal{T},\ x : \mathbb{N}.\ \Gamma_1 \subseteq \Gamma_2 \to \mathcal{V}_T\, \Gamma_1\, x \to \mathcal{V}_T\, \Gamma_2\, x,$$

which reduces to

$$\Gamma_1 \subseteq \Gamma_2 \to x \in \Gamma_1 \to x \in \Gamma_2,$$

which is trivially the case.

3. It remains to show that

$$\forall\, \Gamma_1, \Gamma_2, \Gamma_3.\ \Gamma_1 \subseteq \Gamma_2 \to \Gamma_2\, \mathcal{M}\, \Gamma_3 \to \Gamma_1\, \mathcal{M}\, \Gamma_3.$$

By the definition of $\mathcal{M}$, it suffices to show $\varphi \in \Gamma_3$ under the assumptions $\Gamma_1 \subseteq \Gamma_2$, $\Box\varphi \in \Gamma_1$, and $\Box\varphi \in \Gamma_2 \to \varphi \in \Gamma_3$, which is again trivial. ■

5.5.2 Assumptions

We will allow ourselves to use the following two assumptions without providing a proof for them. These results are well known in modal logic and the proofs have no aspects making them specific for IntK. For reference you can see the analogous proofs for IEL in [26].

Now let in the following $\Gamma_\Box$ be the context that contains a formula φ if and only if Γ contains $\Box\varphi$.

Assumption 21. *For all predicates* $\Gamma : \mathcal{F}_{\mathsf{IM}} \to \mathbb{P}$ *and formulas* φ *it holds that*

$$\Gamma_\Box \vdash \varphi \to \Gamma \vdash \Box\varphi.$$

Assumption 22 (Lindenbaum Lemma). *Let* Γ *be an arbitrary predicate* $\mathcal{F}_{\mathsf{IM}} \to \mathbb{P}$ *and* φ *a* $\mathcal{F}_{\mathsf{IM}}$*-formula. If* $\Gamma \nvdash \varphi$ *and* $\Gamma \nvdash \bot$*, then there exists a consistent prime theory* $\triangle$ *with the properties*

$$\Gamma \subseteq \triangle \quad \textit{and} \quad \varphi \notin \triangle.$$

With those two assumptions in our hands we can prove the Truth Lemma which is key for showing completeness.

5.5.3 Truth Lemma and Completeness

Lemma 23 (Truth Lemma). *Let* Γ *be a consistent prime theory and* φ *an arbitrary formula. Assuming* LEM *it holds that the worlds* Γ *of* M_C *satisfy*

$$\Gamma \Vdash \varphi \leftrightarrow \varphi \in \Gamma.$$

Proof. By induction on φ with Γ quantified. The induction results in six cases. The cases for $\bot$, variables, and conjunction are straightforward. We concern ourselves with the cases for implication, disjunction, and necessity.

($\varphi = \varphi \supset \psi$). We need to show $M_C, \Gamma \Vdash \varphi \supset \psi \leftrightarrow (\varphi \supset \psi) \in \Gamma$ using the two induction hypotheses $\forall \Gamma.\, M_C, \Gamma \Vdash \varphi \leftrightarrow \varphi \in \Gamma$ and $\forall \Gamma.\, M_C, \Gamma \Vdash \psi \leftrightarrow \psi \in \Gamma$. The right-to-left direction only requires simple classical reasoning. We show the left-to-right direction. By assuming LEM, instead of proving $M_C, \Gamma \Vdash \varphi \supset \psi \to (\varphi \supset \psi) \in \Gamma$, we can prove the contraposition $(\varphi \supset \psi) \notin \Gamma \to M_C, \Gamma \nVdash \varphi \supset \psi$. We get $(\varphi \supset \psi) \notin \Gamma$ and $M_C, \Gamma \Vdash \varphi \supset \psi$ to show $\bot$. We use LEM to do a case analysis on $\Gamma, \varphi \vdash \psi$. In case $\Gamma, \varphi \vdash \psi$ we can deduce $\Gamma \vdash \varphi \supset \psi$ by $\mathbf{I}_{\supset}$. This is contradictory because we know that Γ is deductively closed. So we proceed by assuming $\Gamma, \varphi \nvdash \psi$. The Lindenbaum Lemma 22 provides us with a theory $\triangle$, for which applies $\Gamma, \varphi \subseteq \triangle$ and $\psi \notin \triangle$. Now we use the induction hypothesis to get $\triangle, M_C \Vdash \varphi$ and then infer $\triangle, M_C \Vdash \psi$ by $M_C, \Gamma \Vdash \varphi \supset \psi$. The proof is concluded with the application of the induction hypothesis to get $\psi \in \triangle$ which is a contradiction.

