The Undecidability of BB(748)

Understanding Gödel’s Incompleteness Theorems

Johannes Riebel

Bachelor Thesis at the Chair of Algebra and Number Theory

March 2023

Universität
Augsburg
University

Introduction

Sentence A: "*If Sentence A is true, then the Eiffel Tower is in Munich.*"

Let us assume that Sentence A is true. Then, by definition, the Eiffel Tower is located in Munich. Discharging the assumption, we can state that "*If Sentence A is true, then the Eiffel Tower is in Munich.*" is true. This, however, is exactly Sentence A. Using the definition of Sentence A, we can conclude that the Eiffel Tower is in Munich.

This argument is known as Curry's paradox. Where did this apparent proof go wrong? One way to resolve the paradox is to accept only formal systems that do not allow the construction of Sentence A.

But how should we determine the appropriate formal system? Many mathematicians tackled this question in the early 20th century. Ideally, they aimed to discover a system capable of proving or disproving any given proposition and demonstrating its consistency through finite means. However, Gödel shattered these aspirations in 1931 by proving the existence of true but unprovable mathematical formulas. Moreover, he demonstrated that any attempt to formalize mathematics that satisfies minimal requirements will be subject to his incompleteness theorem.

To fully appreciate Gödel's remarkable proof, we must first immerse ourselves in the world of formal theories and their construction. We will explore the building blocks of formal theories, including axioms, rules of inference, and the logical language that forms the foundation of these systems. This understanding will later enable us to prove metamathematical properties of systems, such as the aforementioned provability restrictions of Gödel's incompleteness theorem. Furthermore, as we navigate the realm of formal languages, we will differentiate between first-order theories and second- and higher-order theories, which differ in their ability to quantify over objects, properties of objects, and so on.

To help the reader develop an intuition for constructing proofs in a fully formal way, we will provide detailed examples of formal proofs in the first chapter. We will show that seemingly simple properties, such as the commutativity of addition of natural numbers, $k + n = n + k$, require extensive and elaborate proofs.

Similar to how we distinguish between group axioms and examples of groups like $\mathbb{Z}$ and $\mathbb{Q}$, we will distinguish between axiomatic theories and their models – that is, worlds that satisfy these axioms. Since our language is defined in a formal system, but the notion of truth exists only in models, we can only speak of *true* mathematical propositions in a formal theory if they are true in all realizations. A significant milestone in model theory was the completeness theorem, which states that in first-order logic and given a sufficiently expressive logical axiom system, this notion of truth is equivalent to the notion of provability.

This seemingly contradicts Gödel's first incompleteness theorem which states that there are unprovable and unrefutable sentences. However, in this context, Gödel's incompleteness theorem only asserts that there are mathematical statements that are true in some models and false in others. This shows that we cannot axiomatize, for instance, the natural numbers up to isomorphism in first-order logic. We will first prove this result in a different way, using the Löwenheim–Skolem theorem: If a first-order theory has an infinite model, it has arbitrarily large models.

In the second part of our journey, we will examine Gödel's original proof of the first and second incompleteness theorems. Gödel's revolutionary idea was to represent mathematical symbols, and consequently strings of these symbols as well as strings of strings of such symbols by natural numbers. By doing so, he effectively transformed the discussion of metamathematical relations like "$\phi_1, \ldots, \phi_r$ is a proof of ψ" to the discussion of natural numbers: "The natural number n corresponds to a proof of the formula corresponding to the number k". Furthermore, he showed that – given the set of axioms is accessible enough – this relation of natural numbers (n corresponds to a proof of a formula that corresponds to k) has an easy-to-understand structure. In particular, he showed that we can construct an arithmetical formula $\varphi(x, y)$ in our formal theory that captures this relation of natural numbers: If n and k are natural numbers such that the above relation holds, our formal theory proves $\varphi(\bar{n}, \bar{k})$. Conversely, if the relation does not hold, our theory proves $\neg\varphi(\bar{n}, \bar{k})$. Using this technique, Gödel was able to provide a self-referencing sentence χ that semantically talks about its own provability. It was then easy for Gödel to show that the provability of χ as well as of $\neg\chi$ each entail that a contradiction is provable.

