Saarland University
Faculty of Mathematics and Computer Science

Bachelor's Thesis

The Arithmetical Hierarchy, Oracle Computability, and Post's Theorem in Synthetic Computability

Author
Niklas Mück

Supervisor
Prof. Dr. Gert Smolka

Advisors
Dr. Yannick Forster
Dominik Kirst

Reviewers
Prof. Dr. Gert Smolka
Dr. Yannick Forster

Submitted: 18th June 2022

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 18th June, 2022

Abstract

The subject of this thesis is to formalize the arithmetical hierarchy in constructive type theory, set up synthetic oracle computability including a synthetic definition of the Turing jump, and connect both by proving Post's theorem in synthetic computability. We aim at establishing most proofs constructively and mechanize all results in the proof assistant Coq.

Synthetic computability abstracts from concrete models of computation by axiomatically considering all (partial) functions $\mathbb{N} \to \mathbb{N}$ as computable which allows to elegantly establish computability theory. The subject of this thesis is to prove Post's theorem in such a setting.

For that purpose, we formalize two definitions of the arithmetical hierarchy. The first definition is due to Odifreddi and is explicit in first-order arithmetic without relying on a concrete model of computation. It classifies formulas in prenex normal form, for which we mechanize a structurally recursive conversion algorithm, via counting the number of quantifier alternations. The second definition is a synthetic version of the Kleene-Mostowski definition and classifies type-theoretic predicates directly. After studying the equivalence of the synthetic to the better known first-order definition, based on axioms from synthetic computability, we continue working with the second definition which is more convenient to establish synthetic results with.

As the second main ingredient, we advance a synthetic definition of Turing reductions by Forster and Kirst who follow an idea by Bauer to synthetic oracle machines by adjusting one of its main components, the continuity requirement. This enables constructive results like the determination of oracle machines solely by a continuous higher-order partial function. Then, we define a synthetic version of the Turing jump as the halting problem of synthetic oracle machines by assuming an enumerator of continuous higher-order partial functions.

Most of our key results are constructive. Only the final proof of Post's theorem relies on the law of excluded middle whose usage we trace back to exactly two places.

Acknowledgements

First and foremost I owe my full gratitude to my advisors Yannick and Dominik[1] for their immense support. I really know that good advising is not self-evident but you have done even much more than I could have hoped for. Thank you very much for motivating the topic of this thesis, for taking so much of your time, for all the helpful and inspiring meetings, and for your numerous, rigorous and valuable feedback. You found the right balance of guidance and let me explore on my own. It was always a pleasure to discuss difficulties with you and I really enjoyed working with you and being one of your supervised students.

Of course, I would like to thank Prof. Smolka for offering me a thesis at his chair and for introducing me to the exciting topic of constructive type theory. You have aroused my enthusiasm and have greatly influenced how I think about proofs. Also thank you to you and to Yannick for reviewing this thesis.

Thank you to the team of the Campus Library for Computer Science and Mathematics for getting the original papers by Uspenskii and Nerode and thank you to Andrej for helping to understand the Uspenskii paper written in Russian language.

I want to thank my friends and family for their unwavering support during my studies. Thank you Simon for convincing me that I had a good chance to get invited to the Forschungstage at MPI by participating in the second round of BwInf which lead to my decision of studying in Saarbrücken. Thank you Janine for your immeasurable mental support, especially during the last few weeks. I apologize for boring you with all my unsorted thoughts and am very grateful that you stood by me. Thank you for taking care of some of my other duties during the last few days and making sure that I got to bet in time. I definitely owe you a cheesecake, maybe even two.

Finally, I want to thank Marc for proofreading this thesis and Janine for all her intermediate feedback and her help with finding the right formulations.

[1]I always name them in alphabetic order of their last names but I owe them both uncomparable much.

Contents

Chapter 1

Introduction

The subject of this thesis is to formalize the arithmetical hierarchy in constructive type theory, set up synthetic oracle computability including a synthetic definition of the Turing jump, and prove Post's theorem in synthetic computability, all mechanized in the proof assistant Coq. In the following, we briefly introduce the historical results and give a context of the synthetic setting in which we formalize them.

In computability theory, the halting problem of whether a Turing machine [56] halts on a given input is the standard example of a problem that is undecidable. But it is also known that there are problems that are relatively harder to decide. In his PhD thesis, Turing came up with the idea of oracle machines [58] extending the model of computation with a hypothetical solver for a potentially undecidable problem. Oracle machines can be used to study the relative decidability of undecidable problems.

An example of a problem that is undecidable relative to the halting problem is totality, the question of whether a Turing machine halts on all inputs. The best one can do is to construct a semi-decider for the complement of totality that queries a hypothetical solver for the halting problem for all inputs.

The game continues, as cofiniteness – whether a Turing machine halts on all but finitely many inputs – is still undecidable relative to totality. Cofiniteness is only semi-decidable relative to totality by querying a hypothetical solver for totality on all input ranges that exclude a finite prefix.

In fact, for every problem, a problem that is harder to decide can be constructed following Kleene and Post [47, 33, 34]: The Turing jump of a problem Q is the halting problem of oracle machines with an oracle for Q. It is semi-decidable but not decidable relative to Q.

Starting with a decidable problem, e.g. whether a number is in the empty set $\varnothing$, repeated jumping gives rise to a hierarchy of undecidable problems. We write $\varnothing^{(n)}$ for the n-th Turing jump starting with $\varnothing^{(0)} := \varnothing$. Oracle machines with an oracle for $\varnothing$ can compute the same functions as normal Turing machines, as querying the oracle does not give any information. Therefore, the Turing jump of $\varnothing$ is Turing equivalent to the halting problem in the sense that the halting problem can be decided by an oracle machine with an oracle for $\varnothing^{(1)}$ and vice versa. Similarly, $\varnothing^{(2)}$, i.e. the Turing jump of $\varnothing^{(1)}$ is Turing equivalent to totality in the sense that totality can be decided relative to $\varnothing^{(2)}$ and vice versa. And ultimately, $\varnothing^{(3)}$, i.e. the Turing jump of $\varnothing^{(2)}$ is Turing equivalent to cofiniteness in the sense that cofiniteness can be decided relative to $\varnothing^{(3)}$ and vice versa.

Another related hierarchy is the arithmetical hierarchy developed independently by Kleene in 1943 [31] and by Mostowski in 1947 [38]. It classifies problems based on the quantifiers that are needed to formulate the problem.

For that, consider the following example. Let $h(M, i, s)$ be the decidable predicate whether a Turing machine M halts on an input i after at most s steps. Given that, the halting problem, totality, and cofiniteness can be formulated as follows.

$$\begin{array}{rl}
\text{Halting Problem} & H(M,i) := \exists s.\ h(M,i,s) \\
\text{Totality} & \mathsf{Tot}(M) := \forall i.\ \exists s.\ h(M,i,s) \\
\text{Cofiniteness} & \mathsf{Cof}(M) := \exists n.\ \forall i > n.\ \exists s.\ h(M,i,s)
\end{array}$$

One can observe that with increasing undecidability, problems need more alternating quantifiers to be formulated. That is not a coincidence, but the result of Post's Theorem [47] which connects the hierarchy of undecidability gained by iterative jumping to the arithmetical hierarchy.

The levels of the arithmetical hierarchy are denoted with $\sum_n$ and $\prod_n$. Predicates that are decidable form the base and are classified both as $\sum_0$ and $\prod_0$. Then, the hierarchy is defined inductively as follows. If an existential quantifier is added in front of a $\prod_n$ predicate it is classified as $\sum_{n+1}$. If a universal quantifier is added in front of a $\sum_n$ predicate it is classified as $\prod_{n+1}$. Given that, a predicate that can be written with n alternating quantifier blocks including a leading existential is classified as $\sum_n$ and if the leading quantifier is universal, it is classified as $\prod_n$.

For example, the halting predicate h from above is both in $\sum_0$ and $\prod_0$ because it is decidable and can be written without quantifiers. The halting problem is in $\sum_1$ because it can be formulated with a single existential quantifier. Totality is in $\prod_2$ because it can be formulated with two quantifiers and a leading universal. Cofiniteness is in $\sum_3$ because it can be formulated with three quantifiers and a leading existential.

Precisely, Post's Theorem [47] states that $\varnothing^{(n)}$ is $\sum_n$-complete in the sense that all other problems in $\sum_n$ are decidable relative to $\varnothing^{(n)}$ and that any $\sum_{n+1}$-problem is semi-decidable relative to a $\prod_n$-problem.

In the original work, Turing, Kleene, Mostowski, and Post all work in a concrete model of computation like Turing machines or μ-recursive functions. But constructions in such a model of computation quickly become too complex to do explicitly. Therefore, they heavily rely on the so-called Church-Turing thesis [7, 56] saying that all algorithms that can be intuitively described are indeed computable in the respective model of computation. This makes their arguments hard to mechanize.

A complementary approach is synthetic computability which was pioneered by Richman, Bridges, and Bauer [48, 6, 3]. It abstracts away the concrete model of computation by axiomatically considering all functions $\mathbb{N} \to \mathbb{N}$ as computable. This is consistent in constructive logic because functions of type $\mathbb{N} \to \mathbb{N}$ defined in constructive logic correspond to programs in some model of computation which results from Kleene's work on realizability [32].

We are working in the Calculus of Inductive Constructions (CIC) [10, 43, 44] implemented in the proof assistant Coq [54]. In Coq, the correspondence between functions and programs is made transparent because – from a user-end of view – functions need to be defined in a functional programming language. To internalize this correspondence and derive common results of computability theory within CIC, an axiom called CT [35] can be assumed that states that all functions are computable in a concrete model of computation.

Most textbooks on computability theory work classically and establish their results upon the law of excluded middle. Richman, Bridges, and Bauer [48, 6, 3] work in settings where they assume the axiom of countable choice which together with LEM allows the construction of non-computable functions [55]. Consequently, assuming LEM in their synthetic settings is contradictory. Assuming the law of excluded middle (LEM) in CIC, however, is consistent [60] even when assuming synthetic axioms like CT [15, 17] and enables classical reasoning. Most of our results are constructive but some of them, like our synthetic proof of Post's Theorem, rely on LEM which we always carefully label.

Many areas of computability theory are synthetically well explored. Since all functions are considered computable, notions like decidability and many-one reductions can be natively expressed in synthetic computability without the need to establish a whole theory on models of computation first. Also, it is known what axioms are needed to show results like the undecidability of the halting problem which synthetically corresponds to the question of whether a partial function is defined at a given value. For that purpose, assuming the enumerability of all partial functions (EPF) [48, 15] suffices which is a weaker axiom than CT.

In contrast, there is no obvious way to express oracle machines synthetically since assuming an oracle-decider of an undecidable problem in synthetic computability is equivalent to assuming falsity and compared to textbooks, there is no model of computation to extend by an additional operation for oracles. Before this thesis, there was only a little work on synthetic oracle computability. In a seminar talk, Bauer [4] proposed a definition of synthetic Turing reductions based on two layers: Intuitively, one layer describes the reduction detached from any computability restrictions. Given any mapping (no computability restrictions) of problem instances to solutions, the Turing reduction transforms it into another mapping of problem instances to solutions (also with no computability restrictions), similarly as an oracle machine can be seen as a transformer of an oracle-solver for one problem to a solver of another problem. The second layer is a higher-order function that takes an oracle as a function (considered computable in synthetic computability) and returns another function (also considered computable). Now, both layers are required to agree on all computable oracles to ensure that the only non-computable operation possible in the first layer is inspecting the oracle. There is however one additional requirement needed. By now, the first layer of the Turing reduction can inspect whether the oracle is computable and then behave differently. To prevent it from doing so, it is crucial to require it to only inspect the oracle locally by allowing only finitely many oracle queries. This is called continuity and discussed later in detail. The idea by Bauer of expressing Turing reductions in two layers was picked up by Forster and Kirst [16]. In addition, Forster has studied synthetic Turing reductions with respect to different refinements like bounded Turing reductions and total bounded Turing reductions and their relationship to truth-table reducibility. To conclude, before this thesis there was a synthetic definition of Turing reductions around but it was only known that it is weaker than truth-table reducibility.

In this thesis, we advance the synthetic definition of Turing reductions by Forster and Kirst to synthetic oracle machines. By adjusting the continuity requirement to a classically equivalent one, we gain constructive results like that there is a constructive one-to-one correspondence between continuous higher-order functions and oracle machines. We study what axioms are needed to express the synthetic halting problem of oracle machines, which is the Turing jump. We validate our synthetic definition of oracle machines by proving Post's Theorem synthetically which connects our notions of synthetic oracle computability to a synthetic definition of the arithmetical hierarchy. Furthermore, we show the synthetic definition of the arithmetical hierarchy equivalent to a textbook presentation by Odifreddi [42] in first-order arithmetic by assuming a CT-like axiom. Altogether, we ratify the existing notion of synthetic Turing reductions but contribute a constructive advancement of the continuity requirement towards a synthetic theory of oracle computability.

We show the eligibility by proving a deep connection of the synthetic notions of oracle computability to a textbook representation of the arithmetical hierarchy by assuming reasonable axioms.

1.1 Outline

In Chapter 2 of this thesis, we introduce the synthetic setting in which we are working and establish some other preliminaries.

In Chapter 3, we present the two definitions of the arithmetical hierarchy that we have mechanized, prove results about them and study their equivalence by assuming a CT-like axiom from synthetic computability.

The first definition is in the language of first-order arithmetic and is better known in the literature. It classifies problems by analyzing the syntax of their defining first-order formula. If the formula is in prenex normal form, i.e. has all quantifiers in the front, the problem is classified by counting the quantifier alternations of its quantifier prefix.

Formalizing the arithmetical hierarchy explicitly in first-order arithmetic is quite heavyweight because it requires mechanizing the syntax and semantics of first-order arithmetic first. For that, we use the framework provided by the *Coq Library for Mechanised First-Order Logic* [30] and will contribute some of our results back.

Every first-order formula can be converted into its prenex normal form, as proved by Skolem [50]. We present a fully mechanized and verified structurally recursive algorithm for prenex normal form conversion.

The second definition of the arithmetical hierarchy is purely formalized in type-theory without any additional dependencies. It can be seen as the synthetic way of defining the arithmetical hierarchy as predicates decidable by a function of type $\mathbb{N} \to \mathbb{B}$ form its base. The hierarchy is abstracted away from a concrete model of computation or concrete first-order formulas and classifies type-theoretic predicates directly. This makes it more convenient to work with and establish results like a synthetic proof of Post's Theorem later on.

In Chapter 4, we study relative decidability in synthetic computability. For that purpose, we advance the synthetic definition of Turing reductions by Forster [16] that was developed in joint work with Kirst and following an idea by Bauer [4] and derive a synthetic notion of oracle machines. Compared to the previous definition by Forster and Kirst, our synthetic oracle machines are solely determined by a higher-order partial function which enables establishing some crucial results constructively without relying on the law of excluded middle.

Then, we present a synthetic definition of the Turing jump as the halting problem of oracle machines by assuming an enumeration of higher-order partial functions and study its properties.

Chapter 3 and 4 are orthogonal and can be read independently. Chapter 5 connects the arithmetical hierarchy to the hierarchy gained by repeated jumping and with that also the previous chapters and presents a synthetic proof of Post's Theorem.

1.2 Contributions

This thesis includes the following main contributions:

- Two mechanized definitions of the arithmetical hierarchy proven equivalent in Section 3.3 by assuming a CT-like axiom (Axiom 3.34).
  - The first definition (Definition 3.14) is directly in first-order arithmetic. It classifies problems by analyzing the syntax of their defining first-order formula by counting the number of quantifier alternations. This definition is better known in the literature and serves as a sanity check for the second synthetic definition.
  - The second definition (Definition 3.21) is purely synthetic and without additional dependencies. It is more convenient to work with to establish results like Post's theorem synthetically as it classifies type-theoretic predicates directly.
- A mechanized and structurally recursive algorithm for prenex normal form conversion (Definition 3.7) that enables a rough upper-bound classification of first-order formulas in the arithmetical hierarchy.
- A definition of synthetic oracle machines (Definition 4.1) that builds on previous work by Forster and Kirst [16] based on an idea by Bauer [4]. We advance the previous definition by adjusting the continuity requirement which enables constructive results like that synthetic oracle machines are solely determinable by a higher-order partial function (Theorem 4.17).
- The identification of a suitable axiom, namely the enumerability of higher-order partial functions (Axiom 4.25) in order to define the, to the best of our knowledge, first synthetic definition of the Turing jump (Definition 4.33).
- A synthetic proof of Post's Theorem (Section 5.2) connecting the synthetic definition of the arithmetic hierarchy to the synthetic definition of the Turing jump. Although the proof relies on classical axioms, we were able to trace their usage such that the heart of the proof remains essentially constructive.

All results of this thesis are mechanized in the accompanying Coq development. In the digital version of this thesis, all definitions, lemmas and theorems are hyperlinked to the respective version in the generated Coq documentation at:

`https://ps.uni-saarland.de/~mueck/bachelor/coqdoc/`

Furthermore, in the digital version of this thesis, the reader can click on all notations to get to the corresponding definitions within the pdf.

Chapter 2

Preliminaries

We are working in the Calculus of Inductive Constructions (CIC) [44] implemented in the proof assistant Coq [54]. In this chapter, we introduce constructive type theory and define some well-known notions and constructions. Then, we present synthetic computability and its basic concept and show how the halting problem is formulated and disproven decidable in synthetic computability. Finally, we extend our setting by classical axioms that we always carefully highlight when used to establish results in this thesis.

2.1 Constructive Type Theory

The Calculus of Inductive Constructions (CIC) [44] features a hierarchy of type universes $\mathbb{T}_1 : \mathbb{T}_2 : \mathbb{T}_3 \dots$ with $\mathbb{T}_1 \subset \mathbb{T}_2 \subset \mathbb{T}_3 \dots$ (for our porpuse, we can ignore the index and only write $\mathbb{T}$) and an impredicative universe of propositions $\mathbb{P} \subset \mathbb{T}_1$.

CIC has dependent types $\forall x : s.\ t$ that quantify over values x of type s that may occur in t. When the return type t does not depend on the value x of the argument type s, simple function types $s \to t$ are derived.

Propositions are also types and values of the type are proofs of the proposition. This works out because of the Curry-Howard correspondence [11, 26]. So proofs of implications are values of the corresponding function type and universally quantified propositions are gained from the dependent types where $\forall$ can be interpreted as a quantifier in the usual logical sense.

More types can be defined inductively either in $\mathbb{P}$ or $\mathbb{T}$. While types in $\mathbb{T}$ may contain computational information, types in $\mathbb{P}$ do not. The elimination restriction only allows doing a case analysis on values of noncomputational types in $\mathbb{P}$ when proving propositions. This enables us to assume classical axioms for proving propositions while computational functions stay constructive.

Recursive functions need to be structurally recursive on the structure of an inductive type. This guarantees termination, so all functions of type $X \to Y$ are total.

Logical negation can be defined as $\neg p : \mathbb{P} := P \to \bot$. We call propositions of type $X \to \mathbb{P}$ predicates and write $\overline{p} := \lambda x.\ \neg p\,x$ for the complement of the predicate p. When appropriate we use set notation for predicates. For example, we write $x \in p := p\,x$ and $p \subseteq q := \forall x.\ p\,x \to\ q\,x$. The subset notation can be extended to functional relations and partial functions introduced in Section 2.2.

Also, a lot of the computational types we are using can be defined inductively:

- Unit: $\mathbb{1} : \mathbb{T} ::= \star : \mathbb{1}$
- Booleans: $\mathbb{B} : \mathbb{T} ::= \mathsf{true} : \mathbb{B} \mid \mathsf{false} : \mathbb{B}$

  A function $\mathtt{!} : \mathbb{B} \to \mathbb{B}$ for boolean negation can be defined by case distinction.
- Natural numbers: $\mathbb{N} : \mathbb{T} := \mathsf{O} : \mathbb{N} \mid \mathsf{S} : \mathbb{N} \to \mathbb{N}$

  We use the usual notation $0 := \mathsf{O}, 1 := \mathsf{S}\ \mathsf{O}, 2 := \mathsf{S}(\mathsf{S}\ \mathsf{O})$, and so on. Operations like $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and comparison predicates like $\leq : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ can be defined recursively.
- Option types: $\mathcal{O}(X : \mathbb{T}) : \mathbb{T} ::= \mathsf{None} \mid \mathsf{Some}(x : X)$
- Products: $\times\,(X : \mathbb{T}, Y : \mathbb{T}) : \mathbb{T} ::= \mathsf{pair} : X \to Y \to X \times Y$
- Sum types: $+\,(X : \mathbb{T}, Y : \mathbb{T}) : \mathbb{T} ::= \mathsf{left} : X \to X + Y \mid \mathsf{right} : Y \to X + Y$

  We call a function of type $\forall x\,y : X.\ (x = y) + (x \neq y)$ an equality decider of X and call the type $X : \mathbb{T}$ discrete if such an equality decider can be defined. We write $\ulcorner x = y \urcorner$ for the function that is true if $x = y$ and false else.
- Dependent pairs: $\Sigma\ (X : \mathbb{T}, p : X \to \mathbb{T}) : \mathbb{T} ::= \mathsf{sig} : \forall x : X.\, p\,x \to \Sigma x.\ p\,x$

  We write $\{x : X \mid p\,x\}$ as notation for the dependent pair $\Sigma x.\ p\,x$.
- Lists: $\mathscr{L}(X : \mathbb{T}) : \mathbb{T} ::= [\,] : \mathscr{L}(X) \mid :: : X \to \mathscr{L}(X) \to \mathscr{L}(X)$

  We write $[x, y, z]$ as notation for the list $x{::}y{::}z{::}[\,]$. A function $|\cdot| : \mathscr{L}(X) \to \mathbb{N}$ determining the length of a list, a function $+\!\!+ : \mathscr{L}(X) \to \mathscr{L}(X) \to \mathscr{L}(X)$ concatenating two lists, and on discrete types X also a membership predicate $\in : X \to \mathscr{L}(X) \to \mathbb{P}$ can be defined recursively.
- Vectors: $\forall (X : \mathbb{T})(k : \mathbb{N}).\, X^k ::= [\,] : X^0 \mid :: : X \to X^n \to X^{\mathsf{S}k}$

Vectors are lists with a fixed length. A function extracting the n-th element of a vector can be defined recursively. We write $\vec{x}[n]$ for the n-th element of the vector $\vec{x} : \mathbb{N}^k$ if $n < k$.