($\varphi = \varphi \vee \psi$). We need to show $M_C, \Gamma \Vdash \varphi \vee \psi \leftrightarrow (\varphi \vee \psi) \in \Gamma$ using the same induction hypotheses as in the case for implication. The left-to-right direction is done with straightforward reasoning using that Γ is closed. The key trick in the right-to-left proof is transforming the assumption to $\varphi \in \Gamma \vee \psi \in \Gamma$ since Γ is a prime theory. Now

the proof goal follows by discrimination and than applying induction hypotheses.

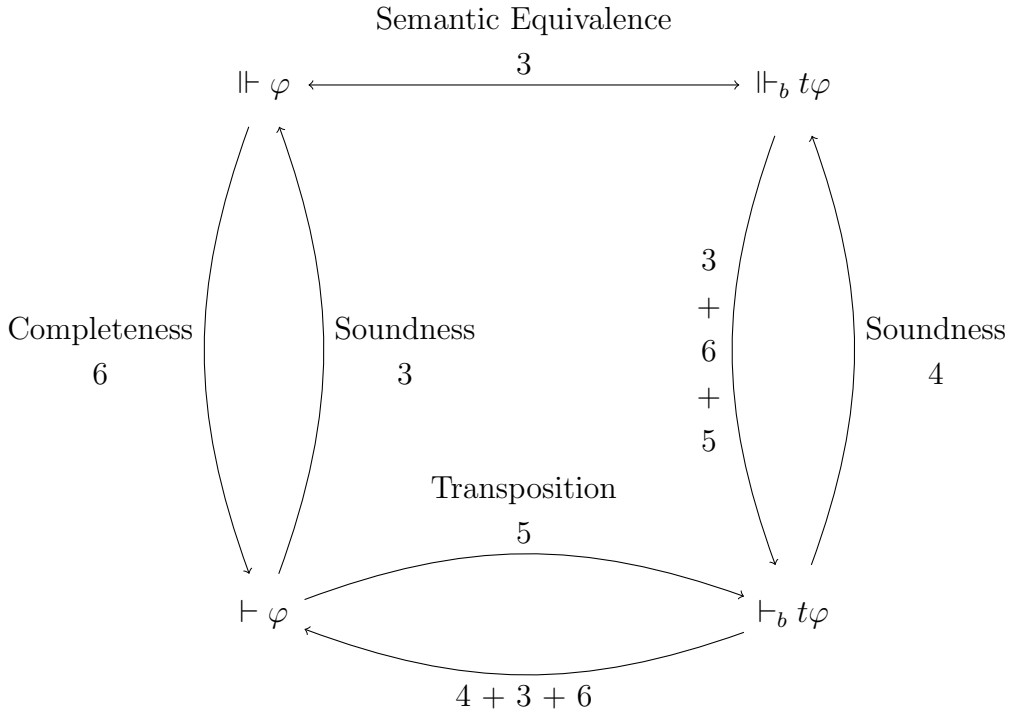
($\varphi = \Box\varphi$). We need to show $M_C, \Gamma \Vdash \Box\varphi \leftrightarrow \Box\varphi \in \Gamma$ using the induction hypothesis $\forall\Gamma.\, M_C, \Gamma \Vdash \varphi$. The right-to-left direction is straightforward. For the left-to-right direction we show the contraposition (using LEM another time) $\Box\varphi \notin \Gamma \to M_C, \Gamma \nVdash \Box\varphi$. More precisely, we infer $\Box\varphi \in \Gamma$ under the assumptions $\Box\varphi \notin \Gamma$ and $\Gamma, M_C \Vdash \Box\varphi$. We know $\Gamma \nvdash \Box\varphi$ (because elsewise $\Box\varphi \in \Gamma$) so 21 gives us $\Gamma_\Box \nvdash \varphi$ by contraposition. Now according to the Lindenbaum Lemma, there is a consistent prime theory $\triangle$ with the properties $\Gamma_\Box \subseteq \triangle$ and $\varphi \notin \triangle$. Because $\Gamma \,\mathcal{M}\, \triangle$, we can finally infer $\triangle, M_C \Vdash \varphi$ which is a contradiction through applying the induction hypothesis. ■

Theorem 6 (Completeness for IntK). *Let Γ be a list of IM-formulas and φ a formula. Assuming* LEM *it holds that*

$$\Gamma \Vdash \varphi \to \Gamma \vdash \varphi.$$

Proof. By contradiction. We assume $\Gamma \Vdash \varphi$ and $\Gamma \nvdash \varphi$ to infer falsity. The Lindenbaum Lemma provides us with a consistent prime theory $\triangle$ with $\Gamma \subseteq \triangle$ and $\varphi \notin \triangle$. It suffices to show $\forall\gamma \in \Gamma.\, \triangle, M_C \Vdash \gamma$, since then $\Gamma \Vdash \varphi$ implies $\triangle, M_C \Vdash \varphi$, which leads to the contradiction $\varphi \in \triangle$ using the Truth Lemma. Now every formula γ from Γ is also an element of $\triangle$, since $\Gamma \subseteq \triangle$. Thus $\triangle, M_C \Vdash \gamma$ via the Truth Lemma. ■

5.6 Main Results



The final result summarizing our endeavors of this chapter is, that the two modal logics IntK and $\mathsf{S4} \otimes \mathsf{K}$ can be translated into another by the translation function t, both on the semantic level and on the deduction level.

6 Conclusion

To come to an end, let me recapitulate the efforts of this thesis. The thesis provided a brief portrayal of Computational Type Theory, covering the technical workings of it as well as the pecularities associated with its philosophical foundations. Additionally, two pragmatic benefits were sketched. They indicate the practical interest of CTT. Lastly, a case study demonstrated exemplarily that CTT is fit for work by embedding classical set-theoretic results of modal logic in CTT. Thereby, a few common tricks for implementing non-computational structures of set theory, e.g. infinite sets (representing contexts of derivations), were shown. So even if many results of classical set theory are not yet available in a computational framework, it is no deal-breaker since most proofs are expected to come along with the classical set-theoretic counterpart quite easily. As a side effect, the case study underpinned the benefit of working with a proof assistant which was described before, since nearly all results were mechanized in the Coq proof assistant.

The obviously restricted scope of a bachelor's thesis did cause the portrayal of both the technical workings and the philosophical implications to have more of an introductory rather than a concluding character. This is especially reflected in the philosophical considerations regarding the nature of mental constructions. Here a lot of questions remained open. To take things further and make a strong argument in favor of CTT, one would need to present a precise notion of mental constructions. The notion should allow for intersubjective reasoning and make it clear whether constructed entities are still abstract or spatially locatable after all. Since the development of Computational Type Theory did put new wind in the sails of intuitionism, it seems only consequent to reopen the philosophical debate whether intuitionism (of CTT) or object realism/platonism often linked with set theory is superior.

The case study on the translation of intuitionistic monomodal logic to classical bimodal logic also contains some points where future considerations could tie up to. On the one hand, the nonconstructive completeness proof for IntK fits the objective of the chapter since it exemplifies how classical proofs can be done trough the assumption of

LEM. On the other hand a constructive version would suit the computational framework better. Furthermore, it would be interesting to research whether a generalization of the translation of IntK to S4 $\otimes$ K to various intuitionistic respectively classical modal logics is possible.

All in all, this bachelor's thesis achieved a concise portrayal of key aspects of Computational Type Theory combined with a practical demonstration of its arguably most compelling feature, namely the ease of mechanization. So the thesis provides a basis for philosophers who want to become acquainted with Computational Type Theory and aspire to engage in the philosophical debate around it.