A pairing function $\langle \cdot, \cdot \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ can be easily constructed that acts as a bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$. We write $\lambda\langle n, m\rangle.\ f\,n\,m$ as notation for a function that unembeds its singe argument to two numbers such that $(\lambda\langle n, m\rangle.\ f\,n\,m)\langle n, m\rangle = f\,n\,m$.

2.2 Partial Functions and Functional Relations

We reuse the definition of partial functions by Forster [16].

Definition 2.1 *Partial functions* $f : X \rightharpoonup Y$ *are functions* $X \to \mathsf{part}\,Y$*, where* $\mathsf{part}\,Y$ *is the following monadic structure and* $f\,x \triangleright y$ *means that* f *is defined at* x *to be* y*.*

$\mathsf{part}\,A : \mathbb{T}$	
$\triangleright : \mathsf{part}\,A \to A \to \mathbb{P}$	$x \triangleright a_1 \to x \triangleright a_2 \to a_1 = a_2$
$\mathsf{ret} : A \to \mathsf{part}\,A$	$\mathsf{ret}\,a \triangleright a$
$\gg\!= : \mathsf{part}\,A \to (A \to \mathsf{part}\,B) \to \mathsf{part}\,B$	$x \gg\!= f \triangleright b \leftrightarrow (\exists a.\ x \triangleright a \land f\,a \triangleright b)$
$\mathsf{undef} : \mathsf{part}\,A$	$\nexists\, a.\ \mathsf{undef} \triangleright a$
$\mu : (\mathbb{N} \to \mathbb{B}) \to \mathsf{part}\ \mathbb{N}$	$\mu f \triangleright n \leftrightarrow f\,n = \mathsf{true} \land \forall m{<}n.\ f\,m = \mathsf{false}$
$\mathsf{seval} : \mathsf{part}\,A \to \mathbb{N} \to \mathcal{O}A$	$x \triangleright a \leftrightarrow \exists n.\ \mathsf{seval}\,x\,n = \mathsf{Some}\,a$
	$\mathsf{seval}\,x\,n = \mathsf{Some}\,a \to m \geq n \to \mathsf{seval}\,x\,m = \mathsf{Some}\,a$

While partial functions can be considered computable in synthetic computability, functional relations form the uncomputable counterpart and can be used to express uncomputable mappings.

A function relation $R : X \rightsquigarrow Y := X \to Y \to \mathbb{P}$ is a binary predicate that is functional such that $R\,x\,y_1 \to R\,x\,y_2 \to y_1 = y_2$. We say that a partial function f computes a functional relation R and write $f \triangleright R$ if $\forall x\,y.\ f\,x \triangleright y \leftrightarrow R\,x\,y$. We again use set notation when appropriate and write $\mathsf{Dom}(R)\,x := \exists y.\ R\,x\,y$, $R_1 \subseteq R_2 := R_1\,x\,y \to R_2\,x\,y$ and $R_1 \subseteq_L R_2 := x \in L \to R_1\,x\,y \to R_2\,x\,y$.

We write $f_1 \approx f_2 := \forall x\,y.\ f_1\,x \triangleright y \leftrightarrow f_2\,x \triangleright y$ and $R_1 \approx R_2 := \forall x\,y.\ R_1\,x\,y \leftrightarrow R_2\,x\,y$ for the extensional equality of partial functions and functional relations.

2.3 Synthetic Computability

In synthetic computability [48, 6, 3, 16], all functions are considered computable. This allows to natively express notions such as (semi-)decidability and many-one reductions without talking about a model of computation. We show these definitions in Subsection 2.3.1. In Subsection 2.3.2, we introduce axioms that can be assumed to among others establish negative results like the undecidability of the halting problem in Subsection 2.3.3.

2.3.1 Basic Notions

The following definitions in constructive type theory are by Forster et al. [18, 16].

Definition 2.2 *A predicate* $p : X \to \mathbb{P}$ *is decidable if there is exists a decider* $f : X \to \mathbb{B}$*:* $\mathcal{D}(p) := \exists f.\ \forall x.\ p\,x \leftrightarrow f\,x = \mathsf{true}$

Definition 2.3 *A predicate* $p : X \to \mathbb{P}$ *is semi-decidable if there exists is a semi-decider* $f : X \to \mathbb{N} \to \mathbb{B}$*:* $\mathcal{S}(p) := \exists f.\ \forall x.\ p\,x \leftrightarrow \exists n.\ f\,x\,n = \mathsf{true}$

Alternatively, semi-decidability can be defined using partial functions. We use Definition 2.3 in Chapter 3 and the alternative characterization given by Lemma 2.4 in Chapter 4.

Lemma 2.4 $\mathcal{S}(p) \leftrightarrow \exists g : X \rightharpoonup \mathbb{1}\,.\ p\,x \leftrightarrow g\,x \triangleright \star$

Proof As $\mathcal{O}\,\mathbb{1}$ is isomorphic to $\mathbb{B}$, the construction using seval is straightforward. □

Definition 2.5 *A predicate* $p : X \to \mathbb{P}$ *is many-one reducible to a predicate* $q : A \to \mathbb{P}$ *if there is a many-one reduction* $f : X \to A$ *that translates instances of* p *to instances of* q*:* $p \preceq_m q := \exists f.\ \forall x.\ p\,x \leftrightarrow q\,(f\,x)$

Lemma 2.6 *Many-one reductions are transitive:* $p \preceq_m q \to\ q \preceq_m r \to\ p \preceq_m r$

Proof Composing the two reductions leads to the claimed many-one reduction. □

2.3.2 Axioms of Synthetic Computability

The axiom CT [35, 15] makes explicit that all functions $\mathbb{N} \to \mathbb{N}$ are considered computable in synthetic computability by assuming that there exists a universal function ϕ in a concrete model of computation like μ-recursive functions.

Definition 2.7 *There exists a step-wise interpreter of* μ*-recursive functions* ϕ_μ *where* $\phi_\mu\,i\,x\,n$ *is the output of the* i*-th* μ*-recursive function on input* x *after* n *steps, or* None *if the* i*-th* μ*-function does not terminate after* n *steps:*
$\mathsf{CT} := \forall f : \mathbb{N} \to \mathbb{N}\,.\ \exists i : \mathbb{N}\,.\ \forall x : \mathbb{N}\,.\ \exists n : \mathbb{N}\,.\ \phi_\mu\,i\,x\,n = \mathsf{Some}\,(f\,x)$

We introduce a CT-like axiom in Subsection 3.3.2 for studying the equivalence of our two definitions of the arithmetical hierarchy.

To establish negative results like the synthetic halting problem in Subsection 2.3.3, a weaker and purely synthetic axiom, that there exists an enumeration (can also be seen as a universal partial function) of partial functions can be assumed [48, 15]:

Definition 2.8 *There exists an enumeration* $\theta : \mathbb{N} \to (\mathbb{N} \rightharpoonup \mathbb{N})$ *of partial functions:*
$\mathsf{EPF} := \forall f : \mathbb{N} \rightharpoonup \mathbb{N}.\ \exists i.\ \theta\, i \approx f$

2.3.3 Synthetic Halting Problem

Using the enumeration of partial functions EPF, the synthetic self halting problem can be defined as whether the i-th partial functions to $\mathbb{N} \rightharpoonup \mathbb{N}$ is defined on input i.

Definition 2.9 $\mathcal{K}\, i := \exists n.\ \theta\, i\, i \triangleright n$

Fact 2.10 *The synthetic halting problem is semi-decidable:* $\mathcal{S}(\mathcal{K})$

Proof In the notion of Lemma 2.4, $\lambda i.\ (\theta\, i\, i) \gg= (\lambda n.\ \mathsf{ret}\ \star)$ is a semi-decider. □

Fact 2.11 *The complement of the synthetic halting problem is not semi-decidable:* $\neg\mathcal{S}(\overline{\mathcal{K}})$

Proof By Lemma 2.4, assume that there is a partial function f semi-deciding $\overline{\mathcal{K}}$ such that $\overline{\mathcal{K}}\, i \leftrightarrow f\, i \triangleright \star$. By EPF, let i be given such that $\theta\, i\, i \triangleright n \leftrightarrow (f\, i) \gg= (\lambda u.\ \mathsf{ret}\ 0) \triangleright n$. After unfolding the definition of $\mathcal{K}$, we get the contradiction $\mathcal{K}\, i \leftrightarrow \overline{\mathcal{K}}\, i$. □

2.4 Classical Logic

Some of our results rely on classical axioms. Assuming the law of excluded middle LEM allows classical reasoning. For some of our classical results, Markov's principle MP which is a weaker classical axiom is enough. We always carefully label the usage of classical axioms. Most of our results are constructive.

Definition 2.12 $\mathsf{LEM} := \forall p : \mathbb{P}.\ p \lor \neg p$

Definition 2.13 $\mathsf{MP} := \forall f : \mathbb{N} \to \mathbb{B}.\ \neg(\forall x.\ f\, x = \mathsf{false}) \to \exists n.\ f\, n = \mathsf{true}$

Chapter 3

Arithmetical Hierarchy

The arithmetical hierarchy was developed independently by Kleene in 1943 [31] and Mostowski in 1947 [38]. It classifies predicates on numbers into the classes $\sum_n$ and $\prod_n$ according to the number of quantifiers alternations that are required to formulate them on top of a decidable predicate. The decidability notion in the base of the hierarchy differs in multiple equivalent definitions in the literature. Mostowski defines the hierarchy in first-order arithmetic with formulas decidable according to the ordinary inference rules of first-order logic in the base. Kleene defines the hierarchy with predicates decidable by μ-recursive functions in the base.

In the arithmetical hierarchy, many problems of computability theory can be classified. For example, consider the predicate $h(M, i, s) :=$ *"Turing machine* M *halts on input* i *after* $\leq s$ *steps"* which is a decidable and can be used as a base for more complex predicates. Then, totality – whether a Turing machine M halts on all inputs – can be formulated as $\forall i.\ \exists s.\ h(M, i, s)$. The quantifier prefix consists of two alternating quantifiers with a leading universal quantifier and hence totality can be classified as $\prod_2$.

There is also another definition of the arithmetical hierarchy known in the literature that does not rely on the notion of decidability. For example, Odifreddi [42] mentions as a side note that the arithmetical hierarchy can be solely defined in first-order arithmetic with a quantifier-free formula in the base and classifies all first-order formulas that have all their quantifiers in the front, the so-called prenex normal form. This definition is equivalent to the other ones with a decidable predicate in the base starting with $\sum_1$ and $\prod_1$. The 0-levels of the hierarchies differ as pointed out by Odifreddi [42]. For example, the formula $\exists n.\ \forall k.\ n \geq k$ is composed of a quantifier prefix and a quantifier-free base formula. It consists of two alternating quantifiers with a leading existential quantifier and hence the formula can be classified as $\sum_2$.

In this chapter, we formalize two definitions of the arithmetical hierarchy in constructive type theory. Both hierarchies are defined without relying on decidability in a concrete model of computation, in contrast to most definitions in the literature. In Section 3.1, we formalize a syntactic definition of the arithmetical hierarchy using formulas in first-order arithmetic, with quantifier-free formulas in the base. This definition is known in the literature and serves as a sanity check for our second definition. In Section 3.2, we formalize a semantic definition of the arithmetical hierarchy based on the synthetic notion of decidability, i.e. predicates decidable by a function $\mathbb{N} \to \mathbb{B}$. As the second definition classifies type-theoretic predicates directly, it is more convenient to establish results in synthetic computability later. In Section 3.3, we study the equivalence of both definitions, based on axioms of synthetic computability.

3.1 Arithmetical Hierarchy in First-Order Arithmetic

Each predicate defined by a first-order formula can be classified in the arithmetical hierarchy. For this, the formula needs to be transformed into prenex normal form with all quantifiers in front. Then, the predicate can be classified based on its quantifier prefix:

Let n be the number of quantifier alternations in prenex normal form. If the first quantifier is an existential then the formula is a $\sum_n$-formula else it is a $\prod_n$-formula. If the formula does not contain quantifiers it is both a $\sum_0$, and a $\prod_0$-formula

In Subsection 3.1.1 we will present the inductive definition of first-order formulas that we work with. Then, we present a structurally recursive algorithm for transforming formulas into prenex normal form, in Subsection 3.1.2. The algorithm gives an upper bound classification in the arithmetical hierarchy for every first-order formula. In Subsection 3.1.3, we set up a syntactic definition of the arithmetical hierarchy that classifies type-theoretic predicates that are reflected by a first-order formula.

3.1.1 First-order Arithmetic

We are working with the definition of first-order arithmetic from the *Coq Library for Mechanised First-Order Logic* [30]:

Definition 3.1 (Syntax)
Formulas $\varphi, \psi : \mathcal{F}$ *that contain terms* $t_1, t_2 : \mathcal{T}$ *are defined inductively as follows:*

$t_1, t_2 : \mathcal{T} ::= n \mid x_i \mid t_1 \oplus t_2 \mid t_1 \otimes t_2 \qquad n, i \in \mathbb{N}$

$\varphi, \psi : \mathcal{F} ::= \dot\bot \mid t_1 \doteq t_2 \mid \varphi \dot\land \psi \mid \varphi \dot\lor \psi \mid \varphi \dot\to \psi \mid \dot\forall \varphi \mid \dot\exists \varphi$

For convenience, we write $\diamond$ *as place holder for* $\dot\land$, $\dot\lor$ *or* $\dot\to$ *and define a type* $\mathcal{Q} ::= \dot\forall \mid \dot\exists$.

The syntax presented on paper differs from the syntax of the framework we use in the accompanying Coq development in two aspects.

First, the syntax from the framework generalizes over a signature. This means that the basic predicates and functions are variable and can be instantiated appropriately. For our need, we instantiate the predicates with a single binary predicate $\doteq$ and the functions with two binary functions $\oplus$ and $\otimes$ such that we obtain the arithmetical hierarchy we want. Only the results on prenex normal form conversion in Subsection 3.1.2 can be generalized to an arbitrary signature.

Secondly, the framework does not contain primitive numerals. Instead, it only has a zero constant and some syntax for the successor. In our on-paper presentation, we however write n for the n-th successor of 0 and $n \oplus 1$ for the successor of n. This change can be seen just as notation or else as an equivalent definition (according to the semantics we define below).

Variable binding is implemented using de Bruijn indices [13] where quantifiers do not introduce explicit names for variables they bind. Instead, variables are encoded as numbers that correspond to the number of quantifiers that are in scope between the usage of the variable and the quantifier that binds it. As quantifiers bind wide the whole term after a quantifier is in its scope. For example, $\dot\forall n.\ \dot\exists k.\ n \doteq 2 \otimes k$ is encoded as $\dot\forall \dot\exists\ x_1 \doteq 2 \otimes x_0$. A single variable bound by a single quantifier might have different de Bruijn indices depending on where it is used. For example, the term $\dot\forall n.\ n \doteq 42 \mathbin{\dot\vee} (\dot\exists k.\ n \doteq k)$ is encoded as $\dot\forall\ x_0 \doteq 42 \mathbin{\dot\vee} (\dot\exists\ x_1 \doteq x_0)$ where n is translated to x_0 in its first occurrence and to x_1 in its second occurrence, because there is another quantifier in between. Numbers greater than the number of quantifiers in scope correspond to free variables. For example, the formula $\dot\forall\ x_0 \doteq 42 \mathbin{\dot\wedge} (\dot\exists\ x_1 \doteq x_{27} \oplus x_0)$ can be seen to encode $\dot\forall x.\ x \doteq 42 \mathbin{\dot\wedge} (\dot\exists z.\ x \doteq a \oplus z)$. Note that the free variable x_{27} and the name a have been chosen arbitrarily in the example before. Instead any x_i with $i \geq 2$ (where 2 is the number of bound variables) and similarly any name that is not already bound by a quantifier could have been chosen.

Formally, this encoding is realized by the following two definitions, instantiation of variables and the semantics of formulas.

Definition 3.2 (De Bruijn Instantiation)
Instantiation of a (parallel) substitution $\sigma : \mathbb{N} \to \mathcal{T}$ on terms is defined as follows:

$$n[\sigma] := n \qquad x_i[\sigma] := \sigma\, i \qquad (t_1 \odot t_2)[\sigma] := t_1[\sigma] \odot t_2[\sigma] \qquad \odot \in \{\oplus, \otimes\}$$

and on formulas as follows:

$$\dot\bot[\sigma] := \dot\bot \qquad (t_1 \doteq t_2)[\sigma] := t_1[\sigma] \doteq t_2[\sigma] \qquad (\varphi \diamond \psi)[\sigma] := \varphi[\sigma] \diamond \psi[\sigma]$$

$$(\dot\forall \varphi)[\sigma] := \dot\forall(\varphi[\Uparrow \sigma]) \qquad (\dot\exists \varphi)[\sigma] := \dot\exists(\varphi[\Uparrow \sigma])$$

where $\Uparrow \sigma := x_0; \lambda i.\ (\sigma\ i)[\uparrow]$ *at what* $t; \sigma$ *maps* x_0 *to* t *and* x_{i+1} *to* $\sigma\ i$*, and* $\uparrow := \lambda i.\ x_{i+1}$

We write $\Uparrow^{n} := \Uparrow \circ \ldots \circ \Uparrow$ *and* $\uparrow^{n} := \uparrow \circ \ldots \circ \uparrow$ *for applying* $\Uparrow$ *or* $\uparrow$ *for* n *times.*

The crucial rules are those for the quantifiers. Consider the following example, first with explicit variable names: If we want to instantiate y with $z \oplus 1$ in the formula $\dot\forall x.\ x \oplus y \doteq 0$, the instantiation is $\dot\forall x.\ x \oplus (z \oplus 1) \doteq 0$. But what if we instead instantiate y with $x \oplus 1$? Simply replacing the sub-term to $\dot\forall x.\ x \oplus (x \oplus 1) \doteq 0$ would be wrong as x is a free variable in $x \oplus 1$ and should also be free in the resulting term. To avoid this kind of problem, traditional on-paper proofs often rely on Barendregt's convention [2] that free variables are distinct from bound variables. But as we work formally in a proof assistant, we cannot do so and instead use de Bruijn indices which enable a capture-avoiding substitution.

Reconsider the same example as before but now with de Bruijn indices. The variable y from before corresponds to the first and z to the second free variable, namely x_0 and x_1 in the outer scope. The formula translates to $\dot\forall\ x_0 \oplus x_1 \doteq 0$. The instantiation that substitutes the first free variable with the successor of the second free variable and leaves all other variables in place is $\sigma := \lambda i.\ \texttt{if}\ i = 0\ \texttt{then}\ x_1 \oplus 1\ \texttt{else}\ x_i$. But the first free variable occurs under a quantifier that introduces another variable. The instantiation only instantiates free variables and to this end should not touch the newly introduced variable. With one quantifier in scope, the first free variable is x_1. Consequently the instantiation σ needs to be shifted such that it keeps x_0 and substitutes x_{i+1} with $\sigma\ i$, namely the term by which the i-th free variable would have been instantiated by σ. However, only shifting σ is not good enough yet as we instantiate a variable with an arbitrary term that could also contain another variable. For this reason, the variables in the instantiated term also need to be shifted. This is how the instantiation rule for quantifiers operates. In our example, the resulting formula is $\dot\forall\ x_0 \oplus (x_2 \oplus 1) \doteq 0$.

Definition 3.3 (Semantics) *Term evaluation* $[\![\cdot]\!]_\rho : \mathcal{T} \to \mathbb{N}$ *and formula satisfaction* $\rho \vDash \cdot : \mathcal{F} \to \mathbb{P}$ *on an environment* $\rho : \mathbb{N} \to \mathbb{N}$ *is defined as follows:*

$$[\![n]\!]_\rho := n \qquad [\![x_i]\!]_\rho := \rho\ i \qquad [\![t_1 \oplus t_2]\!]_\rho := [\![t_1]\!]_\rho + [\![n_2]\!]_\rho \qquad [\![t_1 \otimes t_2]\!]_\rho := [\![t_1]\!]_\rho \cdot [\![n_2]\!]_\rho$$

$$\rho \vDash \dot\bot := \bot \qquad \rho \vDash t_1 \doteq t_2 := [\![t_1]\!]_\rho = [\![t_2]\!]_\rho \qquad \rho \vDash \varphi \dot\land \psi := (\rho \vDash \varphi) \land (\rho \vDash \psi)$$

$$\rho \vDash \varphi \dot\lor \psi := (\rho \vDash \varphi) \lor (\rho \vDash \psi) \qquad \rho \vDash \varphi \dot\to \psi := (\rho \vDash \varphi) \to (\rho \vDash \psi)$$

$$\rho \vDash \dot\forall \varphi := \forall n.\ (n;\rho) \vDash \varphi \qquad \rho \vDash \dot\exists \varphi := \exists n.\ (n;\rho) \vDash \varphi$$

where $n;\rho$ *is the environment that maps* x_0 *to* n *and* x_{i+1} *to* $\rho\ i$.