Appendix

Figure 6.1: Derivation tree for Theorem 5

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_b \Box_m(\Box_i(t\psi) \supset (t\psi)) \supset \Box_m\Box_i(t\psi) \supset \Box_m(t\psi)}\,\mathbf{K}_m \quad \dfrac{}{\vdash_b \Box_m(\Box_i\psi \supset \psi)}\,\mathbf{Nec}_m + \mathbf{T}}{\vdash_b \Box_m\Box_i(t\psi) \supset \Box_m(t\psi)}\,\mathbf{E}_\supset \quad \dfrac{\dfrac{}{\vdash_b \Box_m(t\varphi \supset \Box_i(t\varphi)) \supset \Box_m(t\varphi) \supset \Box_m\Box_i(t\varphi)}\,\mathbf{K}_m \quad \dfrac{}{\vdash_b \Box_m(t\varphi \supset \Box_i(t\varphi))}\,\mathbf{Nec}_m + \mathbf{4}}{\vdash_b \Box_m(t\varphi) \supset \Box_m\Box_i(t\varphi)}\,\mathbf{E}_\supset}{[\Box_m\Box_i(t\varphi) \supset \Box_m\Box_i(t\psi), \Box_m(t\varphi)] \vdash_b \Box_m(t\psi)}\,\mathbf{E}_\supset}{\vdash_b (\Box_m\Box_i(t\varphi) \supset \Box_m\Box_i(t\psi)) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)}\,\mathbf{I}_\supset}{\vdash_b (\Box_m(\Box_i(t\varphi) \supset \Box_i(t\psi))) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)}\,\approx \textit{Figure } 6.3}{\vdash_b \Box_m\Box_i(t\varphi \supset t\psi) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)}\,\textit{Figure } 6.3}{[t(\Box(\varphi \supset \psi))] \vdash_b \Box_m(t\varphi) \supset \Box_m(t\psi)}\,\mathbf{I}^{\leftarrow}_{\supset} + 19}{[t(\Box(\varphi \supset \psi))] \vdash_b t(\Box\varphi) \supset t(\Box\psi)}\,\textit{Figure } 6.2}{\vdash_b t(\Box(\varphi \supset \psi)) \supset t(\Box\varphi \supset \Box\psi)}\,\mathbf{I}_\supset + 19}{t\Gamma \vdash_b t(\Box(\varphi \supset \psi) \supset \Box\varphi \supset \Box\psi)}\,\mathbf{Nec}_i + 15$$

Figure 6.2

$$\dfrac{\dfrac{[t(\Box(\varphi \supset \psi))] \vdash_b \Box_m(t\varphi) \supset \Box_m(t\psi) \quad \dfrac{}{t(\Box\varphi) \supset \Box_m(t\varphi)}\,\mathbf{T}}{[t(\Box(\varphi \supset \psi)), t(\Box\varphi)] \vdash_b \Box_m(t\psi)}\,\mathbf{E}_\supset + \mathbf{I}^{\leftarrow}_{\supset}}{[t(\Box(\varphi \supset \psi)), t(\Box\varphi)] \vdash_b t(\Box\psi)}\,19}{[t(\Box(\varphi \supset \psi))] \vdash_b t(\Box\varphi) \supset t(\Box\psi)}\,\mathbf{I}_\supset$$

Figure 6.3

$$\dfrac{\dfrac{\dfrac{}{\vdash_b \Box_m(\Box_i(t\varphi \supset t\psi) \supset \Box_i(t\varphi) \supset \Box_i(t\psi)) \supset \Box_m\Box_i(t\varphi \supset t\psi) \supset \Box_m(\Box_i(t\varphi) \supset \Box_i(t\psi))}\,\mathbf{K}_m}{\vdash_b \Box_m\Box_i(t\varphi \supset t\psi) \supset \Box_m(\Box_i(t\varphi) \supset \Box_i(t\psi))} \quad \vdash_b (\Box_m(\Box_i(t\varphi) \supset \Box_i(t\psi))) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)}{\vdash_b \Box_m\Box_i(t\varphi \supset t\psi) \supset \Box_m(t\varphi) \supset \Box_m(t\psi)}\,\mathbf{I}_\supset + \mathbf{E}_\supset + \mathbf{I}^{\leftarrow}_{\supset}$$

Bibliography

[1] Frank Wolter and Michael Zakharyaschev: *Intuitionistic modal logics as fragments of classical bimodal logics.* Logic at Work, January 1998.

[2] Per Martin-Löf: *"An intuitionistic theory of types", in Sambin and Smith 1998: 127-172.* Written in 1972 without publishment.