Note that in the standard Tarski semantics we are using, all connectives are just reflected into the type-theoretic level. For that, we rely on the *Coq Library for Mechanised First-Order Logic* [30].

Again, we slightly differ from the framework in our on-paper presentation. As mentioned above, the framework describes a more general setting where a signature abstracts the basic predicates and functions in the syntax. Accordingly, the semantics is defined with respect to a model that gives an interpretation of the basic predicates and functions. For our purpose, we rely on the standard model and interpret the predicate $\dot{=}$ and the functions $\oplus$ and $\otimes$ with their corresponding type-theoretic equivalent.

3.1.2 Prenex Normal Form

Every first-order formula can be transformed to an equivalent formula with all quantifiers in the front, the so-called prenex normal form, or PNF for short. As far as we have found out, this normal form was first formalized and proved by Skolem in 1920 [50] but is attributed to Kuratowski and Tarski [36] by Odifreddi [42]. The rules needed for transforming a formula into prenex normal form seemed to be folklore already during that time.

Although we only present the results of this subsection in the standard model of first-order arithmetic on paper, all definitions, lemmas and theorems easily generalize to a general model with other basic predicates and functions as we prove in the accompanying Coq development.

Definition 3.4 (Prenex Normal Form)
We define a prenex normal form predicate $\mathsf{PNF} : \mathcal{F} \to \mathbb{P}$ *inductively. A* PNF*-formula is a quantifier-free formula with quantifiers stacked to the front:*

$$\frac{\mathsf{PNF}\ \varphi}{\mathsf{PNF}\ (\dot\forall \varphi)} \qquad \frac{\mathsf{PNF}\ \varphi}{\mathsf{PNF}\ (\dot\exists \varphi)} \qquad \frac{\mathsf{noQuant}\ \varphi}{\mathsf{PNF}\ \varphi}$$

$$\frac{}{\mathsf{noQuant}\ \dot\bot} \qquad \frac{}{\mathsf{noQuant}\ (t_1 \dot{=} t_2)} \qquad \frac{\mathsf{noQuant}\ \varphi \qquad \mathsf{noQuant}\ \psi}{\mathsf{noQuant}\ (\varphi \diamond \psi)}$$

for all binary operators $\diamond \in \{\dot\land, \dot\lor, \dot\to\}$

In 3.4 textbooks [42, 49], for each operator a rule is given to pull out the quantifiers by one level. Some of the rules only hold classically.

Lemma 3.5 *All these rules hold in intuitionistic logic for an arbitrary environment* ρ*:*

$$\begin{array}{rr}
\rho \vDash ((\dot{\forall}\varphi)\dot{\wedge}\psi) \overset{\cdot}{\leftrightarrow} (\dot{\forall}(\varphi\dot{\wedge}\psi[\uparrow])) & \rho \vDash ((\dot{\exists}\varphi)\dot{\wedge}\psi) \overset{\cdot}{\leftrightarrow} (\dot{\exists}(\varphi\dot{\wedge}\psi[\uparrow])) \\
\rho \vDash (\varphi\dot{\wedge}(\dot{\forall}\psi) \overset{\cdot}{\leftrightarrow} (\dot{\forall}(\varphi[\uparrow]\dot{\wedge}\psi)) & \rho \vDash (\varphi\dot{\wedge}(\dot{\exists}\psi)) \overset{\cdot}{\leftrightarrow} (\dot{\exists}(\varphi[\uparrow]\dot{\wedge}\psi)) \\
\rho \vDash ((\dot{\forall}\varphi)\dot{\vee}\psi) \overset{\cdot}{\rightarrow} (\dot{\forall}(\varphi\dot{\vee}\psi[\uparrow])) & \rho \vDash ((\dot{\exists}\varphi)\dot{\vee}\psi) \overset{\cdot}{\leftrightarrow} (\dot{\exists}(\varphi\dot{\vee}\psi[\uparrow])) \\
\rho \vDash (\varphi\dot{\vee}(\dot{\forall}\psi)) \overset{\cdot}{\rightarrow} (\dot{\forall}(\varphi[\uparrow]\dot{\vee}\psi)) & \rho \vDash (\varphi\dot{\vee}(\dot{\exists}\psi)) \overset{\cdot}{\leftrightarrow} (\dot{\exists}(\varphi[\uparrow]\dot{\vee}\psi)) \\
\rho \vDash ((\dot{\forall}\varphi) \overset{\cdot}{\rightarrow} \psi) \overset{\cdot}{\leftarrow} (\dot{\exists}(\varphi \overset{\cdot}{\rightarrow} \psi[\uparrow])) & \rho \vDash ((\dot{\exists}\varphi) \overset{\cdot}{\rightarrow} \psi) \overset{\cdot}{\leftrightarrow} (\dot{\forall}(\varphi \overset{\cdot}{\rightarrow} \psi[\uparrow])) \\
\rho \vDash (\varphi \overset{\cdot}{\rightarrow} (\dot{\forall}\psi)) \overset{\cdot}{\leftrightarrow} (\dot{\forall}(\varphi[\uparrow] \overset{\cdot}{\rightarrow} \psi)) & \rho \vDash (\varphi \overset{\cdot}{\rightarrow} (\dot{\exists}\psi)) \overset{\cdot}{\leftarrow} (\dot{\exists}(\varphi[\uparrow] \overset{\cdot}{\rightarrow} \psi))
\end{array}$$

The remaining directions hold when assuming LEM*.*

Proof All proofs are straightforward as the semantics reflects the type-theoretic level. □

The rules proved in Lemma 3.5 give rise to an algorithm for converting a formula into prenex normal form that Mostowski – in his paper defining the arithmetical hierarchy [38] – revers to as "Kuratowski-Tarski method" [36] and Rogers names "Tarski-Kuratowski algorithm" [49]. The algorithm is non-deterministic as also the PNF of a formula is not unique. For example, the PNF of $(\dot{\exists}\dot{\forall}\varphi) \wedge (\dot{\exists}\psi)$ depends on whether the left or right quantifiers are pulled out first. In our implementation of a PNF-conversion algorithm, we apply the rules in a fixed order such that the algorithm becomes deterministic.

The algorithm – as derived from the rules – is terminating but not structurally recursive. For example, $(\dot{\exists}\dot{\forall}\varphi) \wedge (\dot{\exists}\psi)$ can be transformed by a single rule to $\dot{\exists}((\dot{\forall}\varphi) \wedge (\dot{\exists}\psi))$ and then $(\dot{\forall}\varphi) \wedge (\dot{\exists}\psi)$ needs to be converted to PNF recursively, which is not structurally smaller. In Coq only terminating algorithms can be defined. For that reason, structurally recursive algorithms are desirable because termination is obvious and does not need to be proven separately. In order to be structurally recursive, two PNF formulas that are composed together with a binary operator need to be converted to a PNF formula in a single step.

We propose a structurally recursive algorithm that computes the quantifier prefix as a list and the quantifier-free suffix formula separately. When attaching the quantifiers to the quantifier-free formula, we get the prenex normal of the initial formula.

Definition 3.6 *We define a function* $\overline{\cdot} : \mathscr{L}(\mathcal{Q}) \to \mathscr{L}(\mathcal{Q})$ *that inverts a list of quantifiers and another function* $\_ \mathbin{\dot{+\!\!+}} \_ : \mathscr{L}(\mathcal{Q}) \to \mathcal{F} \to \mathcal{F}$ *that stacks a list of quantifiers to the front of a formula.*

$$\begin{aligned}
\overline{[]} &:= [] & [] \dot{+\!\!+}\ \varphi &:= \varphi \\
\overline{\dot\forall :: L} &:= \dot\exists :: \overline{L} & (\dot\forall :: L) \dot{+\!\!+}\ \varphi &:= \dot\forall(L \dot{+\!\!+}\ \varphi) \\
\overline{\dot\exists :: L} &:= \dot\forall :: \overline{L} & (\dot\exists :: L) \dot{+\!\!+}\ \varphi &:= \dot\exists(L \dot{+\!\!+}\ \varphi)
\end{aligned}$$

Definition 3.7 (PNF Conversion) *We define two auxillary functions* $c_L : \mathcal{F} \to \mathscr{L}(\mathcal{Q})$ *and* $c_F : \mathcal{F} \to \mathcal{F}$ *that determine the quantifier prefix and the quantifier-free suffix of the prenex normal form of a formula.*

$$\begin{aligned}
c_L\ \dot\bot &:= [] & c_F\ \dot\bot &:= \dot\bot \\
c_L\ (t_1 \doteq t_2) &:= [] & c_F\ (t_1 \doteq t_2) &:= t_1 \doteq t_2 \\
c_L\ (\varphi \dot\land \psi) &:= (c_L\ \varphi) +\!\!+ (c_L\ \psi) & c_F\ (\varphi \diamond \psi) &:= \\
c_L\ (\varphi \dot\lor \psi) &:= (c_L\ \varphi) +\!\!+ (c_L\ \psi) & &\quad (c_F\ \varphi)[\uparrow^{|c_L\ \psi|}] \diamond (c_F\ \psi)[\Uparrow^{|c_L \psi|}(\uparrow^{|c_L\ \varphi|})] \\
c_L\ (\varphi \dot\to \psi) &:= (\overline{c_L\ \varphi}) +\!\!+ (c_L\ \psi) & c_F\ (\dot\forall\ \varphi) &:= c_F\ \varphi \\
c_L\ (\dot\forall\ \varphi) &:= \dot\forall :: (c_L\ \varphi) & c_F\ (\dot\exists\ \varphi) &:= c_F\ \varphi \\
c_L\ (\dot\exists\ \varphi) &:= \dot\exists :: (c_L\ \varphi) & &
\end{aligned}$$

Using c_L *and* c_F*, we then define a function* convert $: \mathcal{F} \to \mathcal{F}$ *that converts a formula into its prenex normal form.*

$$\text{convert}\ \varphi := (c_L\ \varphi) \dot{+\!\!+} (c_F\ \varphi)$$

The trick that makes the algorithm structurally recursive and enables composing two sub-formulas in PNF by a binary operator is to detach the quantifier prefix from the quantifier-free suffix and compute it separately in a list. This allows that the quantifier lists of the two sub-formulas can simply be concatenated in a single step instead of pulling out quantifiers one after another which has caused a non-structurally recursive algorithm. In the case of $\dot\to$, the quantifiers of the premises list need to be inverted (cf. Lemma 3.5).

The handling of the de Bruijn indices needs more explanation.

First, we have detailed look at $\Uparrow^{\ell}(\uparrow^{k})$ which is the function application of the substitution transformer $\Uparrow^{\ell} : (\mathbb{N} \to \mathcal{T}) \to (\mathbb{N} \to \mathcal{T})$ to the argument $\uparrow^{k} : \mathbb{N} \to \mathcal{T}$. The following lemma gives more insight by proving the explicit form.

Lemma 3.8 $\Uparrow^{\ell} \sigma\ i = \begin{cases} x_i & \textit{if}\ i < \ell \\ (\sigma\ (i - \ell)) \uparrow^{\ell} & \textit{if}\ i \geq \ell \end{cases}$

Proof If $i < \ell$, the proof is by induction on i with ℓ generalized. If $i \geq \ell$, the proof is by induction on ℓ with i generalized. □

Corollary 3.9 $\Uparrow^{\ell}(\uparrow^{k})\ i = \begin{cases} x_i & \textit{if}\ i < \ell \\ x_{(i+k)} & \textit{if}\ i \geq \ell \end{cases}$

So $\Uparrow^{\ell}(\uparrow^{k})$ keeps the first ℓ indices in place and shifts the other indices by k.

With that in mind, we have another look at the renaming in the algorithm. The quantifier lists are concatenated such that the quantifiers from the left sub-formula are on the left and those from the right sub-formula are on the right. To ensure that the quantifiers added between the left sub-formula and its quantifiers do not bind anything in the left sub-formula, the de Bruijn indices in the left sub-formula are raised above the newly added quantifiers. From the viewpoint of the right sub-formula, there is no quantifier added in the scope of its quantifiers. So all bound variables are kept in place. However, there are more quantifiers added to the outside, such that the free variables need to be shifted. Let us consider following example:

$$\mathsf{convert}\ ((\dot\forall\dot\forall\varphi)\dot\land(\dot\exists\dot\exists\dot\exists\varphi)) = \dot\forall\dot\forall\dot\exists\dot\exists\dot\exists(\varphi[\uparrow^{3}]\dot\land\psi[\Uparrow^{3}\uparrow^{2}])$$

The indices of the left sub-formula φ are shifted by 3 because there are three additional $\dot\exists$-quantifiers added that should not bind anything in φ. The first three indices $x_0; x_1; x_2$ of ψ are held in place because they are bound by the inner existential quantifiers. The free indices are shifted by 2 because the two additional $\dot\forall$-quantifier should not bind anything in ψ.

The PNF conversion algorithm presented in Definition 3.7 is quite naive and does not minimize the quantifier alternations. For example (for the sake of readability with explicit variable naming) $\mathsf{convert}\ (\dot\exists x.\dot\forall y.\ \varphi) \land (\dot\exists z.\ \psi) = \dot\exists x.\dot\forall y.\dot\exists z.\ (\varphi \land \psi)$ but $\dot\exists x.\dot\exists z.\dot\forall y.\ (\varphi \land \psi)$ would be an equivalent formula in prenex normal form with less quantifier alternations.

This is caused by simply concatenating the two quantifier lists of the sub-formulas of a binary operator. A possible solution would be to merge the quantifier lists in a more intelligent way by optimizing the number of quantifier alternations. We however leave such optimization to future work, as the de Bruijn renaming as well as the verification would be more complicated and would require more work.

Also note that even if the PNF conversion algorithm would optimize quantifier alternations, the algorithm would still only give an upper bound classification of formulas in the arithmetical hierarchy. Finding an equivalent formula with a minimal number of quantifiers is undecidable in general.[1]

The verification of the PNF conversion algorithm consists of showing two properties. First, that the result is actually a PNF formula (Theorem 3.11) and second, that the result is still equivalent to the initial formula (Theorem 3.13).

[1]The Entscheidungsproblem [23] whether a formula is generally valid is known to be undecidable due to Turing [56] and Church [8]. A reduction from the Entscheidungsproblem can be done as follows. Given a formula φ, compute its equivalent formula with minimal quantifiers ψ. Now φ is universally valid if and only if ψ is quantifier-free and equivalent to the true formula.

We start by showing the easier property that the result is actually a PNF formula. For that purpose, we first prove that the resulting formula of c_L is quantifier-free.

Lemma 3.10 noQuant $c_F\, \varphi$

Proof The proof is straightforward by induction on φ, after showing that variable substitution of a quantifier-free formula does not add additional quantifiers. □

Theorem 3.11 PNF (convert φ)

Proof After unfolding the definition of convert, the proof is straightforward by induction on the list and using Lemma 3.10. □

Showing that the result is an equivalent formula is a bit more involved and needs the right lemma for de Bruijn handling.

Lemma 3.12 $(L \dot{+\!\!+}\, \varphi)[\sigma] = L \dot{+\!\!+}\, (\varphi[\Uparrow^{|L|}\sigma])$

Proof The proof is straightforward by induction on L with σ generalized, as each application of the instantiation rule for quantifiers (cf. Definition 3.2) adds a $\Uparrow$. □

Theorem 3.13 *When assuming* LEM, *each formula is equivalent to its* PNF*-conversion:* $\rho \vDash \varphi \dot{\leftrightarrow} (\text{convert } \varphi)$

Proof The proof is by induction on φ. The two base cases are trivial as falsity and equality are converted to itself. The case for quantifiers follows directly from the inductive hypothesis. The proofs for the binary operators are very similar to each other. So we only prove the $\dot{\land}$-case here:

It suffices to show $\rho \vDash (\text{convert } \varphi)\dot{\land}(\text{convert } \psi) \dot{\leftrightarrow} \text{convert } (\varphi \dot{\land} \psi)$ by the inductive hypothesis. Let $L_\varphi := c_L\,\varphi$, $\varphi' := c_F\,\varphi$, $L_\psi := c_L\,\psi$, and $\psi' := c_F\,\psi$. The claim reduces to: $\rho \vDash (L_\varphi \dot{+\!\!+}\, \varphi')\dot{\land}(L_\psi \dot{+\!\!+}\, \psi') \dot{\leftrightarrow} (L_\varphi +\!\!+ L_\psi) \dot{+\!\!+}\, (\varphi'[\uparrow^{|L_\psi|}]\dot{\land}\psi'[\Uparrow^{|L_\psi|}(\uparrow^{|L_\varphi|})])$

We prove the claim by induction on L_φ and in the [ ]-case by another induction on L_ψ, both inductions with ρ, φ' and ψ' quantified. If both lists are empty, the claim is trivial. In the two remaining cases, either a quantifier is pulled out from the right or from the left. Both cases are symmetric but the second one needs a bit more attention on the de Bruijn handling. So we only prove the second one.

Assume $L_\varphi = \dot{\forall} :: L'_\varphi$ (the $\dot{\exists}$-case is analogous). The proof is as follows:

$$\begin{aligned}
& \rho \vDash (\dot{\forall}(L'_\varphi \mathbin{\dot{+\!\!+}} \varphi'))\dot{\wedge}(L_\psi \mathbin{\dot{+\!\!+}} \psi') && \\
\leftrightarrow\ & \rho \vDash \dot{\forall}((L'_\varphi \mathbin{\dot{+\!\!+}} \varphi')\dot{\wedge}(L_\psi \mathbin{\dot{+\!\!+}} \psi')[\uparrow]) && \text{Lem. 3.5} \\
\leftrightarrow\ & \rho \vDash \dot{\forall}((L'_\varphi \mathbin{\dot{+\!\!+}} \varphi')\dot{\wedge}(L_\psi \mathbin{\dot{+\!\!+}} \psi'[\Uparrow^{|L_\psi|}(\uparrow)])) && \text{Lem. 3.12} \\
\leftrightarrow\ & \forall n.\ n;\rho \vDash ((L'_\varphi \mathbin{\dot{+\!\!+}} \varphi')\dot{\wedge}(L_\psi \mathbin{\dot{+\!\!+}} \psi'[\Uparrow^{|L_\psi|}(\uparrow)])) && \text{Def. 3.3} \\
\leftrightarrow\ & \forall n.\ n;\rho \vDash (L'_\varphi +\!\!+ L_\psi) \mathbin{\dot{+\!\!+}} (\varphi'[\uparrow^{|L_\psi|}]\dot{\wedge}\psi'[\Uparrow^{|L_\psi|}(\uparrow^{|L'_\varphi|+1})]) && \text{Ind. hyp.} \\
\leftrightarrow\ & \rho \vDash \dot{\forall}(L'_\varphi +\!\!+ L_\psi) \mathbin{\dot{+\!\!+}} (\varphi'[\uparrow^{|L_\psi|}]\dot{\wedge}\psi'[\Uparrow^{|L_\psi|}(\uparrow^{|L'_\varphi|+1})]) && \text{Def. 3.3}
\end{aligned}$$

In the penultimate step, we apply the induction hypothesis for environment $n;\rho$ and formulas φ' and $\psi'[\Uparrow^{|L_\psi|}(\uparrow)]$. In addition, we merge $\uparrow^{|L'_\varphi|}$ and $\uparrow$.
Also note that in the first step, Lemma 3.5 needs LEM for pulling a $\dot{\forall}$ out of an $\dot{\vee}$ and for pulling any quantifier out of the premises of an $\dot{\rightarrow}$. □

The general case of Theorem 3.13 for an arbitrary semantical model is equivalent to LEM.

3.1.3 Syntactic Definition of the Arithmetical Hierarchy

For the syntactic definition of the arithmetical hierarchy, we first define the hierarchy on formulas.

We define it upwards-closed such that $\sum_n \subseteq \sum_{n+1}$ and $\prod_n \subseteq \prod_{n+1}$ by defining that quantifier-free formulas are $\sum_n$-formulas for all n (and similar for $\prod_n$). As we later only talk about formulas that reflect a predicate, this does not matter (since there is always an equivalent formula with an unused additional quantifier) and is only for convenience.

Additionally, we also allow stacking multiple quantifiers of the same type (without raising complexity). This is also only for convenience as proved for the semantic arithmetical hierarchy in Subsection 3.2.3 which follows for this definition due to the equivalence proved in Section 3.3.

Definition 3.14 (Arithmetical Hierarchy on Formulas)
We define $\sum_n : \mathcal{F} \to \mathbb{P}$ *and* $\prod_n : \mathcal{F} \to \mathbb{P}$ *mutually inductive:*

$$\frac{\mathsf{noQuant}\ \varphi}{\sum_n \varphi} \qquad \frac{\prod_n \varphi}{\sum_{n+1} \dot{\exists}\varphi} \qquad \frac{\sum_{n+1} \varphi}{\sum_{n+1} \dot{\exists}\varphi}$$

$$\frac{\mathsf{noQuant}\ \varphi}{\prod_n \varphi} \qquad \frac{\sum_n \varphi}{\prod_{n+1} \dot{\forall}\varphi} \qquad \frac{\prod_{n+1} \varphi}{\prod_{n+1} \dot{\forall}\varphi}$$

In the base case, quantifier-free formulas are both $\sum_n$- and $\prod_n$-formulas. Adding an $\dot\exists$ to a $\prod_n$-formula adds a quantifier alternation and consequently, the resulting formula is $\sum_{n+1}$. Similarly, adding a $\dot\forall$ to a $\sum_n$-formula adds a quantifier alternation and consequently, the resulting formula is $\prod_{n+1}$. If a formula already is $\sum_{n+1}$ another $\dot\exists$ can be added without increasing the complexity. Similarly, if a formula already is $\prod_{n+1}$ another $\dot\forall$ can be added without increasing the complexity.

To that effect, $\sum_0$- and $\prod_0$-formulas are exactly the quantifier-free formulas. And a $\sum_{n+1}$-formula is allowed to have at least n quantifier alternations with a leading $\dot\exists$. Similarly, a $\prod_{n+1}$-formula is allowed to have at least n quantifier alternations with a leading $\dot\forall$.

The structure of having a quantifier-free formula in the base case and then allowing quantifiers added to the front reminds of the prenex normal form. The following fact shows that the formulas in the arithmetical hierarchy are exactly those in prenex normal form.

Fact 3.15 $\mathsf{PNF}\ \varphi \leftrightarrow \{n : \mathbb{N} \mid \sum_n \varphi + \prod_n \varphi\}$

Proof First, note that $\mathsf{PNF}\ \varphi$ on the left is of type $\mathbb{P}$ and that we have a dependent pair of type $\mathbb{T}$ on the right. By the elimination restriction (cf. Section 2.1), induction on $\mathsf{PNF}\ \varphi$ is not possible by default when constructiong a dependent pair. However, a strong eliminator of $\mathsf{PNF} : \mathcal{F} \to \mathbb{P}$ to $\mathbb{T}$ can be defined by case distinction on the formula φ which is possible because $\mathcal{F} : \mathbb{T}$ and PNF is defined inductively on the structure of formulas. (We construct the eliminator in the corresponding Coq development.)

Then, the actual proof is straightforward by induction using the strong eliminator on $\mathsf{PNF}\ \varphi$, respectively by mutual induction on $\sum_n\ \varphi$ and $\prod_n\ \varphi$ for the other direction. □

Because Fact 3.15 is formulated computationaly with a dependent pair, it gives rise to an algorithm that – together with the PNF-conversion algorithm from Subsection 3.7 – enables an upper bound classification of formulas in the arithmetical hierarchy.

Next, we define a syntactic arithmetical hierarchy, that classifies type-theoretical predicates that are reflected by a formula in first-order arithmetic. For that purpose, we first establish the notion of a reflecting formula.

Definition 3.16 *We call $\varphi : \mathcal{F}$ a reflecting formula for $p : \mathbb{N}^k \to \mathbb{P}$, written $\varphi \equiv p$ if:*

$$\forall v : \mathbb{N}^k .\ \rho_v \vDash \varphi \leftrightarrow p\ v$$

where $\rho_v\ i := \texttt{if}\ i < k\ \texttt{then}\ v[i]\ \texttt{else}\ 0$ is the environment (a function $\mathbb{N} \to \mathbb{N}$) that looks up the value i at the i-th position of the vector or returns a dummy value if it is out of range.

Definition 3.17 (Syntactic Arithmetical Hierarchy)
We lift the arithmetical hierarchy of formulas to predicates $p : \mathbb{N}^k \to \mathbb{P}$ *in type theory:*

$$\textstyle\sum_n^k p := \exists \varphi \equiv p.\ \sum_n \varphi \qquad\qquad \prod_n^k p := \exists \varphi \equiv p.\ \prod_n \varphi$$

Note that the upper index k *of* $\sum_n^k$ *and* $\prod_n^k$ *is the arity of the predicate in the style of Mostowski [38] but is sometimes used differently in the literature. When the arity is arbitrary or clear from the context, we omit it.*

We do not show many properties of the syntactic arithmetical hierarchy here, as they are inherited by the semantic arithmetical hierarchy (Section 3.2) which we prove equivalent in Section 3.3. The equivalence proof however needs a lemma.

Lemma 3.18 $\sum_n^k p \to \sum_{n+\ell}^k p$ *and respectively* $\prod_n^k p \to \prod_{n+\ell}^k p$

Proof Choose the same reflecting formula. The intermediate step $\sum_n \varphi \to \sum_{n+\ell} \varphi$ is straightforward by induction on $\sum_n \varphi$ and similarly for $\prod_n \varphi \to \prod_{n+\ell} \varphi$. □

3.2 Arithmetical Hierarchy in Type Theory

In this section, we present a definition of the arithmetical hierarchy in synthetic computability. k-ary predicates that are computed by functions build the base. Quantifying some arguments of a k-ary predicate leads to a predicate of higher complexity.

In Subsection 3.2.1 we discuss decidable k-ary predicates using which we define the semantic arithmetical hierarchy in Subsection 3.2.2. Then, in Subsection 3.2.3 we prove some properties of the hierarchy.

3.2.1 Decidable Predicates on Vectors

In synthetic computability all functions $\mathbb{N} \to \mathbb{B}$ are considered computable (cf. Section 2.3). Vectors $\mathbb{N}^k$ consist of finitely many natural numbers that can be encoded into a single natural number by Cantor's pairing function. Thus, functions of type $\mathbb{N}^k \to \mathbb{B}$ can be encoded by functions of type $\mathbb{N} \to \mathbb{B}$ and therefore we also consider them computable.

So decidable predicates on vectors are of the following form:

Definition 3.19 *A predicate* $p : \mathbb{N}^k \to \mathbb{P}$ *is decidable if there is a function* $f : \mathbb{N}^k \to \mathbb{B}$ *such that:* $p = (\lambda \vec{n}.\ f\ \vec{n} = \mathsf{true})$.

When working with predicates on vectors, we assume the following axiom that can be derived from functional and propositional extensionality.

Axiom 3.20 (Predicate Extensionality)
$\mathsf{PredExt} := \forall p\ q : \mathbb{N}^k \to \mathbb{P}.\ (\forall \vec{n}.\ p\vec{n} \leftrightarrow q\vec{n}) \to\ p = q$

The axiom assumes that point-wise equivalent predicates on vectors are equal. This becomes handy when applying a constructor of the inductive definition of the semantic arithmetical hierarchy (Definition 3.21, below) because the predicate can be rewritten by a point-wise equivalent one that meets the required syntactical shape. Instead, we could have added another rule to the inductive definition of the hierarchy, that a predicate is in the hierarchy if a point-wise equivalent one is. We discuss this design decision in more detail after defining the semantic arithmetical hierarchy.

3.2.2 Semantic Definition of the Arithmetical Hierarchy

Definition 3.21 (Semantic Arithmetical Hierarchy)
We define $\tilde{\Sigma}^k_n : (\mathbb{N}^k \to \mathbb{P}) \to \mathbb{P}$ and $\tilde{\Pi}^k_n : (\mathbb{N}^k \to \mathbb{P}) \to \mathbb{P}$ mutually inductive:

$$\frac{f : \mathbb{N}^k \to \mathbb{B}}{\tilde{\Sigma}^k_0(\lambda\vec{n}.\ f\vec{n} = \mathsf{true})} \qquad \frac{f : \mathbb{N}^k \to \mathbb{B}}{\tilde{\Pi}^k_0(\lambda\vec{n}.\ f\vec{n} = \mathsf{true})}$$

$$\frac{\tilde{\Pi}^{k+1}_n\ p}{\tilde{\Sigma}^k_{n+1}(\lambda\vec{n}.\ \exists x.\ p(x :: \vec{n}))} \qquad \frac{\tilde{\Sigma}^{k+1}_n\ p}{\tilde{\Pi}^k_{n+1}(\lambda\vec{n}.\ \forall x.\ p(x :: \vec{n}))}$$

Note that the upper index k of $\tilde{\Sigma}^k_n$ and $\tilde{\Pi}^k_n$ is the arity of the predicate in the style of Mostowski [38] but is sometimes used differently in the literature. When the arity is arbitrary or clear from the context, we omit it.

In the base case, predicates that are computed by a function $\mathbb{N}^k \to \mathbb{B}$ are both in $\tilde{\Sigma}_0$ and $\tilde{\Pi}_0$. Quantifying the first argument of a $k{+}1$-ary $\tilde{\Pi}_n$-predicate with an $\exists$-quantifier (in the type-theoretic level) leads to a $\tilde{\Sigma}_{n+1}$-predicate with smaller arity k. Similarly, quantifying the first argument of a $k{+}1$-ary $\tilde{\Sigma}_n$-predicate with an $\forall$-quantifier (in the type-theoretic level) leads to a $\tilde{\Pi}_{n+1}$-predicate with smaller arity k.

Note that compared to our definition of the arithmetical hierarchy on formulas (cf. Definition 3.14) there is no native rule that decidable predicates without quantifiers are in $\tilde{\Sigma}_n$ and $\tilde{\Pi}_n$ for all n and no rule that allows stacking multiple quantifiers of the same kind. Both rules can however be derived as proved in Lemma 3.23 and Lemma 3.26.

The inductive rules are very strict on the exact form of the predicate. For example, the quantified variables always need to be combined with the argument vector to a single vector that then gets passed to an inner predicate. The inner predicate cannot access the quantified variable directly and needs to decompose the vector again.

At this point predicate extensionality (Axiom 3.20) can be used to successively rewrite the predicate being classified such that it fits the syntactical form of the inductive rule.

For example if we want so show $\lambda\vec{n}.\ \exists k.\ \vec{n}[0] = 2 \cdot k \in \tilde{\sum}_1^1$. First of all, we rewrite the type-theoretic predicate using PredExt to $\lambda\vec{n}, \exists k, (\lambda\vec{m}.\ \vec{m}[1] = 2 \cdot \vec{m}[0])(k :: \vec{m})$. Then by definition of the semantic arithmetical hierarchy it suffices to show that $\lambda\vec{m}.\ \vec{m}[1] = 2 \cdot \vec{m}[0] \in \tilde{\prod}_0^2$. Again by PredExt we rewrite the predicate such that it relies on a decidable predicate to $\lambda\vec{m}.\ (\lambda\vec{n}.\ \ulcorner\vec{m}[1] = 2 \cdot \vec{m}[0]\urcorner)(\vec{m}) \in \tilde{\prod}_0^2$ which follows by definition. (Note that evenness is decidable and can be shown to be in $\tilde{\sum}_0$ by giving a function directly.)

Another example of how we work with predicate extensionality is the following lemma.

Lemma 3.22 $\mathcal{S}(p) \leftrightarrow p \in \tilde{\sum}_1$

Proof We prove both directions separately.

→ Assume $\mathcal{S}(p)$ that means that there is a function $f : \mathbb{N}^k \to \mathbb{N} \to \mathbb{B}$ such that $p\ \vec{n} \leftrightarrow \exists \ell.\ f\ \vec{n}\ \ell = \mathsf{true}$. By PredExt it suffices to show $\exists \ell.\ f\ \vec{n}\ \ell = \mathsf{true} \in \tilde{\sum}_n$. Again by PredExt we can rewrite to $\lambda\vec{n}.\ \exists \ell.\ (\lambda\vec{m}.\ (\lambda \ell :: \vec{n}.\ f\ \vec{n}\ \ell)\ \vec{m} = \mathsf{true})(\ell :: \vec{m})$. Which is in the right form such that two inductive rules can be applied successively in order to show that it is a $\tilde{\sum}_1$-predicate.

← Assume $p \in \tilde{\sum}_1$. By inversion there exists a function $f : \mathbb{N}^{k+1} \to \mathbb{B}$ such that $p = (\lambda\vec{n}.\ \exists \ell.\ f\ (\ell :: \vec{n}) = \mathsf{true})$. The claimed semi-decider is $\lambda\vec{n}\ \ell.\ f\ (\ell :: \vec{n})$. □

We continue relying on PredExt but will not do the rewriting explicitly to apply the inductive rules in the subsequent proofs.

As mentioned earlier, instead of assuming predicate extensionality as an axiom in order to make predicates fit the exact form of the inductive rules, we could have also added a rule to the definition of the semantic arithmetical hierarchy that says a predicate is $\tilde{\sum}_n$ if a point-wise equivalent predicate is $\tilde{\sum}_n$, formally $p \in \tilde{\sum}_n \land (\forall\vec{n}.\ p\vec{n} \leftrightarrow q\vec{n}) \to\ q \in \tilde{\sum}_n$ and similarly of $\tilde{\prod}_n$. We have however decided not to do so as this would have added two rules to the definition of the hierarchy that would have needed to be considered in proofs.

3.2.3 Closure Properties of the Semantic Arithmetical Hierarchy

In this subsection, we prove several closure properties of the semantic arithmetical hierarchy like that, it is closed upwards, under many-one reductions and intersection, by adding additional quantifiers of the same type, that bounded quantifiers do not increase the complexity, that it the complement of $\tilde{\sum}_n$ is $\tilde{\prod}_n$ (when assuming LEM), and more.

We start by proving the two laws that were built-in in the inductive definition of the syntactic arithmetical hierarchy but not in the inductive definition of the semantic arithmetical hierarchy and have been claimed to not be necessary as they can be derived. One is Lemma 3.23 the other ist 3.26.

Lemma 3.23 *Decidable predicates are in the semantic arithemtical hierarchy for any* n*:* $\tilde{\sum}^k_n(\lambda\vec{n}.\ f\vec{n} = \mathsf{true})$ *and* $\tilde{\prod}^k_n(\lambda\vec{n}.\ f\vec{n} = \mathsf{true})$.

Proof The idea is to stack quantifiers whose bound variables are not used. The proof is by induction on n with k and f generalized. If the base case for $n = 0$ the rule is within the definition of the hierarchy. The inductive case can be rewritten to $\lambda\vec{n}.\ \exists x.\ (\lambda x :: \vec{n}.\ f\ \vec{n})(x :: \vec{n}) \in \tilde{\sum}^k_{n+1}$ and then follows by the inductive hypothesis and the definition of the hierarchy. □

Corollary 3.24 $\tilde{\sum}_n \subseteq \tilde{\sum}_{n+\ell}$ *and* $\tilde{\prod}_n \subseteq \tilde{\prod}_{n+\ell}$

Proof The proof is by mutual induction on $\tilde{\sum}_n$ and $\tilde{\prod}_n$. The base cases follow by Lemma 3.23. The other cases by applying the respective inductive rule of the hierarchy's definition. □

Next, we prove that stacking multiple quantifiers of the same kind is possible without increasing the complexity. Intuitively this holds because multiple natural numbers can be encoded into a single one by Cantor's pairing function. The idea is to change the decidable predicate at the base such that it first destructs numbers that encode pairs appropriately.

Coming up with the proof shown below, however, needed several attempts. Our first proof was by generalizing a single Cantor pairing by a general embedding function of $\mathbb{N}^\ell$ to $\mathbb{N}^k$. Later we found a shorter and more meaningful proof using that the arithmetical hierarchy is closed under many-one reducibility.

Lemma 3.25 *The semantic arithmetical hierarchy is closed under many-one reducibility. Formally, for two predicates* $p : \mathbb{N}^k \to \mathbb{P}$ *and* $q : \mathbb{N}^\ell \to \mathbb{P}$ *such that* $p \preceq_m q$*:* $q \in \tilde{\sum}^\ell_n \to p \in \tilde{\sum}^k_n$ *and* $q \in \tilde{\prod}^\ell_n \to p \in \tilde{\prod}^k_n$.

Proof The proof is by mutual induction on $q \in \tilde{\sum}^\ell_n$ and $q \in \tilde{\prod}^\ell_n$ with p generalized. We only prove the two $\sum$ cases, because the two cases for $\prod$ are analogous.

In the base case, we know $\lambda\vec{n}.\ f\ \vec{n} = \mathsf{true} \in \tilde{\Sigma}^{\ell}_{0}$, for some f and $p \preceq_m (\lambda\vec{n}.\ f\ \vec{n} = \mathsf{true})$, namely that there is an encoding $e : \mathbb{N}^k \to \mathbb{N}^{\ell}$ such that $p\ \vec{n} \leftrightarrow f\ (e\ \vec{n}) = \mathsf{true}$. By $\mathsf{PredExt}$ it suffices to show $\lambda\vec{n}.\ f\ (e\ \vec{n}) = \mathsf{true} \in \tilde{\Sigma}^{k}_{0}$ which holds by definition.

In the other case, we know $\lambda\vec{n}.\ \exists x.\ q\ (x :: \vec{n}) \in \tilde{\Sigma}^{\ell}_{n+1}$ for some $q \in \tilde{\Pi}^{\ell+1}_{n}$ and $p \preceq_m (\lambda\vec{n}.\ \exists x.\ q\ (x :: \vec{n}))$, namely that there is an encoding $e : \mathbb{N}^k \to \mathbb{N}^{\ell}$ such that $p\ \vec{n} \leftrightarrow \exists x.\ q\ (x :: (e\ \vec{n}))$. By $\mathsf{PredExt}$ is suffices to show $\lambda\vec{n}.\ \exists x.\ q\ (x :: (e\ \vec{n})) \in \tilde{\Sigma}^{k}_{n+1}$ which is by the definition of the hierarchy $\lambda x :: \vec{n}.\ q\ (x :: (e\ \vec{n})) \in \tilde{\Pi}^{k+1}_{n}$. This predicate is many-one reducible to q and thus the claim follows by the inductive hypothesis. □

Lemma 3.26 *Quantifiers of the same kind can be stacked without raising the complexity:* $p \in \tilde{\Sigma}^{k+1}_{n+1} \to \exists x.\ \lambda\vec{n}.\ p(x :: \vec{n}) \in \tilde{\Sigma}^{k}_{n+1}$ *and* $p \in \tilde{\Pi}^{k+1}_{n+1} \to \forall x.\ \lambda\vec{n}.\ p(x :: \vec{n}) \in \tilde{\Pi}^{k}_{n+1}$.

Proof We only show the claim for $\tilde{\Sigma}$ as the proof for $\tilde{\Pi}$ is analogous. We assume $p \in \tilde{\Sigma}^{k+1}_{n+1}$. By inversion there exists a $q \in \tilde{\Pi}^{k+2}_{n}$ with $\lambda\vec{n}.\ \exists x.\ q\ (x :: \vec{n}) \in \tilde{\Pi}^{k+1}_{n}$. Hence it suffices to show $\lambda\vec{n}.\ \exists x_1 x_2.\ q\ (x_1 :: x_2 :: \vec{n}) \in \tilde{\Sigma}^{k}_{n+1}$. By $\mathsf{PredExt}$ this claim can be rewritten to only quantifying a single number that then is destructed by Cantor's pairing function: $\lambda\vec{n}.\ \exists x.\ (\lambda\langle x_1, x_2\rangle :: \vec{n}.\ q\ (x_1 :: x_2 :: \vec{n}))(x :: \vec{n}) \in \tilde{\Sigma}^{k}_{n+1}$. By the definition of the hierarchy it suffices to show $\lambda\langle x_1, x_2\rangle :: \vec{n}.\ q\ (x_1 :: x_2 :: \vec{n}) \in \tilde{\Pi}^{k+1}_{n}$. The claim follows by Lemma 3.25 as obviously $\lambda\langle x_1, x_2\rangle :: \vec{n}.\ q\ (x_1 :: x_2 :: \vec{n}) \preceq_m q$. □

Next, we prove two lemmas that connect $\tilde{\Sigma}$ to $\tilde{\Pi}$. The first lemma states that every $\tilde{\Pi}_n$ predicate is also a $\tilde{\Sigma}_{n+1}$ predicate because it is equivalent to a predicate with an unused $\exists$-Quantifier (and similarly for $\tilde{\Sigma}$ and $\tilde{\Pi}$ swapped). The second lemma states that – in classical logic – $\tilde{\Sigma}_n$ is exactly the complement of $\tilde{\Pi}_n$.

Lemma 3.27 $\tilde{\Pi}^{k}_{n} \subseteq \tilde{\Sigma}^{k}_{n+1}$ *and* $\tilde{\Sigma}^{k}_{n} \subseteq \tilde{\Pi}^{k}_{n+1}$

Proof Assume $p \in \tilde{\Pi}^{k}_{n}$. By Lemma 3.25 also $\lambda x :: \vec{n}.\ p\vec{n} \in \tilde{\Pi}^{k+1}_{n}$, because the two predicates are many-one reducible. Consequently, by definition of the hierarchy $\lambda\vec{n}.\ \exists x.\ (\lambda x :: \vec{n}.\ p\vec{n})(x :: \vec{n}) \in \tilde{\Sigma}^{k}_{n+1}$. The predicate is equivalent to p. Thus, the claim follows by $\mathsf{PredExt}$. The proof of the other claim is analogous. □

Lemma 3.28 *For* $n \leq 1$ *and else assuming* LEM*:* $p \in \tilde{\Sigma}_n \to \overline{p} \in \tilde{\Pi}_n$
For $n = 0$*, for* $n = 1$ *assuming* MP*, and else assuming* LEM*:* $p \in \tilde{\Pi}_n \to \overline{p} \in \tilde{\Sigma}_n$

Proof In the $n = 0$ cases, by inversion there exists a function f that computes p such that such that $p\vec{n} = (f\vec{n} = \mathsf{true})$. By case distinction on $f\vec{n}$, $\neg p\vec{n} \leftrightarrow (f\vec{n} = \mathsf{false})$. Consequently, $\lambda\vec{n}.\ !(f\vec{n})$ decides $\lambda\vec{n}.\ \neg p\vec{n}$ and both claims follow by the definition of the hierarchy.