[3] Gottlob Frege: *Grundlagen der Arithmetik: Studienausgabe MIT Dem Text der Centenarausgabe.* Breslau: Wilhelm Koebner Verlag, 1884.

[4] Gottlob Frege and Betrand Russell: *Briefwechsel mit bertrand russell*, 1902. `https://www.hs-augsburg.de/~harsch/germanica/Chronologie/19Jh/Frege/fre_brif.html`.

[5] Alfred N. Whitehead and Bertrand Russell: *Principia Mathematica.* Cambridge University Press, 1910.

[6] Neil Tennant: *Logicism and Neologicism.* In Zalta, Edward N. (editor): *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Winter 2017 edition, 2017.

[7] Rosalie Iemhoff: *Intuitionism in the Philosophy of Mathematics.* In Zalta, Edward N. (editor): *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Fall 2020 edition, 2020.

[8] Maximilian Doré: *Constructivity in homotopy type theory.* Master's thesis, Ludwig Maximilian University of Munich, 2019.

[9] Ernst Zermelo: *Neuer Beweis für die Möglichkeit einer Wohlordnung.* Mathematische Annalen, 65:107–128, 1908, ISSN 0025-5831; 1432-1807/E. `https://eudml.org/doc/158340,https://gdz.sub.uni-goettingen.de/id/PPN235181684_0065`.

[10] Ernst Zermelo: *Untersuchungen über die grundlagen der mengenlehre. i.* Mathematische Annalen, 65:261–281, 1908. http://eudml.org/doc/158344.

[11] Joan Bagaria: *Set Theory.* In Zalta, Edward N. (editor): *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Winter 2021 edition, 2021.

[12] Errett Bishop: *Foundations of Constructive Analysis.* New York, NY, USA: Mcgraw-Hill, 1967.

[13] Gert Smolka: *Modeling and proving in computational type theory using the coq proof assistant*, 2021. https://www.ps.uni-saarland.de/~smolka/drafts/icl2021.pdf.

[14] Per Martin-Löff: *Intuitionistic type theory: Notes by giovanni sambin of a series of lectures given in padua, june 1980*, 1985.

[15] Dominik Kirst: *Foundations of mathematics: A discussion of sets and types*, 2018. Bachelor's thesis.

[16] Alonzo Church: *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies).* Princeton University Press, USA, 1985, ISBN 0691083940.

[17] Øystein Linnebo: *Platonism in the Philosophy of Mathematics.* In Zalta, Edward N. (editor): *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Spring 2018 edition, 2018. https://plato.stanford.edu/archives/spr2018/entries/platonism-mathematics/.

[18] Edited by D. van Dalen: *Brouwer's cambridge lectures on intuitionism*, 1981. http://www.cs.cornell.edu/courses/cs4860/2019fa/resources/Brouwers_Cambridge_Lectures.pdf.

[19] The Coq development team: *The coq proof assistant*, 1989. https://coq.inria.fr/.

[20] *Agda*, 1973. https://wiki.portal.chalmers.se/agda/pmwiki.php.

[21] Leonardo de Moura and Team: *Lean theorem prover*, 2013. https://leanprover.github.io/.

[22] *Metamath proof explorer.* http://us.metamath.org/mpeuni/mmset.html.

[23] *Mizar project*, 1973. http://www.mizar.org/.

[24] Jean-Christophe Filliâtre and Pierre Letouzey: *Program extraction*. https://coq.inria.fr/refman/addendum/extraction.html#.

[25] Sergej Artemov and Tudor Protopopescu: *Intuitionistic epistemic logic*. The Review of Symbolic Logic, 9:266–298, January 2016. https://doi.org/10.48550/arXiv.1406.1582.

[26] Christian Hagemeier and Dominik Kirst: *Constructive and mechanised metatheory of intuitionistic epistemic logic*. 2022. https://www.ps.uni-saarland.de/Publications/documents/HagemeierKirst_2022_Constructive.pdf.

[27] Yannick Forster, Dominik Kirst, and Dominik Wehr: *Completeness theorems for first-order logic analysed in constructive type theory*. LCFS, 2020. https://www.ps.uni-saarland.de/extras/fol-completeness/.