In the $p \in \tilde{\Sigma}_1$ case, by inversion $p = (\lambda\vec{n}.\ \exists x.\ q(x :: \vec{n}))$ for some $q \in \tilde{\Pi}_0$. We have just shown that consequently $\lambda\vec{n}.\neg q\vec{n} \in \tilde{\Sigma}_0$. The claim follows by $\mathsf{PredExt}$ and the following intuitionistic fact: $\neg\exists x.\ q(x :: \vec{n}) \leftrightarrow \forall x.\ \neg q(x :: \vec{n})$.

In the $p \in \tilde{\Pi}_1$ case, by inversion $p = (\lambda\vec{n}.\ \exists x.\ f(x :: \vec{n}) = \mathsf{true})$ for a function f. Now, MP states that $\neg(\forall x.\ g\ x = \mathsf{false}) \to \exists x.\ \neg(g\ x = \mathsf{true})$ for a function $g : \mathbb{N} \to \mathbb{B}$ (the back direction holds in intuitionistic logic). The claim follows together with $\mathsf{PredExt}$ when choosing $g := \lambda x.\ f(x :: \vec{n})$ for a fixed $\vec{n}$.

For the general case, the proof is by mutual induction on $p \in \tilde{\Sigma}_n$ and $p \in \tilde{\Pi}_n$. The base cases have been shown above. In the $p \in \tilde{\Sigma}_{n+1}$ case, the claim can be rewritten similarly as above and without using classical axioms to $\lambda\vec{n}.\ \forall x.\ \neg q(x :: \vec{n}) \in \tilde{\Pi}_{n+1}$ which reduces to the induction hypothesis by the definition of the hierarchy. The $p \in \tilde{\Pi}_{n+1}$ case is similar, but rewriting the claim form $\lambda\vec{n}.\ \neg(\forall x.\ p(x :: \vec{n})) \in \tilde{\Sigma}_{n+1}$ into $\lambda\vec{n}.\ \forall x.\ \neg p(x :: \vec{n}) \in \tilde{\Sigma}_{n+1}$ relies on LEM. □

We have shown $p \in \tilde{\Pi}_1 \to \overline{\overline{p}} \in \tilde{\Sigma}_1$ by assuming MP in Lemma 3.28. Actually, assuming $\mathcal{S}(p) \to \mathcal{S}(\overline{\overline{p}})$ – which seem to be weaker than MP – would have been enough and is equivalent to the statement for $n = 1$.

The general case for $n \geq 2$ also seems to be weaker than LEM because LEM is about arbitrary predicates and not only about predicates in the arithmetical hierarchy. Akama et al. [1] present an arithmetical hierarchy of classical axioms that seems to be related.

Lemma 3.29 *The semantic arithmetical hierarchy is closed under intersection, formally:* $p \in \tilde{\Sigma}_n \wedge q \in \tilde{\Sigma}_n \to \lambda\vec{n}.\ p\vec{n} \wedge q\vec{n} \in \tilde{\Sigma}_n$ *and* $p \in \tilde{\Pi}_n \wedge q \in \tilde{\Pi}_n \to \lambda\vec{n}.\ p\vec{n} \wedge q\vec{n} \in \tilde{\Pi}_n$

Proof The proof is by induction on n generalized over p and q and followed by inversion on the assumptions. In the base cases, construct a function that returns true if and only if the two functions deciding p and q return true. In the other cases, pull out the quantifiers similarly to the PNF rules in Lemma 3.5. Then we obtain two quantifiers of the same kind. Consequently, the claim follows by Lemma 3.26, the definition of the hierarchy, and the inductive hypothesis. □

Showing that the arithmetical hierarchy is closed under union – when connecting two predicates with a logical or – would be very similar. However, pulling a universal quantifier from a logical or needs classical assumptions (cf. Lemma 3.5).

For proving Post's Theorem later in Chapter 5, we need to show that if $p \in \tilde{\Sigma}_n$ so is the predicate lifted to lists $\lambda L.\ \forall\vec{n} \in L.\ p\vec{n} \in \tilde{\Sigma}_n$.[2] For that, we first prove that

[2]To be exact, this does not type check and we implicitly assume an encoding of vectors into numbers and lists of vectors into natural numbers. Here and in the following, we abuse the notation and treat them as first-class.

adding bounded quantifiers to a predicate does not increase its complexity.

Lemma 3.30 *Bounded universal quantifiers do not increase the complexity, formally*
$p \in \tilde{\sum}_n \to (\lambda N :: \vec{n}.\ \forall x \le N.\ p(x :: \vec{n})) \in \tilde{\sum}_n$
and $p \in \tilde{\prod}_n \to (\lambda N :: \vec{n}.\ \forall x \le N.\ p(x :: \vec{n})) \in \tilde{\prod}_n$

Proof First, observe that adding a bounded quantifier to a decidable predicate remains decidable as checking finitely many values can be done recursively. Then, the proof idea is to push the bounded quantifier all the way down to the decidable base predicate. When lifting a bounded quantifier over a normal quantifier e.g. $\forall x \le N.\ \exists y.\ p(x :: y :: \vec{n})$, we can instead quantify over a vector $\vec{y}$ of N numbers where the x-th number in the vector corresponds to the y associated with the number $x \le N$, e.g. $\exists \vec{y}.\ \forall x \le N.\ p(x :: \vec{y}[x] :: \vec{n})$. □

Lemma 3.31 *Predicates can be lifted to lists:* $p \in \tilde{\sum}_n \to \lambda L.\ \forall \vec{n} \in L.\ p\vec{n} \in \tilde{\sum}_n$

Proof The proof idea is to rewrite the claim such that it has a bounded quantifier: $\lambda L.\ \forall i \le |L|\ .\ p\ (L[i])$ and then apply Lemma 3.30. Formally, the proof needs some more carefulness on encoding and decoding vectors and lists of vectors. The coding does not increase the complexity of the predicate as the hierarchy in closed under many-one reductions by Lemma 3.25. □

3.3 Equivalence of Both Definitions

We have defined two versions of the arithmetical hierarchy, in the previous sections. A syntactic definition explicitly in first-order arithmetic and a semantic definition using notions of synthetic computability. In this section, we study the equivalence of both definitions.

Whenever relating synthetic computability to a concrete model of computation or first-order arithmetic, an axiom is needed that expresses that all functions that are considered computable in synthetic computability are in fact computable in a concrete model of computation. We discuss this in Subsection 3.3.2.

Also, observe that the equivalence does not hold in the base case. There are decidable predicates than cannot be expressed as a first-order formula without quantifiers. For example, consider the decidable evenness predicate on numbers, it can only be expressed as $\dot{\exists} k.\ n \doteq 2 \otimes k$ or $\dot{\forall} k.\ n \doteq 2 \otimes k \oplus 1 \dot{\to} \dot{\bot}$ but not without a quantifier[3]. As a consequence we can only show that our semantic definition includes the syntactic one starting with $\tilde{\sum}_1$ and $\tilde{\prod}_1$, in Subsection 3.3.3. This is an inherent

[3]Evenness can be expressed with a bounded quantifier though, but we do not have bounded quantifiers in the syntactical arithmetical hierarchy.

artifact of defining the syntactic hierarchy with a quantifier-free formula in the base and was observed e.g. by Odifreddi [42].

The other directions that the syntactic definition includes the semantic one, however, can be shown directly without relying on any axioms and for all $\sum_n$ and $\tilde{\prod}_n$. We do so in Subsection 3.3.1.

3.3.1 Inclusion of the Syntactic in the Semantic Hierarchy

In this subsection we show that every predicate $p \in \sum_n$ that is reflected by a first-order formula $\varphi \in \sum_n$ also is in the same level of the semantic arithmetical hierarchy, that is $p \in \tilde{\sum}_n$, and similar for $\prod_n$.

Intuitively this holds because quantifier-free first-order formulas in the standard model are decidable as they only consist of some computable functions ($\oplus$ and $\otimes$) and a decidable equality predicate on numbers $\dot{=}$ (Lemma 3.32). Quantifiers that are stacked before a quantifier-free base reflect the type-theoretic ones so the quantifier prefix can be translated into the semantic hierarchy one by one.

Lemma 3.32 *Quantifier-free formulas are decidable:* $\mathsf{noQuant}\ \varphi \to\ \forall \rho.\ \mathcal{D}(\rho \vDash \varphi)$

Proof The proof is by induction on φ using the equality decider for numbers in the $\dot{=}$-case. The quantifier cases are contradictory by inversion on $\mathsf{noQuant}\ \varphi$. □

Theorem 3.33 $\sum_n \subseteq \tilde{\sum}_n$ *and* $\prod_n \subseteq \tilde{\prod}_n$

Proof As in the syntactic hierarchy predicates are reflected by formulas, it suffices to show $\varphi \in \sum_n \to \lambda\vec{n}.\ \rho_{\vec{n}} \vDash \varphi \in \tilde{\sum}_n$ and $\varphi \in \prod_n \to \lambda\vec{n}.\ \rho_{\vec{n}} \vDash \varphi \in \tilde{\prod}_n$ by mutual induction on $\varphi \in \prod_n$ and $\varphi \in \sum_n$.

In the base cases, quantifier-free formulas are decidable by Lemma 3.32 so the claim follows for $\tilde{\sum}_0$ and $\tilde{\prod}_0$ and hence for any n by Lemma 3.23.

There are two $\sum_{n+1}$-cases where it remains to show $\lambda\vec{n}.\ \rho_{\vec{n}} \vDash \dot{\exists}\varphi \in \tilde{\sum}_{n+1}$. In one case $\varphi \in \sum_{n+1}$ and in the other case $\varphi \in \prod_n$. By the definition of the semantics, the claim reduces to $\lambda\vec{n}.\ \exists x.\ (x;\rho_{\vec{n}}) \vDash \varphi \in \tilde{\sum}_n$ with a type-theoretic $\exists$-quantifier. The claim follows by Lemma 3.26 in one case and by the semantic definition of the hierarchy in the other case. The $\prod_{n+1}$-cases are analogous. □

3.3.2 Axiom Relating Decidable Predicates to First-Order Arithmetic

The Church-Turing thesis [7, 56] states that all intuitively computable functions are indeed computable in a concrete model of computation (e.g. in the λ-calculus or by a Turing machine). In constructive mathematics, this thesis was made formal by Kreisel [35] as an axiom called CT stating that all functions $\mathbb{N} \to \mathbb{B}$ are computable in a concrete model of computation. CT can be assumed consistently in the setting we are working with [62, 53, 17].

We assume a CT-like axiom, we call first-order arithemetical CT (ACT), that every function of type $\mathbb{N}^k \to \mathbb{B}$ is reflected by a formula $\varphi \in \sum_1$. This axiom is not known in the literature but can be probably derived from a more common version of CT as we discuss at the end of this subsection and therefore seems to be consistent.

Axiom 3.34 $\mathsf{ACT} := \lambda\vec{n}.\ f\ \vec{n} = \mathsf{true} \in \sum_1$

The same for $\prod_1$ follows directly.

Lemma 3.35 $\forall f.\ (\lambda\vec{n}.\ f\ \vec{n} = \mathsf{true} \in \sum_1) \to \forall f.\ (\lambda\vec{n}.\ f\ \vec{n} = \mathsf{true} \in \prod_1)$

Proof We equivalently prove $\lambda\vec{n}.\ !f\ \vec{n} \neq \mathsf{true} \in \prod_1$. And gain from the assumption that $\lambda\vec{n}.\ !(f\ \vec{n}) = \mathsf{true} \in \sum_1$. Now the claim follows as $(\exists x.\ \neg p\ x) \to \neg\forall x.\ p\ x$ holds in intuitionistic logic. □

Now, Axiom 3.34 and Lemma 3.35 give the base case of the inclusion of the semantic in the syntactic hierarchy for $n = 1$.

Corollary 3.36 *When assuming* ACT*:* $\tilde{\sum}_1 \subseteq \sum_1 \wedge \tilde{\prod}_1 \subseteq \prod_1$

Proof Follows by inversion and by the fact that multiple $\dot{\exists}$ (or $\dot{\forall}$ respectively) can be stacked without increasing the complexity. □

We are confident that in future work our axiom ACT can be reduced to a more common variant of CT. In his Bachelor's thesis, Peters [45] derives a similar fact, namely that when assuming a common variant of CT, each synthetically enumerable predicate is in $\sum_1$ for an arithmetical hierarchy built on Q-decidable formulas by combining the work of Larchey-Wendling and Forster [37] – who have proved that μ-enumerable predicates are Diophantine (as a part of their mechanization of Hilbert's tenth problem in Coq) – with the work of Kirst and Hermes [29] – who have mechanized a many-one reduction of Hilbert's tenth problem to the Q-fragment of first-order Peano arithmetic and using representability results by Hermes and Kirst [22]. Combining all those works, ACT could confidently be derived from a common variant of CT.

3.3.3 Inclusion of the Semantic in the Syntactic Hierarchy

In this subsection, we show that every predicate $p \in \tilde{\sum}_n$ is reflected by a first-order formula $\varphi \in \sum_n$, when assuming ACT. The actual proof and consequently also this subsection is very short as the base case already follows by Corollary 3.36.

Theorem 3.37 *When assuming the* ACT*:* $\tilde{\sum}_{n+1} \subseteq \sum_{n+1}$ *and* $\tilde{\prod}_{n+1} \subseteq \prod_{n+1}$

Proof The proof is by mutual induction. The base cases follow by Corollary 3.36 and the CT-like axiom. The other two cases follow directly by the inductive hypothesis as the semantics of first-order formulas reflects the type-theoretic level. □

Chapter 4

Oracle Machines and Turing Jump

In Section 2.3 we have introduced synthetic computability, the setting we are working in. The main idea of synthetic computability was to treat all (partial) functions $\mathbb{N} \rightharpoonup \mathbb{N}$ (and $\mathbb{N} \rightharpoonup \mathbb{B}$) as computable. This works very well when studying only computable problems.

This chapter on the other hand will introduce notions that allow talking about uncomputable problems synthetically. In Section 4.1 we set up synthetic oracle computability and then define a synthetic notion of the Turing jump in Section 4.2.

4.1 Synthetic Oracle Computability

In order to study relative computability, Turing came up with the idea of oracle machines [57] which was developed further by Post [46]. Oracle machines are adapted Turing machines [56] with an additional operation for querying an oracle solver for a given problem.

Having such a model of computation, the notions of oracle semi-decidability and Turing reducibility follow. A problem P is (oracle) semi-decidable relative to a problem Q if P can be semi-decided by an oracle machine with an oracle for Q. Likewise, a problem P is Turing reducible to a problem Q if P can be decided by an oracle machine with an oracle for Q.

Equivalently Kleene [33] builds relative computability on partial μ-recursive functionals. A partial μ-recursive functional is variable in a partial function on numbers and can be obtained from the given function and some primitive functions by composition, primitive recursion and unrestricted μ-recursion. In this context, partial functions are set-theoretic so not necessarily computable. Only when the given function (and all primitive functions) are computable, then also the partial μ-recursive functional is computable.

Although we are using constructions more related to partial μ-recursive functionals, we still speak about "oracles" as we believe that the notion is more intuitive.

In this section, we set up a synthetic version of oracle computability. Our work builds upon a synthetic definition of Turing reducibility by Forster [16] which was developed in joint work with Kirst following an idea by Bauer [4]. We first present the notions and then discuss how they are derived as an adjustment of the definition by Forster and Kirst in Subsection 4.1.5.

4.1.1 Oracle Machines

Oracle machines are the central concept of oracle computability. We present its synthetic definition and then explain the intuition behind it in detail.

Definition 4.1 (Oracle Machines : $\mathbb{M}_{X,Y}^{A,B}$) *Given Types* X, Y *and* Z*:*
An oracle machine $M : \mathbb{M}_{X,Y}^{A,B}$ *consists of a relation transformer and a computational core*

$$M : (A \leadsto B) \to (X \leadsto Y)$$
$$M_c : (A \rightharpoonup B) \to (X \rightharpoonup Y)$$

and needs to fulfill the following properties:

- *core specification*

$$\forall f\, R.\ f \rhd R \to\ M_c\, f \rhd M\, R$$

- *continuity*

$$\forall R\, x\, y.\ M\, R\, x \blacktriangleright y \to\ \exists L \subseteq \mathsf{Dom}(R).\ \forall R_2 \approx_L R.\ M\, R_2\, x \blacktriangleright y$$

Functional relations are the potentially uncomputable counterpart to partial functions. The **relation transformer** takes a functional relation as an input which can be seen as a potentially uncomputable oracle. It then returns another functional relation, a solver for another potentially uncomputable problem. So the relation transformer M provides information about the "observable behavior" of the oracle machine M. We write $M\, R\, x \blacktriangleright y$ in order to express that M halts given an oracle relation R on input x with output y.

But the relation transformer cannot return a solver of an arbitrary uncomputable problem. Intuitively the only non-computable operation an oracle machine is allowed to do is querying the oracle. That is why we require the relation transformer to agree with its **computational core** on computational input according to the **core specification**.

Intuitively, whenever an oracle machine halts (given a fixed oracle and input) it can only have queried the oracle finitely many times and the oracle has answered all queries. Formally **continuity** requires that there is a list containing oracle queries such that the oracle machine halts with the same output when changing the oracle to another oracle that agrees on the queries in the list.

Continuity of the core follows:

Lemma 4.2 *The core of* $M : \mathbb{M}^{A,B}_{X,Y}$ *is continuous, formally:*
$\forall f\, x\, y.\ M_c\ f\ x \triangleright y \rightarrow\ \exists L \subseteq \mathsf{Dom}(f).\ \forall f_2 \approx_L f.\ M_c\ f_2\ x \triangleright y$

Proof Follows by the core specification and continuity of the relation transformer. □

The reader might see synthetic oracle machines as a synthetic version of partial μ-recursive functionals as both are variable in an oracle and computable if the oracle is. The definition of synthetic oracle machines axiomatically requires the relation transformer to be continuous. Partial μ-recursive functionals are monotone and compact by construction which is due to Kleene [33] and Davis [12]. The following fact that Odifreddi [42] attributes to Uspenskii [59] and Nerode [41] states that both notions are equivalent.

Fact 4.3 *Continuity is equivalent to monotonicity and compactness*
$M \text{ is continous} \leftrightarrow (\forall R\, R'.\ R \subseteq R' \rightarrow\ M\ R \subseteq M\ R')$
$\quad \wedge (\forall R\, x\, y.\ M\, R\, x \blacktriangleright y \rightarrow \exists R_L \subseteq R.\ (\exists L.\ \forall a.\ a \in L \leftrightarrow \exists b.\ R_L\ a\ b) \wedge M\ R_L\ y\ z)$

The type $\mathbb{M}^{A,B}_{X,Y}$ of oracle machines is parametric in the argument type A and return type B of the oracle and on the input type X and the output type Y of the actual machine. We omit the types A, B and X when they are clear from the context and only annotate the type parameter Y. We often choose $B := \mathbb{B}$ as we want the oracle to be a decider of a given problem. In subsection 4.1.2 we choose $\mathbb{M}_{\mathbb{B}}$ in order to define Turing reductions and in subsection 4.1.3 we choose $\mathbb{M}_{\mathbb{1}}$ in order to define oracle semi-decidability.

4.1.2 Turing Reductions

A decision problem P is Turing reducible to a decision problem Q if there exists an oracle machine that computes the characteristic relation of P when given an oracle for the characteristic relation of Q.

Definition 4.4 (Turing Reductions) *Given* $P : X \rightarrow \mathbb{P}$ *and* $Q : A \rightarrow \mathbb{P}$ *we define:*
$P \preceq_T Q := \exists M : \mathbb{M}^{A,\mathbb{B}}_{X,\mathbb{B}}.\ M\ Q \approx P$

Here and in the following, we identify predicates $Q : A \to \mathbb{P}$ with their characteristic relation $\lambda a\, b.\, b = \mathsf{true} \leftrightarrow Q\, a$.

Lemma 4.5 $P \preceq_m Q \to P \preceq_T Q$

Proof By definition of $P \preceq_m Q$ there is a function $f : X \to A$ that translates an instance of P to an instance of Q such that $\forall x.\, P\, x \leftrightarrow Q\, (f\, x)$. In order to show Turing reducibility we construct an oracle machine of type $\mathbb{M}_{\mathbb{B}}$. Choose $M\, R\, x := R\, (f\, x)$ and $M_c\, r\, x := r\, (f\, x)$. The core specification is easily fulfilled, for continuity choose the singleton list containing $f\, x$.
It remains to show that $M\, Q\, x = Q\, (f\, x)$ agrees with P which follows from the fact that $P\, x \leftrightarrow Q\, (f\, x)$. □

Using classical assumptions we can prove that each predicate is Turing equivalent to its complement.

Lemma 4.6 $\mathsf{LEM} \to P \preceq_T \overline{P} \land \overline{P} \preceq_T P$

Proof For both directions the same oracle machine $M : \mathbb{M}_{\mathbb{B}}$ suffices. Given by $M\, R\, x \blacktriangleright b := R\, x\, (!b)$ and $M_c\, r\, x := r\, x \gg\!= \lambda b.\, \mathsf{ret}\ !b$.
The core specification follows from the specification of $\gg\!=$ and ret . For continuity, if $M\, R\, x \blacktriangleright b$ the oracle R was queried for x and has answered with $!b$. It remains to show that M computes the characteristic relation, so $M\, P \approx \overline{P}$ which is $\forall x\, b.\, \overline{P}\, x\, (!b) \leftrightarrow P\, x\, b$ and $M\, \overline{P} \approx P$ which is $\forall x\, b.\, P\, x\, (!b) \leftrightarrow \overline{P}\, x\, b$. For both reductions this reduces to showing both $\overline{P}\, x \leftrightarrow \overline{P}\, x$, which is trivial, and $P\, x \leftrightarrow \overline{\overline{P}}\, x$, which is double negation elimination and equivalent to LEM. □

4.1.3 Oracle Semi-decidability

A decision problem P is oracle semi-decidable relative to a decision problem Q if there exists an oracle machine that halts on x if and only if $P\, x$ when given an oracle for the characteristic relation of Q.

The actual output of the oracle machine does not matter. Therefore we choose $\mathbb{1}$ as the return type, having only a single value $\star : \mathbb{1}$.

Definition 4.7 (Oracle Semi-decidability) *Given* $P : X \to \mathbb{P}$ *and* $Q : A \to \mathbb{P}$*:*
$\mathcal{S}_Q(P) := \exists M : \mathbb{M}_{\mathbb{1}}.\ \forall x.\, P\, x \leftrightarrow M\, Q\, x \blacktriangleright \star$

Lemma 4.8 *If the oracle is decidable, oracle semi-decidability and semi-decidability agree:* $\mathcal{S}(P) \to \mathcal{S}_Q(P)$ *and* $\mathcal{D}(Q) \to \mathcal{S}_Q(P) \to \mathcal{S}(P)$

Proof We prove both claims separately:

- For the first claim, by $\mathcal{S}(P)$ and Lemma 2.4 there is a partial function $f : X \rightharpoonup \mathbb{1}$ semi-deciding P such that $\forall x.\ P\,x \leftrightarrow f\,x \triangleright \star$. It is enough to construct an oracle machine of type $\mathbb{M}_{\mathbb{1}}$ that semi-decides P by ignoring its oracle and consulting f. Choose $M\,R\,x \blacktriangleright \star := P\,x$ and $M_c\ R\ x := f\ x$. As M ignores its oracle, continuity is trivial. The core specification reduces to $\forall x.\ P\,x \leftrightarrow f\ x \triangleright \star$ which is exactly the specification of f.
- For the second claim, by definition of $\mathcal{D}(Q)$ there is a function $d : A \to \mathbb{B}$ that decides Q such that $\forall a.\ Q\,a \leftrightarrow d\ a = \mathsf{true}$. Assuming $\mathcal{S}_Q(P)$ there is an oracle machine M that semi-decides P when passing an oracle for Q i.e. $P\,x \leftrightarrow M\,Q\,x \blacktriangleright \star$. Again by Lemma 2.4 showing $S(P)$ reduces to finding a partial function $f : X \rightharpoonup \mathbb{B}$ such that $\forall x.\ P\,x \leftrightarrow f\,x \triangleright \star$. By the core specification $M_c\ (\lambda a.\ \mathsf{ret}\ (d\ a))$ suffices, as d is a total function deciding Q. □

Lemma 4.9 $P_1 \preceq_m P_2 \to \mathcal{S}_Q(P_2) \to \mathcal{S}_Q(P_1)$

Proof The proof essentially does not differ form the respective property of semi-decidability without oracles. Assume $P_1 \preceq_m P_2$, that there is a function f such that $\forall x.\ P_1\ x \leftrightarrow P_2\ (f\,x)$ and assume $\mathcal{S}_Q(P_2)$, that there is an oracle machine $M : \mathbb{M}_{\mathbb{1}}$ that semi-decides P_2 relative to Q such that $\forall y.\ P_2\,y \leftrightarrow M\,Q\,y \blacktriangleright \star$.
The oracle machine $M' : \mathbb{M}_{\mathbb{1}}$ with relation transformer $M' R\,x \blacktriangleright \star := M\,R\,(f\ x) \blacktriangleright \star$ and core ${M'}_c\ r\ x := M_c\ r\ (f\ x)$ is a semi-decider for P_1 relative to Q. The core specification and continuity follow from the respective properties of M. □

Lemma 4.10 $Q_1 \preceq_T Q_2 \to \mathcal{S}_{Q_1}(P) \to \mathcal{S}_{Q_2}(P)$

Proof Assume $Q_1 \preceq_T Q_2$ i.e. that there is an oracle machine $M_T : \mathbb{M}_{\mathbb{B}}$ that computes the characteristic function of Q_1 when given an oracle for Q_2 and assume $\mathcal{S}_{Q_1}(P)$ i.e. that there is an oracle machine $M_S : \mathbb{M}_{\mathbb{1}}$ that semi-decides P relative to Q_1.
The oracle machine $M : \mathbb{M}_{\mathbb{1}}$ with relation transformer $M\,R\,x \blacktriangleright \star := M_S\ (M_T\ R)\,x \blacktriangleright \star$ and core $M_c\ r\ x := M_{S_c}\ (M_{T_c}\ r)\ x$ is a semi-decider for P relative to Q_2. The core specification follows from the respective property of M_S.
For continuity, given R and x such that $M\,R\,x \blacktriangleright \star$ we need to find a list of oracle queries to R. By continuity of M_S there exists a list L_{M_S} of oracle queries to $M_T\ R$ such that $M_T\ R$ halts on all queries. This fact is important here because halting is a precondition for the continuity of M_T which says that there exists a list of queries to R for each query to $M_T\ R$ that halts. The claimed list can be constructed by concatenating the lists obtained by the continuity of M_T for each element in L_{M_S}. □

Corollary 4.11 $\mathsf{LEM} \to \mathcal{S}_Q(P) \leftrightarrow \mathcal{S}_{\overline{Q}}(P)$

Proof A direct consequence of Lemma 4.6 and Lemma 4.10. □

Instead of choosing $\mathbb{1}$ as the return type of oracle machines used for semi-decision, we could have defined oracle semi-decidability with any other inhabited type. Particularly, we could have chosen $\mathbb{B}$ and reused oracle machines with the same type as for Turing reducibility. However, we have chosen $\mathbb{M}_{\mathbb{1}}$ to emphasize at the level of types that the oracle machine is used for semi-decision only and does not carry other information.

Lemma 4.12 *Given predicates* $Q : A \to \mathbb{P}$, $P : X \to \mathbb{P}$ *and an inhabited type* Y*:*
$\mathcal{S}_Q(P) \leftrightarrow \exists M : \mathbb{M}^{A,\mathbb{B}}_{X,Y}.\ \forall x.\ P\,x \leftrightarrow \exists y.\ M\,Q\,x \blacktriangleright y$

Lemma 4.13 *Given predicates* $Q : A \to \mathbb{P}$, $P : X \to \mathbb{P}$ *and a discrete type* Y *with value* y*:*
$\mathcal{S}_Q(P) \leftrightarrow \exists M : \mathbb{M}^{A,\mathbb{B}}_{X,Y}.\ \forall x.\ P\,x \leftrightarrow M\,Q\,x \blacktriangleright y$

4.1.4 Determinacy of Oracle Machines by Their Cores

In this subsection, we study general oracle machines in more detail. The key result is Corollary 4.18 that oracle machines with the same core behave equally. Remarkably, all of the following results are constructive and solely enabled by carefully choosing the continuity requirement of oracle machines.

The key idea is to express the relation transformer only by using the computational core. This is possible because due to continuity the oracle has only been queried finitely many times if the oracle machine halts. In addition, continuity ensures that on all these queries the oracle has either answered true or false.
Therefore, if the oracle machine halts, there exist two lists L_{true} and L_{false} containing all queries whose answer was true or false. From those two lists, a partial function can be constructed that searches for the answer in one of the finite lists and is undefined if no answer is found. This function can be passed to the computational core which agrees with the relation transformer due to monotonicity (Fact 4.3).

We first define a partial function lookup that takes two lists L_{true} and L_{false} containing the elements on which the partial function should return true or false and that is undefined elsewhere. The argument type needs to be discrete in order to check whether a value is in one of the lists.

Definition 4.14 *Given a discrete type* A, *define* $\mathsf{lookup} : \mathscr{L}(A) \to \mathscr{L}(A) \to A \rightharpoonup \mathbb{B}$*:*

$$\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}}\ a := \begin{cases} \mathsf{ret\ true} & \textit{if } a \in L_{\mathsf{true}} \\ \mathsf{ret\ false} & \textit{if } a \in L_{\mathsf{false}} \\ \mathsf{undef} & \textit{else} \end{cases}$$

Lemma 4.15 *Given a discrete type* A, *two disjoint lists* L_{true} *and* L_{false} *and a value* $a : A$*:*

- $\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}}\ a \triangleright \mathsf{true} \leftrightarrow a \in L_{\mathsf{true}}$

- $\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}}\ a \rhd \mathsf{false} \leftrightarrow a \in L_{\mathsf{false}}$

Next, we prove a lemma that allows us to split the list from continuity into two lists L_{true} and L_{false}.

Lemma 4.16 *Given a list* L *and a functional relation* $R : A \rightsquigarrow \mathbb{B}$ *with a discrete argument type* A *such that* $\forall a \in L.\ \exists b.\ R\ a\ b$*, there exist two lists* L_{true} *and* L_{false} *that can be constructed constructively and are fulfilling the following properties:*

- $\forall a \in L_{\mathsf{true}}.\ R\ a\ \mathsf{true}$
- $\forall a \in L_{\mathsf{false}}.\ R\ a\ \mathsf{false}$
- $\forall a \in L.\ a \in L_{\mathsf{true}} \lor a \in L_{\mathsf{false}}$
- $\forall a.\ \neg(a \in L_{\mathsf{true}} \land a \in L_{\mathsf{false}})$

Proof by induction on L.
Base case: Choose $L_{\mathsf{true}} := []$ and $L_{\mathsf{false}} := []$
Inductive step $L = a :: L'$:
By the inductive hypothesis, there are two lists L_{true} and L_{false} that fulfill the properties for L'. By assumption $\exists b.\ R\ ab$. If $b = \mathsf{true}$ add a to L_{true} else to L_{false}. The first three properties are trivial. Disjointness follows by the functionality of the functional relation R. □

Next, we show that the relation transformer of oracle machines (with a discrete oracle argument type) can be expressed only by talking about its computational core.

Theorem 4.17 *Given* $M : \mathbb{M}_{X,Y}^{A,\mathbb{B}}$ *with a discrete oracle argument type* A*, constructively:*

$$M\,R\,x \blacktriangleright y \leftrightarrow \exists L_{\mathsf{true}}\ L_{\mathsf{false}}.\ (\forall a.\ a \in L_{\mathsf{true}} \to\ R\ a\ \mathsf{true}) \land (\forall a.\ a \in L_{\mathsf{false}} \to\ R\ a\ \mathsf{false}) \\ \land M_c\ (\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}})\ x \rhd y$$

Proof We show both directions of the equivalence independently:

→ Assume $M\,R\,x \blacktriangleright y$. By continuity there exists a list L such that $\forall a \in L.\ \exists b.\ R\,x\,b$ and $\forall R' \supseteq_L R.\ M\,R'\,x \blacktriangleright y$. By the first property together with Lemma 4.16 there are two lists L_{true} and L_{false} that fulfill the first two claims and the premises of the lookup specification Lemma 4.15.
It remains to show $M_c\ (\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}})\ x \rhd y$. By the result of Lemmas 4.15 and 4.16 $\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}}$ computes the oracle R on all argument in L. Therefore by continuity and the core specification the claim follows.

- $\leftarrow$ Assume there are two lists L_{true} and L_{false} such that for all elements of L_{true} the oracle R outputs true and false for all elements in L_{false}. Also assume $M_c\ (\mathsf{lookup}\ L_{\mathsf{true}} L_{\mathsf{false}})\ x \triangleright y$.
  Together with the lookup specification Lemma 4.15, $\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}}$ computes R on all arguments in the lists. The claim now follows by the core specification and monotonicity (Fact 4.3). □

Therefore, oracle machines with extensionally equal cores are extensionally equal.

Corollary 4.18 *Oracle machines with same core behave equally:*
Given two oracle machines M *and* $M' : \mathbb{M}_{X,Y}^{A,\mathbb{B}}$ *with a discrete oracle argument type:*
$M_c \approx M'_c \to \forall R.\ M\ R \approx M'\ R$

Also, we can construct an oracle machine for each continuous core.

Lemma 4.19 *Given a continuous partial function* f *with a discrete oracle argument type:*
$\{M : \mathbb{M}_{X,Y}^{A,\mathbb{B}} \mid M_c = f\}$

Proof Choose the relation transformer as specified by Theorem 4.17. Functionality of the relation follows by monotonicity of f (Fact 4.3) on lookup in the concatenation of the given oracle lists. The proof of the core specification is similar to the proof of Theorem 4.17, continuity follows by concatenating L_{true} and L_{false}. □

4.1.5 Comparison to Related Work

Our definition of oracle machines is a slight modification of the Turing functionals that Forster [16] uses in his definition of Turing reducibility developed in joint work with Dominik Kirst. Their definition goes back to an idea by Bauer [4].

Bauer does not work in constructive type theory but in the intuitionistic effective topos. He first came up with the idea of defining synthetic Turing reductions in two layers as a transformer of solvers of potentially uncomputable problems that factors through a computable core on computable oracles.

More precisely, Bauer's oracles $\mathbb{O} := \{(S_0, S_1) \in \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \mid S_0 \cap S_1 = \varnothing\}$ are pairs of disjoint sets of numbers containing partial positive and negative answers. A Turing reduction $r : \mathbb{O} \to \mathbb{O}$ is a map taking an oracle and returning another oracle. When passing a pair of two disjoint enumerable sets also the result is required to be a pair of two disjoint enumerable sets. Similar to our oracle machines r needs to be continuous in the notion Bauer uses i.e. it needs to preserve directed suprema.

The Forster-Kirst definition of Turing functionals was mainly taken as the oracle machines we are working with:

Compared to our oracle machines the return type of Turing functionals is fixed to $\mathbb{B}$. This is however only a cosmetic difference since we later instantiate the return type with $\mathbb{B}$ for Turing reductions and $\mathbb{1}$ for oracle semi-decidability. As discussed in the Remark at the end of Section 4.1.3, we could have also chosen $\mathbb{B}$ as a return type for semi-decidability and therefore used the Forster-Kirst definition as it is, for that matter.

What paid off, however, and enabled the constructive results of Subsection 4.1.4 was adapting the definition of continuity. Forster and Kirst are working with a constructively weaker definition of continuity with a double-negated existential quantifier. They call F continuous, here called weakly continuous if:

$$\forall R\, x.\ \neg\neg\exists L.\ \forall R_2.\ (\forall a \in L\ b.\ R\ ab \to\ R_2\ ab) \to\ \forall y.\ F\ R\ x \blacktriangleright y \to\ F\ R_2\ x \blacktriangleright y$$

The following proofs show that our definition of Turing reducibility $\preceq_T$ implies the Forster-Kirst definition, which we call $\preceq_{FK}$, and that they are equivalent when assuming the law of excluded middle.

Fact 4.20 *Given* $F : (Y \rightsquigarrow W) \to (X \rightsquigarrow Z)$*:* F is continous $\to$ F is weakly continous

Proof Showing a stable claim allows local classical reasoning. The interesting question is $(\exists y.\ F\ R\ x \blacktriangleright y) \lor \neg(\exists z.\ F\ R\ x \blacktriangleright y)$. In the first case, the assumption gives a list that suffices, the second case is contradictory when choosing e.g. the empty list. □

Corollary 4.21 $P \preceq_T Q \to\ P \preceq_{FK} Q$

Fact 4.22 *When assuming* LEM *and given* $F : (Y \rightsquigarrow W) \to (X \rightsquigarrow Z)$*:*
F is weakly continous $\to$ F is continous

Proof LEM allows dropping the double negation and case distinction whether the oracle halts for the elements in the list. Therefore constructing a list that contains all necessary elements on that the oracle halts is possible. This list is sufficient because both definitions only talk about other oracles that agree when the oracle has known the answer. □

Corollary 4.23 LEM $\to\ P \preceq_{FK} Q \to\ P \preceq_T Q$

Strengthening the continuity definition compared to Forster and Kirst has enabled Theorem 4.17 and Corollary 4.18. On the other hand, showing the stronger notion of continuity does not seem to make proofs more complicated. For all oracle machines that we have constructed, knowing that the machine halts was good enough to construct a list of queries.

With the weak continuity notion by Forster and Kirst oracle machines with the same computational core only behave weakly equally:

Fact 4.24 $\forall M\, M' : \mathbb{M}_{FK}\,.\ M_c \approx M'_c \to\ \forall R\, x\, z.\ \neg M\ R\ x \blacktriangleright z \leftrightarrow \neg M'\ R\ x \blacktriangleright z$

4.2 Turing Jump

In this section, we set up a Q-complete predicate, such that every predicate that is semi-decidable relative to Q is many-one reducible to the Q-complete predicate. Such a predicate is called the jump of Q, written Q' and is due to Post and Kleene.

In his preliminary report from 1948 [47], Post has first reported that he has set up a Q-complete predicate. The first construction of a Q-complete predicate using Kleene's T-predicate was published by Kleene in his book "Introduction to metamathematics" in 1952 [33]. In the Kleene-Post paper from 1954 [34], a jump operator was introduced and it was proven that if two predicates are Turing reducible so are their jumps.

In textbooks [42, 9, 51] the Turing jump is defined as the halting problem of oracle machines. In this section, we translate the definition to our synthetic setting. We take the "normal" synthetic halting problem 2.3.3 as an orientation that is built upon an enumeration of partial functions.

In Subsection 4.2.1 we address enumerating oracle machines. In Subsection 4.2.2 we define a synthetic version of the Turing jump and show that it gives rise to a harder problem. In Subsection 4.2.3 we then prove that the jump of Q is Q-complete.

4.2.1 Enumerating Oracle Machines

To formulate the halting problem of oracle machines (aka the Turing jump), we need to have a notion connecting oracle machines to natural numbers, their codes. Similar to the construction of the synthetic halting problem in Subsection 2.3.3 using EPF, we would like to enumerate oracle machines.

As we are only interested in whether an oracle machine halts and not in the actual output, we again choose $\mathbb{1}$ as the return type.

In Subsection 4.1.4 we have seen that oracle machines with the same core behave equally. Therefore an enumerator for computational cores suffices to construct an enumerator for oracle machines. Unfortunately, it is not known how an enumerator for higher-order partial functions can be constructed only by assuming known axioms like EPF.

Axiom 4.25 *There is a function* $\xi : \mathbb{N} \to (\mathbb{N} \rightharpoonup \mathbb{B}) \to (\mathbb{N} \rightharpoonup \mathbb{1})$ *that is:*

- *surjective* $\quad \forall f : (\mathbb{N} \rightharpoonup \mathbb{B}) \to (\mathbb{N} \rightharpoonup \mathbb{1}).\ f$ *is continous* $\to\ \exists i.\ \xi\, i \approx f$
- *continuous* $\quad \forall i\, g\, x.\ \xi\, i\, g\, x \triangleright \star \to \exists L \subseteq \mathsf{Dom}(g).\ \forall g_2 \approx_L g.\ \xi\, i\, g_2\, x \triangleright \star$

First of all, we upgrade ξ to a parametric enumerator. This allows us to construct a parametric enumerator for oracle machines that enables SMN-like reasoning later.

Definition 4.26 $\xi' \langle j, i\rangle\ f\ x := \xi\ i\ f\ \langle j, x\rangle$

Lemma 4.27 ξ' *is parametric:*
$\forall F : \mathbb{N} \to (\mathbb{N} \rightharpoonup \mathbb{B}) \to (\mathbb{N} \rightharpoonup \mathbb{1}).\ (\forall i.\ F\ i$ *is continous*$) \to\ \exists \gamma.\ \forall j.\ \xi'\ (\gamma\ j) \approx F\ j$

Proof Let $F : \mathbb{N} \to (\mathbb{N} \rightharpoonup \mathbb{B}) \to (\mathbb{N} \rightharpoonup \mathbb{1})$ be given. Because ξ is surjective there exists a code i for the function $\lambda f\ \langle j, n\rangle.\ F\ j\ f\ n$ such that $\forall f\ j\ n.\ \xi\ i\ f\ \langle j, n\rangle \triangleright \star \leftrightarrow F\ j\ f\ n \triangleright \star$. Choose $\gamma\ j := \langle j, i\rangle$. Now, given j, f, and n: $\xi'(\gamma\ j)\ f\ n = \xi'\langle j, i\rangle\ f\ n = \xi\ i\ f\ \langle j, n\rangle \approx F\ j\ f\ n$ □

Corollary 4.28 ξ' *is surjective:* $\forall f : (\mathbb{N} \rightharpoonup \mathbb{B}) \to (\mathbb{N} \rightharpoonup \mathbb{1}).\ f$ *is continous* $\to\ \exists i.\ \xi'\ i \approx f$

Proof After choosing $F\ j := f$, the claim follows by Lemma 4.27. □

Lemma 4.29 ξ' *is continuous:* $\forall i\ f\ x.\ \xi'\ i\ f\ x \triangleright \star \to \exists L \subseteq \mathsf{Dom}(f).\ \forall f_2 \approx_L f.\ \xi'\ i\ f_2\ y \triangleright \star$

Proof After unfolding the definition of ξ', the claim follows respectively from the continuity of ξ. □

Next, we construct a (parametric) enumerator for oracle machines.

Definition 4.30 $\Xi : \mathbb{N} \to \mathbb{M}^{\mathbb{N},\mathbb{B}}_{\mathbb{N},\mathbb{1}}$

Proof By Lemma 4.19, there is an oracle machine for each continuous computational core. Therefore, we can upgrade the i-th computational core given by ξ' (that is continuous because of Lemma 4.29) to the i-th oracle machine.

Lemma 4.31 Ξ *is parametric:* $\forall F : \mathbb{N} \to \mathbb{M}^{\mathbb{N},\mathbb{B}}_{\mathbb{N},\mathbb{1}}.\ \exists \gamma : \mathbb{N} \to \mathbb{N}.\ \forall i.\ \Xi\ \gamma_i \approx F\ i$

Proof Follows from the parametricity of ξ' (Lemma 4.27) and Corollary 4.18 saying that oracle machines with the same core behave equally. □

Corollary 4.32 Ξ *is surjective:* $\forall M : \mathbb{M}^{\mathbb{N},\mathbb{B}}_{\mathbb{N},\mathbb{1}}.\ \exists i.\ \Xi\ i \approx M$

Proof After choosing $F\ j := M$, the claim follows by Lemma 4.31. □

4.2.2 Synthetic Turing Jump

We define the Turing jump of Q as all numbers i such that the i-th oracle machine halts on input i.

Definition 4.33 (Turing jump) $Q'\ i := \Xi\ i\ Q\ i \blacktriangleright \star$

Next, we prove that jumping gives rise to a harder problem, namely that the jump Q' is semi-decidable relative to Q, but its complement is not.

Lemma 4.34 $\mathcal{S}_Q(Q')$

Proof The oracle machine given by $M\ R\ i := \Xi\ i\ R\ i$ and $M_c\ f\ i := (\Xi\ i)_c\ f\ i$ is a semi-decider. The core specification and continuity follow from the respective properties of $\Xi\ i$. □

Lemma 4.35 $\neg\mathcal{S}_Q(\overline{Q'})$

Proof Assume that there is an oracle machine M that semi-decides $\overline{Q'}$ when given an oracle for Q, i.e. $\forall i.\ M\ Q\ i \blacktriangleright \star \leftrightarrow \overline{Q'}\ i$. Let i be the code of M, it exists because of the surjectivity of Ξ (Corollary 4.32). By definition $Q'\ i$ if and only if M halts on its own code i. This leads to $\overline{Q'}\ i \leftrightarrow Q'\ i$, a contradiction. □

4.2.3 Completeness of the Turing Jump

In this subsection, we prove that all predicates that are semi-decidable relative to Q are many-one reducible to the Turing jump Q', given any predicate Q.

In order to do so, we first set up an alternative definition of the Turing jump, namely the predicate of all numbers encoding pairs $\langle i, x\rangle$ such that the i-th oracle machine halts on input x, and prove that it is many-one equivalent. The two definitions are related to the halting and self-halting problem of Turing machines in the literature.

Definition 4.36 (Universal jump) $Q^\circ\ \langle i, x\rangle := \Xi\ i\ Q\ x \blacktriangleright \star$

Lemma 4.37 $Q' \equiv_m Q^\circ$

Proof First we show $Q' \preceq_m Q^\circ$, then $Q^\circ \preceq_m Q'$:

- $\lambda i.\ \langle i, i\rangle$ is a many-one reduction.
- This direction needs SMN-like reasoning enabled by the parametricity of the enumerator Ξ (Lemma 4.31).
  Therefore, we first define a parametric family of oracle machines $\mathbb{N} \to \mathbb{M}^{\mathbb{N},\mathbb{B}}_{\mathbb{N},\mathbb{1}}$: Given an index $\langle i, x\rangle$ choose $M_{\langle i,x\rangle}\ R\ y := \Xi\ i\ R\ x$ and $M_{\langle i,x\rangle_c}\ f\ y := \Xi\ i_c\ f\ x$. In other words, $M_{\langle i,x\rangle}$ is the oracle machine that ignores its input and is hard coded to query the i-th oracle machine on input x. The core specification and continuity follow respectively from $\Xi\ i$.
  Now, by Lemma 4.31 there is a function $\gamma : \mathbb{N} \to \mathbb{N}$ that takes numbers encoding pairs $\langle i, x\rangle$ and returns the code of $M_{\langle i,x\rangle}$. As $M_{\langle i,x\rangle}$ halts on its code (or any other input) if and only if the machine with code i halts on x, the function γ translates instances of Q° into instances of Q' and therefore is a sufficient many-one reduction. □

Next we prove that Q' is Q-complete. We make use of the fact that $Q' \equiv_m Q^\circ$ as one direction is more convenient to prove using Q° and the other one directly using Q'.

Lemma 4.38 $\mathcal{S}_Q(P) \leftrightarrow P \preceq_m Q'$

Proof We show $\mathcal{S}_Q(P) \to P \preceq_m Q^\circ$ and $P \preceq_m Q' \to \mathcal{S}_Q(P)$. The claim then follows by $Q' \equiv_m Q^\circ$ (Lemma 4.37) and the transitivity of $\preceq_m$ (Lemma 2.6).

- $\to$ Assume there exists an oracle machine M that semi-decides P when given an oracle for Q. We need to show $P \preceq_m Q^\circ$. Let i be the code of M, it exists because of surjectivity of Ξ (Corollary 4.32). The function $\lambda x.\ \langle i, x\rangle$ is a many-one reduction.
- $\leftarrow$ Assume there exists a function $f_m : \mathbb{N} \to \mathbb{N}$ such that $P\,x \leftrightarrow Q'\,(f_m\ x)$. Now, construct an oracle machine that semi-decides P when given an oracle for Q. Choose $M\ R\ x := \Xi\ (f_m\ x)\ R\ (f_m\ x)$ and $M_c\ f\ x := \Xi\ (f_m\ x)_c\ f\ (f_m\ x)$. The core specification and continuity follow respectively from $\Xi\ (f_m\ x)$. □

Chapter 5

Post's Theorem

In Chapter 3, we have seen how predicates on numbers can be classified based on the arithmetic formulas that define them. In Chapter 4, we have studied oracle decidability and set up notions that allow comparing predicates of numbers based on their degree of undecidability. Post's theorem gives a deep connection between the arithmetical hierarchy and repeated Turing jumps starting with a decidable predicate like the empty predicate.

This connection was first reported by Post in his preliminary report in 1948 [47]. He claimed that he was able to prove that more quantifiers in the arithmetical hierarchy allow predicates of a higher degree of unsolvability. Unfortunately, Post did not publish any proof of his finding, but the preliminary report indicates that the result was established by a connection to repeated jumps starting with the empty predicate.

In 1952, the first proof was published by Kleene in his book "Introduction to Metamathematics" [33] but attributed by him to Post. Kleene did not explicitly talk about jumps starting from the empty predicate. He proved that a predicate is both in $\sum_{n+1}$ and $\prod_{n+1}$ if and only if it is Turing reducible to possibly multiple predicates in $\sum_n$ or $\prod_n$ ([33] Theorem XI^{C}, page 293).

Nowadays, in textbooks e.g. by Odifreddi [42], Cooper [9], and Soare [51], there are usually multiple statements proven that together are called Post's Theorem. The initial statement of Odifreddi and Soare is that a predicate is in $\sum_{n+1}$ if and only if it is semi-decidable relative to a predicate in $\prod_n$ or in $\sum_n$ (or semi-decidable relative to the n-th jump of the empty predicate in case of Cooper). Then it can be shown that the $(n+1)$-th Turing jump is many-one-complete in $\sum_{n+1}$, done so by Soare. Cooper shows the statements in reverse order.

We will present a synthetic proof of Post's theorem following the order of Soare [51] in Section 5.2. Beforehand we formally set up the hierarchy of repeated Turing jumps in Section 5.1.

5.1 Connecting the Arithmetical Hierarchy and Turing Jumps

We define "repeated jumping starting with the empty predicate" as follows:

Definition 5.1 *The* n*-th Turing jump of the empty predicate* $\varnothing^{(n)} : \mathbb{N} \to \mathbb{P}$ *is defined as:*

$$\varnothing^{(0)} := \lambda i.\ \bot$$
$$\varnothing^{(n+1)} := \varnothing^{(n)\prime}$$

The predicate $\varnothing^{(n)}$ is defined on numbers, while the arithmetical hierarchy is defined on predicates of vectors of numbers. This is however not a big deal as vectors can be encoded into a single number using Cantor's pairing function repeatedly. Thus, encodings between vectors and numbers are many-one reductions between the respective predicates and the arithmetical hierarchy is closed under many-one reduction (cf. Lemma 3.25).

In this chapter, we will abstract away from underlying vectors and treat the arithmetical hierarchy as it would classify (unary) predicates on numbers. We encode multi-arity predicates by using Cantor's pairing functions. Keeping the encodings implicit shifts the attention to the main part of the proofs.

In the accompanying Coq development, we cast $\varnothing^{(n)}$ to vectors by only extracting the last element from the vector. During the proofs of Post's Theorem, we explicitly construct the many-one reductions to convert vectors of different arity.

5.2 Synthetic Proof of Post's Theorem

The main theorem that connects oracle machines and the arithmetical hierarchy is Theorem 5.4 stating that a predicate is $\tilde{\sum}_{n+1}$ if and only if it is semi-decidable relative to a predicate in $\tilde{\prod}_n$. We prove both directions separately in Lemma 5.2 and Lemma 5.3. The other statements of Post's Theorem are more or less direct consequences of Theorem 5.4 and lemmas we have proven before.

The following Lemma, that a $\tilde{\sum}_{n+1}$ predicate is semi-decidable relative to a $\tilde{\prod}_n$ predicate can be easily shown by constructing a semi-decider that linearly searches for the existentially quantified value.

Lemma 5.2 $P \in \tilde{\sum}_{n+1} \to \exists Q \in \tilde{\prod}_n.\, \mathcal{S}_Q(P)$

Proof Assume $\mathsf{P} \in \tilde{\sum}_{n+1}$, by inversion $\mathsf{P} = \lambda x.\ \exists y.\ \mathsf{Q}\langle y, x\rangle$ for $\mathsf{Q} \in \tilde{\prod}_n$. We show $\mathcal{S}_\mathsf{Q}(\mathsf{P})$ by constructing an oracle machine that semi-decides P relative to Q by linearly searching x. Choose $\mathsf{M}\,\mathsf{R}\,x \blacktriangleright \star := \exists y.\ \mathsf{R}\ \langle y, x\rangle$ true and $\mathsf{M_c}\ f\ x :=$
$\mu\,\lambda\langle y, m\rangle. \begin{cases} \mathsf{Some}\ \star & \text{if seval } (f\ \langle x, y\rangle)\ m = \mathsf{Some\ true} \\ \mathsf{None} & \text{else} \end{cases}$.
The core specification follows by the specification of μ and seval. For continuity choose the singleton list only containing $\langle y, x\rangle$. Because if M halts then there exists a y such that the oracle halts on $\langle y, x\rangle$ with output true. □

For the other direction, we first want to give some intuition. Intuitively, it states that if there is an oracle machine that semi-decides a predicate then the predicate is only a ∃-quantifier away from the oracle. In the introduction of this thesis in Chapter 1, we relied on the intuition that the halting problem for Turing machines can be expressed as "there exists a number of steps s such that the Turing machine has halted after $\leq s$ steps". Soare [51] extends this intuition to a proof of Theorem 5.4.

This intuition, however, can not be applied to our setting as synthetic oracle machines as introduced in Chapter 4 can not be simulated step-wise. Instead, we extract the proof idea of Odifreddi [42] who exploits compactness and monotonicity of μ-recursive functionals. The idea behind our proof of Theorem 5.4 is that by Theorem 4.17 the behavior of synthetic oracle machines can be expressed only by its core (which is computable) by existentially quantifying the answeres (stored in two lists $\mathsf{L_{true}}$ and $\mathsf{L_{false}}$) to the finitely many oracles queries that the oracle machine does during an execution that halts.

We classically show Lemma 5.3 and consequently all results derived from it by assuming LEM. Still, the classical reasoning of the proofs can be traced back to exactly two usages of LEM, as we discuss at the end of this section.

Lemma 5.3 $\mathsf{LEM} \to \mathsf{P} \in \tilde{\sum}_{n+1} \leftrightarrow \exists \mathsf{Q} \in \tilde{\prod}_n.\ \mathcal{S}_\mathsf{Q}(\mathsf{P})$

Proof Assume $\mathsf{Q} \in \tilde{\prod}_n$ and $\mathcal{S}_\mathsf{Q}(\mathsf{P})$. Then there is an oracle machine M such that $\mathsf{P}\,x \leftrightarrow \mathsf{M}\,\mathsf{Q}\,x \blacktriangleright \star$. By PredExt it suffices to show that $\mathsf{M}\,\mathsf{Q}\,y \blacktriangleright \star$ is in $\tilde{\sum}_{n+1}$. By Theorem 4.17 the behavior of an oracle machine can be described as follows $\lambda x.\ \exists \mathsf{L_{true}}\ \mathsf{L_{false}}.\ (\forall a {\in} \mathsf{L_{true}}.\ \mathsf{Q}\ a) \wedge (\forall a {\in} \mathsf{L_{false}}.\ \overline{\mathsf{Q}}\ a) \wedge \mathsf{M_c}\ (\mathsf{lookup}\ \mathsf{L_{true}}\ \mathsf{L_{false}})\ x \triangleright \star$ which suffices to show in $\tilde{\sum}_{n+1}$. For this purpose, we use[1] a bijection that encodes lists of vectors of natural numbers into natural numbers. Therefore we treat the existentials that quantify lists normally and apply Lemma 3.26 that allows to stack additional quantifiers on a $\tilde{\sum}_{n+1}$ predicate. Because $\tilde{\sum}_{n+1}$ is closed under $\wedge$ (Lemma 3.29) we only need to show that the three predicates are in $\tilde{\sum}_{n+1}$ separately.

[1]Made explicit in the corresponding Coq development.

- $\lambda\,\langle\langle L_{\mathsf{true}}, L_{\mathsf{false}}\rangle, x\rangle.\ \forall a \in L_{\mathsf{true}}.\ Q\ a \in \tilde{\sum}_{n+1}$
  By Lemma 3.31 the bounded quantifier does not increase the complexity. It remains to show $Q \in \tilde{\sum}_{n+1}$ which follows by Lemma 3.27 from the assumption $Q \in \tilde{\prod}_n$.
- $\lambda\,\langle\langle L_{\mathsf{true}}, L_{\mathsf{false}}\rangle, x\rangle.\ \forall a \in L_{\mathsf{false}}.\ \overline{Q}\ a \in \tilde{\sum}_{n+1}$
  By Lemma 3.31 the bounded quantifier does not increase the complexity. It remains to show $\overline{Q} \in \tilde{\sum}_{n+1}$. We know $Q \in \tilde{\prod}_{n+1}$ by Corollary 3.24. The claim follows by LEM and Lemma 3.28.
- $\lambda\,\langle\langle L_{\mathsf{true}}, L_{\mathsf{false}}\rangle, x\rangle.\ M_c\ (\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}})\ x \triangleright \star \in \tilde{\sum}_{n+1}$
  Showing $\exists n.\ \mathsf{seval}\ (M_c\ (\mathsf{lookup}\ L_{\mathsf{true}}\ L_{\mathsf{false}})\ x)\ n = \mathsf{Some}\,\star \in \tilde{\sum}_{n+1}$ suffices, by the specification of seval. The existential does not increase the complexity of a $\tilde{\sum}_{n+1}$ formula, by Lemma 3.26. The claim follows as seval is decidable. □

Theorem 5.4 $\mathsf{LEM} \to\ P \in \tilde{\sum}_{n+1} \leftrightarrow \exists Q \in \tilde{\prod}_n.\ \mathcal{S}_Q(P)$

Proof We have proven both directions separately in Lemma 5.2 and Lemma 5.3. □

As classically a predicate is Turing equivalent to its complement and oracles can be replaced by a Turing equivalent one, we can also show that a predicate is in $\tilde{\sum}_{n+1}$ if and only if it is semi-decidable relative to a predicate in $\tilde{\sum}_n$ as a corollary.

Corollary 5.5 $\mathsf{LEM} \to\ P \in \tilde{\sum}^k_{n+1} \leftrightarrow \exists Q \in \tilde{\sum}^{k+1}_n.\ \mathcal{S}_Q(P)$

Proof By LEM and Lemma 3.28, $\overline{Q} \in \tilde{\sum}_n \leftrightarrow Q \in \tilde{\prod}_n$. Further, by LEM and Corollary 4.11, $\mathcal{S}_Q(P) \leftrightarrow \mathcal{S}_{\overline{Q}}(P)$. Therefore, the claim follows by LEM and Theorem 5.4. □

Next, we prove that the $n+1$-th Turing jump is $\tilde{\sum}_{n+1}$-many-one-complete. This is not true for $\varnothing^{(0)}$ because a many-one reduction would need to encode instances of a given predicate into an instance of the empty predicate but no number fulfills the empty predicate. Consequently, $\varnothing$ is not a many-one-complete decidable predicate.

Theorem 5.6 $\mathsf{LEM} \to\ \varnothing^{(n)} \in \tilde{\sum}_n$ *and* $\mathsf{LEM} \to\ \forall P \in \tilde{\sum}_{n+1}.\ P \preceq_m \varnothing^{(n+1)}$

Proof We prove both claims separately, both by induction on n.

- In the base case, $\varnothing$ is decidable. In the induction step, $\varnothing^{(n+1)} \in \tilde{\sum}_{n+1}$ if it is semi-decidable relative to $\varnothing^{(n)} \in \tilde{\sum}_n$, by employing the induction hypothesis, LEM and Corollary 5.5. The claim follows as $\mathcal{S}_{\varnothing^{(n)}}(\varnothing^{(n)\prime})$ by Lemma 4.34.

- In the base case, given $P \in \tilde{\sum}_1$, we need to show $P \preceq_m \varnothing'$. By Lemma 4.38 it suffices to show $\mathcal{S}_\varnothing(P)$. As $\varnothing$ is decidable, by Lemma 4.8 it suffices to show $\mathcal{S}(P)$. The claim follows by Lemma 3.22 and the assumption $P \in \tilde{\sum}_1$.

  In the induction case, assume $P \in \tilde{\sum}_{n+2}$. By LEM and Corollary 5.5 there is $Q \in \tilde{\sum}_{n+1}$ such that $\mathcal{S}_Q(P)$. We need to show $P \preceq_m \varnothing^{(n+2)}$, so it suffices to show $\mathcal{S}_{\varnothing^{(n+1)}}(P)$ by Lemma 4.38. By Lemma 4.10 and the assumption $\mathcal{S}_Q(P)$ the claim reduces to $Q \preceq_T \varnothing^{(n+1)}$, which follows by Lemma 4.5 because $\varnothing^{(n+1)}$ is $\tilde{\sum}_{n+1}$-complete by the inductive hypothesis. □

In order to fix the $n = 0$ case of Theorem 5.6, we further show for all n that $\varnothing^{(n)}$ is $\tilde{\sum}_n$-Turing-complete.

Corollary 5.7 $\mathsf{LEM} \to \forall P \in \tilde{\sum}_n.\ P \preceq_T \varnothing^{(n)}$

Proof For $n = 0$ the claim is trivial as P is decidable. For $n > 0$ the claim follows by Lemma 4.5 as a corollary of Theorem 5.6. □

As a final corollary, we show that a predicate is in $\tilde{\sum}_{n+1}$ if and only if it is semi-decidable relative to $\varnothing^{(n)}$.

Corollary 5.8 $\mathsf{LEM} \to P \in \tilde{\sum}_{n+1} \leftrightarrow \mathcal{S}_{\varnothing^{(n)}}(P)$

Proof We prove both directions separately.

→ Assume $P \in \tilde{\sum}_{n+1}$ so by LEM and Corollary 5.5 that there is a $Q \in \tilde{\sum}_n$ such that $\mathcal{S}_Q(P)$. By Lemma 4.10 it suffices to show $P \preceq_T \varnothing^{(n)}$ which follows by LEM and Corollary 5.7

← Assume $\mathcal{S}_{\varnothing^{(n)}}(P)$. By LEM and Corollary 5.5 it suffices to show $\varnothing^{(n)} \in \tilde{\sum}_n$ which holds by LEM and Theorem 5.6. □

All proofs in this section require LEM. However, the usage can be traced back to exactly two places. First, in Theorem 5.4 we use that if a predicate is in $\tilde{\sum}_n$ then its complement is in $\tilde{\prod}_n$ (Lemma 3.28) in order to reason that the specification of L_{false} is in $\tilde{\sum}_{n+1}$. Secondly, in Corollary 5.5 we again use Lemma 3.28 and that $P \equiv_T \overline{P}$ (Lemma 4.6) in order to reason that a predicate is semi-decidable in a $\tilde{\sum}_n$ predicate if and only if it is semi-decidable in a $\tilde{\prod}_n$ predicate. Besides those two usages of LEM, our proofs are constructive.

Chapter 6

Discussion

We conclude this thesis with a brief discussion of our results.

In Chapter 3 we have formalized two definitions of the arithmetical hierarchy in constructive type theory and have studied their equivalence by assuming axioms from synthetic computability.

The first definition was directly in first-order arithmetic and serves as a sanity check for the second synthetic definition. For that, we used the mechanization of the syntax and semantics of first-order arithmetic provided by the *Coq Library for Mechanised First-Order Logic* [30]. This kind of defining the arithmetical hierarchy can be found as a side note in the textbook by Odifreddi [42]. In contrast to the majority of other definitions in the literature (as we discuss in Section 6.2), it enables classifying formulas solely based on their syntax via counting the number of quantifier alternations in prenex normal form, so especially without incorporating computational properties.

To enable a rough upper-bound classification of arbitrary formulas, we have implemented and verified a prenex normal form algorithm. The algorithm is structurally recursive which allows a direct implementation in Coq. A consequential difficulty we were able to solve was to correctly rename variables when pulling multiple quantifiers at once. The prenex normal form is a classical result and thus we assumed the law of excluded middle for the verification of the algorithm. We generalized our results on prenex normal form to a general signature and model of first-order arithmetic, for which it is equivalent to the law of excluded middle. The algorithm does not optimize the number of quantifier alternations and thus only gives a rough upper-bound classification in the arithmetical hierarchy. We leave a further improvement for future work (cf. Section 6.3).

The second definition of the arithmetical hierarchy classifies type-theoretic predicates directly. It is purely synthetic without depending on a concrete model of computation. This makes it more convenient to work with and establish Post's theorem synthetically.

In Chapter 4 we have presented synthetic oracle machines and the first synthetic definition of the Turing jump.

The synthetic definition of Turing reducibility by Forster and Kirst [16] who have followed an idea by Bauer [4] has served as a starting point. However, how and under which axioms a synthetic Turing jump could be defined was unclear in the beginning since explicit constructions as they are presented in textbooks are in a concrete model of computation (e.g. oracle Turing machines or μ-recursive partial functionals) and cannot be translated to the synthetic setting.

Our key insight was that by adjusting the continuity requirement of synthetic oracle machines compared to Forster and Kirst, constructive results like that oracle machines can be solely described by a higher-order partial function were possible. This allows for constructing an enumeration of synthetic oracle machines by assuming an enumeration of continuous higher-order partial functions that can be used to define the synthetic Turing jump. We leave further investigation on the assumption and the question of whether an enumeration of continuous higher-order partial functions could eventually be constructed by assuming known axioms like EPF for future work (cf. Section 6.3).

As synthetic oracle machines can be solely described by a higher-order partial function, the question arises whether notions like Turing reducibility or oracle semi-decidability could be defined only by using higher-order partial functions. At the moment the characterization given by Theorem 4.17 is not very informative. Thus, we keep the definitions using oracle machines for the moment as we believe that they are more intuitive.

Ultimately, we have found a way to connect the synthetic arithmetic hierarchy to the synthetic Turing jump by proving Post's theorem synthetically. Meanwhile, the question has arisen if a step-wise interpretation of oracle machines would have been needed to establish Post's theorem (like in the proof by Soare [51]), but we were able to exploit continuity instead, following the proof idea of Odifreddi [42]. Most textbooks work in classical set theory and establish all their results classically. Therefore it was open how constructive a possible proof of Post's theorem would be. Our proof of Post's theorem does rely on the law of excluded middle but we were able to trace the classical reasoning in the proofs back to exactly two usages of LEM. We leave the further investigation on whether weakening the classical assumptions is possible for future work (cf. Section 6.3).

In Section 6.1, we briefly discuss our Coq development. Then, we discuss some related work in Section 6.2 and finally point out possible future work in Section 6.3.

6.1 Coq Mechanization

All results of this thesis are mechanized in the accompanying Coq development. In the digital version of this thesis, all definitions, lemmas, and theorems are hyperlinked to the respective position of the HTML that allows interactive navigation. The whole Coq development has been placed in the following Github repository that is self-contained and includes all dependencies.

`https://github.com/uds-psl/coq-posts-theorem/tree/ba-mueck`

All external dependencies are in the `external` folder and have been labeled with the exact source. The primary source of dependencies is the accompanying Coq development of Forster's PhD thesis [16] where we have taken e.g. the definition and proofs from our preliminaries Chapter 2 from. The definition of the syntax and semantics of first-order logic together with some corresponding facts was taken from the *Coq Library of Undecidability Proofs* [30].

Compared to the on-paper proofs, the Coq proofs are not more complex in general. Some of the proofs on the arithmetical hierarchy, especially Lemma 3.30 that the hierarchy is closed under bounded quantifiers and Lemma 3.31 that predicates of the hierarchy can be lifted to lists required some technical encodings that we have omitted on paper. For the bounded quantifier proof, we have assumed an encoding of lists to numbers that is known how to be constructed but not spelled out for this thesis. The proof of Post's theorem in Chapter 5 was presented on unary predicates on paper. In Coq, the respective encodings are made explicit, which adds another rewriting step and a trivial many-one reduction but does not increase the general complexity. In the future, it might be worth trying if an intermediate definition of the arithmetical hierarchy on unary predicates would simply the proofs similarily like on paper.

The Coq development is structured similarly to this thesis. The number of lines of code for the respective sections splits up as follows:

	Specification	**Proofs**
Prenex Normal Form	185	326
Arithemtical Hierarchy in First-order Logic	45	236
Arithemtical Hierarchy in Type Theory	103	459
Arithemtical Hierarchy Equivalence	15	105
Oracle Computability	170	649
Turing Jump	64	152
Post's Theorem	49	168
Total	631	2095

6.2 Related Work

Arithmetical Hierarchy Kleene [31] and Mostowski [38] developed the arithmetical hierarchy independently. Kleene was aware of Mostowski's work at least in 1952 [33] when he attributes the hierarchy to himself and Mostowski. Kleene defines the hierarchy with predicates decidable by μ-recursive functions in the base. Mostowski defines the hierarchy in first-order arithmetic with decidable formulas in the base according to the ordinary rules of inference and the ordinary arithmetical axioms.

Mostowski observes that quantifier-free formulas are at the base level of the hierarchy and that adding quantifiers may give an undecidable formula, but the definition of the arithmetical hierarchy in first-order arithmetical we present in Section 3.1 is not equivalent to Mostowski's hierarchy in the base. In the English Wikipedia entry on the arithmetical hierarchy [61], the hierarchy is defined with only quantifier-free formulas at the base and variations like the hierarchy with decidable predicates in the base are discussed together with the inequality of $\sum_0$ and $\prod_0$. We have checked several textbooks [49, 42, 9, 51, 14, 25, 24] but all define the arithmetical hierarchy with decidable predicates in the base. Only Odifreddi [42] mentions the alternative definition that only allows quantifier-free formulas and the inequality in the base.

Synthetic Computability Richman and Bridges [48, 6] study computability theory in the the context of constructive metamathematics in the sense of Bishop [5]. Richman [48] lays the foundation of synthetic computability by proposing the axiom of the enumerability of all enumerable sets and proving the undecidability of the halting problem solely synthetically without a concrete model of computation.

Bauer [3] develops synthetic computability in Hyland's effective topos [27] by assuming the enumerability of enumerable sets of numbers. He proves many fundamental results including another theorem that is well known under the name "Post's theorem" and should not be confused with what we consider "Post's Theorem" in this thesis. What he considers "Post's Theorem" is that a set is decidable if and only if it is semi-decidable and its complement also is semi-decidable.

Both Richman, Bridges and Bauer assume the axiom of countable choice which makes the law of excluded middle disprovable since both axioms together imply that all predicates are decidable [55].

Synthetic Computability in Coq Forster, Kirst and Smolka [18] have developed a basic framework for synthetic computability theory in Coq by translating some of the ideas of Bauer [3] to constructive type theory. They also prove the same "Post's Theorem" as Bauer – that predicates are decidable if and only if they are enumerable and their complement is enumerable – and mechanize that it is equivalent to Markov's principle [55].

Their framework for synthetic computability in Coq was developed further as part of the *Coq Library of Undecidability Proofs* [19] with many applications on mechanized reductions of undecidable problems.

Forster [15, 17] studies multiple different synthetic axioms in constructive type theory and elaborates on their consistency together with the law of excluded middle.

Synthetic One-One, Many-One, and Truth-Table Reducibility In his Bachelor's thesis [28] and in a resulting preprint together with Forster and Smolka [20], Jahn studies synthetic one-one, many-one, and truth-table reducibility in more detail. They prove a constructive and synthetic version of Myhill's isomorphism theorem [40] that one-one equivalent predicates are isomorphic and solve Post's problem synthetically for many-one and truth-table reducibility by synthetically establishing simple and hyper-simple predicates [46].

Synthetic Turing Reductions The first synthetic definition of Turing reductions was proposed by Bauer [4]. Bauer does not work in constructive type theory but in the intuitionistic effective topos. He first came up with the idea of defining synthetic Turing reductions in two layers as a transformer of solvers of potentially uncomputable problems that factors through a computable core on computable oracles.

The first synthetic definition of Turing reductions in constructive type theory is presented by Forster in his PhD thesis [16] and was conceived in joint work with Kirst following the two-layered idea by Bauer. Furthermore, Forster discusses two well-known refinements of Turing reductions, bounded Turing reductions and total bounded Turing reductions and shows that the latter is equivalent to truth-table reductions. Finally, he shows the well-known fact that truth-table reductions however differ from general Turing reductions and compares their definition to Bauer's.

Our definition of synthetic oracle machines was mostly adopted from the definition of Forster and Kirst, but with an adjustment in the formulation of the continuity requirement that has enabled us to determine synthetic oracle machines solely by a higher-order function.

We discuss the definitions by Bauer and by Forster and Kirst in more detail in Subsection 4.1.5 and carefully work out the differences and what they imply.

6.3 Future Work

The prenex normal form algorithm we verified does not optimize the number of quantifier alternations and therefore is of limited suitability for the classification of formulas in the arithmetical hierarchy. In future work, one could improve the algorithm concerning this matter. A possible approach could be to optimize when merging the two sub-lists in the recursive step of a binary operator instead of simply concatenating the lists. This would require constructing the de Bruijn renamings simultaneously when merging the lists. Another interesting question would be if one could prove the equivalence of such an algorithm to the non-deterministic algorithm that arises out of the rules of Lemma 3.5 and is presented in textbooks [49] (there are only finitely many prenex normal forms for each formula when applying these rules).

We assumed ACT – a CT-like axiom – for our equivalence result of the two arithmetical hierarchies in Section 3.3. In future work this axiom could probably be reduced to a more common variant of CT by using the results of Peters [45] built on work by Larchey-Wendling and Forster [37] and Kirst and Hermes [29, 22] as discussed in the remark at the end of Subsection 3.3.2.

For defining the Turing jump, we have assumed an enumeration for continuous higher-order partial functions. It would be interesting to study this enumerator more carefully. Maybe one could be able to construct it (or a weaker formulation of it) by only using known axioms like EPF. A possible starting point could be the lecture notes by Streicher [52] on Kleene's second algebra where he elaborates on the realizability of continuous functions in the Baire space.

On the other side, one could consider if defining a synthetic Turing jump might be possible without enumerating higher-order functions. One could perhaps change the definition of oracle machines such that the core is a function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{1}$ (which is isomorphic to functions of type $\mathbb{N} \to \mathbb{1}$ that are enumerated by EPF) where the oracle is passed as the first component of the argument such that the argument $i : \mathbb{N}$ means that the oracle is the i-th function enumerated by EPF. It would be interesting to study whether such a definition of oracle machines could give an equivalent definition of Turing reductions or whether this definition might be weaker but sufficient to construct a synthetic Turing jump.

Our synthetic proof of Post's theorem is classical. In future work, one could study whether some of the statements can be proven constructively or study the equivalence to classical axioms.

There is one result essential in textbooks on oracle semi-decidability and Turing reductions that has only lately attracted our attention since it is not necessary for establishing Post's theorem and that we were not able to prove synthetically yet: $\mathcal{S}_Q(P) \land \mathcal{S}_Q(\overline{P}) \leftrightarrow P \preceq_T Q$. Note that the not relativized version is also known as "Post's theorem" and is equivalent to Markov's principle [55], so the relativized version for sure only holds when assuming classical axioms. The difficult direction is to construct a Turing reduction given the two semi-deciders. The obvious core of such a Turing reduction would be to step-wise simulate the partial functions gained when forwarding the oracle to the two semi-deciders and check which semi-decider terminates first. The problem however is, that such a core is not continuous. Continuity requires that if an oracle machine halts it still halts with the same output when replacing the oracle with an oracle that agrees on the finite list in question. For the core above not only termination matters but also the number of steps after which the step-wise evaluation of the cores of the two semi-deciders terminates. When replacing the oracle, another semi-decider could terminate first which would change the answer of the constructed Turing reduction and violates continuity.

As there exists a framework for synthetic relative decidability now, it would be interesting to study more results on relative decidability synthetically. For example, one could study the Kleene-Post theorem [34] that there are uncomparable Turing degrees or solve Post's problem for Turing reductions synthetically for instance by finding a synthetic proof for the Friedberg-Muchnik theorem [21, 39]. This was already formulated as future work by Bauer [4] and Forster [16], but to the best of our knowledge, nobody has worked on it yet.

Bibliography

[1] Yohji Akama, Stefano Berardi, Susumu Hayashi, and Ulrich Kohlenbach. An arithmetical hierarchy of the law of excluded middle and related principles. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 192–201, 2004. doi:10.1109/LICS.2004.1319613.

[2] Hendrik P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

[3] Andrej Bauer. First steps in synthetic computability theory. In Martín Hötzel Escardó, Achim Jung, and Michael W. Mislove, editors, *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2005, Birmingham, UK, May 18-21, 2005*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 5–31. Elsevier, 2005. doi:10.1016/j.entcs.2005.11.049.

[4] Andrej Bauer. Synthetic mathematics with an excursion into computability theory. University of Wisconsin Logic seminar, 2021. URL `http://math.andrej.com/asset/data/madison-synthetic-computability-talk.pdf`.

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

[6] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1987. doi:10.1017/CBO9780511565663.

[7] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. ISSN 00029327, 10806377. doi:10.2307/2371045.

[8] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936. doi:10.2307/2269326.

[9] S. Barry Cooper. *Computability theory*. Chapman & Hall/CRC Press, 2004. ISBN 1584882379.

[10] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988. ISSN 0890-5401. doi:10.1016/0890-5401(88)90005-3.

[11] Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934. doi:10.1073/pnas.20.11.584.

[12] Martin Davis. *Computability and Unsolvability*. McGraw-Hill, 1958.

[13] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. ISSN 1385-7258. doi:10.1016/1385-7258(72)90034-0.

[14] Rodney G. Downey and Denis R. Hirschfeldt. *Algorithmic randomness and complexity*. Springer Science & Business Media, 2010. ISBN 978-0-387-68441-3. doi:10.1007/978-0-387-68441-3.

[15] Yannick Forster. Church's thesis and related axioms in Coq's type theory. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*, volume 183 of *LIPIcs*, pages 21:1–21:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CSL.2021.21.

[16] Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021. doi:10.22028/D291-35758.

[17] Yannick Forster. Parametric Church's thesis: Synthetic computability without choice. In Sergei N. Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science - International Symposium, LFCS 2022, Deerfield Beach, FL, USA, January 10-13, 2022, Proceedings*, volume 13137 of *Lecture Notes in Computer Science*, pages 70–89. Springer, 2022. doi:10.1007/978-3-030-93100-1_6. URL `https://doi.org/10.1007/978-3-030-93100-1_6`.

[18] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 38–51. ACM, 2019. doi:10.1145/3293880.3294091.

[19] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In

CoqPL 2020 The Sixth International Workshop on Coq for Programming Languages, 2020.

[20] Yannick Forster, Felix Jahn, and Gert Smolka. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq (Full Version). working paper or preprint, February 2022. URL `https://hal.inria.fr/hal-03580081`.

[21] Richard M. Friedberg. Two recursively enumerable sets of incomparable degrees of unsolvability (solution of Post's problem, 1944). *Proceedings of the National Academy of Sciences*, 43(2):236–238, 1957. doi:10.1073/pnas.43.2.236.

[22] Marc Hermes and Dominik Kirst. An analysis of Tennenbaum's theorem in constructive type theory. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, 2022.

[23] David Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik*. Springer, Berlin, Germany, 1972. ISBN 9783662119327.

[24] Peter G. Hinman. *Recursion-Theoretic Hierarchies*. Perspectives in Logic. Cambridge University Press, 2017. doi:10.1017/9781316717110.

[25] Steven Homer and Alan L. Selman. *Computability and complexity theory*, volume 194. Springer, 2011. ISBN 978-1-4614-0682-2. doi:10.1007/978-1-4614-0682-2.

[26] William A. Howard. The formulae-as-types notion of construction. In Haskell B. Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[27] J. Martin E. Hyland. The effective topos. In A.S. Troelstra and D. van Dalen, editors, *The L. E. J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 165–216. Elsevier, 1982. doi:10.1016/S0049-237X(09)70129-6.

[28] Felix Jahn. *Synthetic One-One, Many-One, and Truth-Table Reducibility in Coq*. Bachelor's thesis, Saarland University, September 2020. URL `https://ps.uni-saarland.de/~jahn/files/thesis.pdf`.

[29] Dominik Kirst and Marc Hermes. Synthetic undecidability and incompleteness of first-order axiom systems in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 23:1–23:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.23.

[30] Dominik Kirst, Johannes Hostert, Andrej Dudenhefner, Yannick Forster, Marc Hermes, Mark Koch, Dominique Larchey-Wendling, Niklas Mück, Benjamin Peters, Gert Smolka, and Dominik Wehr. A Coq library for mechanised first-order logic. In *The Coq Workshop*, 2022.

[31] Stephen C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943. doi:10.1090/S0002-9947-1943-0007371-8.

[32] Stephen. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10(4):109–124, 1945. doi:10.2307/2269016.

[33] Stephen C. Kleene. Introduction to metamathematics. 1952.

[34] Stephen C. Kleene and Emil L. Post. The upper semi-lattice of degrees of recursive unsolvability. *Annals of mathematics*, pages 379–407, 1954. doi:10.2307/1969708.

[35] Georg Kreisel. Mathematical logic. *Lectures in modern mathematics 3*, pages 95–195, 1965.

[36] Kazimierz Kuratowski and Alfred Tarski. *Les opérations logiques et les ensembles projectifs*. Uniwersytet, Seminarjum Matematyczne, 1931. doi:10.4064/FM-17-1-240-248.

[37] Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in coq (extended version). *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/lmcs-18(1:35)2022.

[38] Andrzej Mostowski. On definable sets of positive integers. *Fundamenta Mathematicae*, 34(1):81–112, 1947. doi:10.4064/fm-34-1-81-112.

[39] Albert A. Muchnik. On strong and weak reducibility of algorithmic problems. *Sibirskii Matematicheskii Zhurnal*, 4(6):1328–1341, 1963.

[40] John Myhill. Creative sets. *Mathematical Logic Quarterly*, 1(2):97–108, 1955. doi:10.1002/malq.19550010205.

[41] A. Nerode. General topology and partial recursive functionals. *Talks Cornell Summ. Inst. Symb. Log.*, pages 247–251, 1957.

[42] Piergiorgio Odifreddi. *Classical recursion theory. The theory of functions and sets of natural numbers.*, volume 125 of *Stud. Logic Found. Math.* Amsterdam etc.: North-Holland, paperback ed. edition, 1992. ISBN 0-444-89483-7.

[43] Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and*

Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. doi:10.1007/BFb0037116.

[44] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic* (*Mathematical logic and foundations*). College Publications, January 2015. URL `https://hal.inria.fr/hal-01094195`.

[45] Benjamin Petres. *Gödel's Theorem Without Tears - Essential Incompleteness in Synthetic Computability*. Bachelor's thesis, Saarland University, June 2022. URL `https://www.ps.uni-saarland.de/~peters/bachelor/resources/thesis.pdf`.

[46] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50(5):284–316, 1944. doi:10.1090/s0002-9904-1944-08111-1.

[47] Emil L. Post. Degrees of recursive unsolvability-preliminary report. In *Bulletin of the American Mathematical Society*, volume 54, pages 641–642, 1948.

[48] Fred Richman. Church's thesis without tears. *J. Symb. Log.*, 48(3):797–803, 1983. doi:10.2307/2273473.

[49] Hartley Rogers. *Theory of recursive functions and effective computability*. MIT Pr., 2. print. edition, 1988. ISBN 0262680521.

[50] Thoralf Skolem. Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Bewiesbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen. *Skrifter utgit av Videnskapsselskapet i Kristiania. I, Matematisk-naturvidenskabelig klasse*, pages 1–36, 1920.

[51] Robert I. Soare. *Turing Computability : Theory and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 9783642319334.

[52] Thomas Streicher. Realizability. Lecture Notes, WS 17/18. URL `https://www2.mathematik.tu-darmstadt.de/~streicher/REAL/REAL.pdf`.

[53] Andrew W. Swan and Taichi Uemura. On church's thesis in cubical assemblies. *Mathematical Structures in Computer Science*, pages 1–20, 2022. doi:10.1017/S0960129522000068.

[54] The Coq Development Team. The Coq proof assistant. January 2022. doi:10.5281/zenodo.5846982. Version 8.15.

[55] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics, Vol 1: Volume 121*. Studies in logic and the foundations of mathematics. Elsevier Science, London, England, January 1988. ISBN 0444702660.

[56] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58:345–363, 1936.

[57] Alan M. Turing. *Systems of Logic Based on Ordinals*. PhD thesis, Princeton University, NJ, USA, 1938. doi:10.1112/plms/s2-45.1.161.

[58] Alan M. Turing. Systems of logic based on ordinals. *Proceedings of the London mathematical society*, 2(1):161–228, 1939.

[59] Vladimir A. Uspenskii. On enumerable operators. *Dokl. Acad. Nauk.*, 103: 773–776, 1955.

[60] Benjamin Werner. Sets in types, types in sets. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, pages 530–546, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69530-1. doi:10.1007/BFb0014566.

[61] Wikipedia contributors. Arithmetical hierarchy — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Arithmetical_hierarchy&oldid=1085052268`, 2022. [Online; accessed 8-June-2022].

[62] Norihiro Yamada. Game semantics of Martin-Löf type theory, part iii: its consistency with Church's thesis. 2020. doi:10.48550/ARXIV.2007.08094.