Gödel initially showed this result for so-called ω-consistent theories, which are slightly stronger than merely consistent theories. Five years later, J. Barkley Rosser improved this result to encompass consistent theories, using what is now known as Rosser's trick. His proof resembles the Liar's Paradox:

"*This sentence is false.*"

If this sentence is true, this sentence is false. Conversely, if it is false, it is true. A similar version of this sentence, talking about provability instead of truth, can be stated in many formal theories, and is precisely Rosser's sentence. However, the process of constructing such a self-reference is non-trivial, and it is the critical part of this proof.

In addition to his first incompleteness theorem, Gödel provided a second startling result: If a formal theory can prove that it produces no contradictions (is consistent), then it must produce contradictions (is inconsistent), and could only prove that it produces no contradictions because it proves everything. Consequently, a consistent theory does not prove that it is consistent. We will demonstrate this statement using Löb's theorem, which is yet another fascinating result. It states that if a theory cannot prove a formula, it cannot prove that a proof of that formula would imply the formula. In some sense, it shows that formal theories do not trust their own judgment.

In the final part of our exploration, we will present an accessible example of the first incompleteness theorem. Given a sufficiently expressive programming language, we can write a program that enumerates all the proofs of a theory and stops if and only if it finds a proof of $0 = 1$. If we incorporate the description of this program into our formal theory, and assume that the theory is consistent, then the theory cannot prove that the program will never stop. If it could, we would have a proof of the consistency of the theory, which contradicts Gödel's second incompleteness theorem.

In this paper, we adopt a possible formal framework for vast areas of modern mathematics, **ZFC**, and choose Turing machines as our programming language. We then discuss a 748-state Turing machine that enumerates all proofs and halts if and only if it finds a contradiction. The above argument can be translated informally as: "Assuming that modern mathematics does not produce contradictions, then modern mathematics is incapable of proving that this particular Turing machine will never stop." This striking result defies intuition, since one might assume that, given enough time and space, it should be possible to determine whether a program will eventually stop or continue looping.

We can reformulate this statement. Given all Turing machines with n states (finitely many), there will be some that will loop forever and some that will eventually stop. If we focus only on those that will halt, there exists a machine with n states that will run the longest. Let us define $\mathbf{BB}(n)$ to be the number of steps this machine runs before it halts. We will show that the function $\mathbf{BB}(n)$ is well defined and rapidly increasing. If we knew the value of $\mathbf{BB}(748)$, we could simulate the above Turing machine for $\mathbf{BB}(748)$ steps, prove that it has not yet halted – assuming that **ZFC** is indeed consistent – and that it will never halt (because that would contradict the definition of $\mathbf{BB}(748)$). This is impossible and shows that **ZFC** cannot settle the value of $\mathbf{BB}(748)$. It is the result obtained by Stefan O'Rear in 2017, [ORe16, zf2.nql], which surpasses his previous bound of 1919 from 2016 [ORe16, zf.nql]. O'Rear's work significantly improved upon the initial 2016 research by Adam Yedidia and Scott Aaronson, which established an upper bound of 7910,[YA16]. Their work was the first of its kind and initiated this line of inquiry.

The main goal of this paper, presented at the very end, is to further refine this optimized result. We discuss the construction of O'Rear's machine and detail our efforts to increase the number of states. Through these improvements, we successfully lowered the upper bound of undecidability from $n = 748$ to $n = 745$.

Contents

1 Introduction to Logic **4**
1.1 Metalogic vs. Logic . . . 4
1.2 First-Order Languages . . . 4
1.3 Axiomatic Systems and Their Models . . . 6
1.4 Logical Axioms and Rules of Inference . . . 11
1.5 Examples of Formal Proofs . . . 13
1.6 Gödel's Completeness Theorem . . . 20
1.7 Consequences of Gödel's Completeness Theorem . . . 22
1.8 Exploring Second-Order and Higher-Order Logic . . . 23
1.9 Type Theory . . . 25

2 Gödel's Incompleteness Theorems **27**
2.1 Gödel's System P . . . 27
2.2 Primitive Recursive Functions . . . 30
2.3 The First Incompleteness Theorem for Extensions of P . . . 38
2.4 Rosser's Trick . . . 45
2.5 The Second Incompleteness Theorem . . . 49

3 BB(745) and Undecidability in ZFC – Exploring the Limits of Set Theory **53**
3.1 Computability and Turing Machines . . . 53
3.2 The Busy Beaver Function . . . 56
3.3 ZF and ZFC . . . 58
3.4 The Original Paper: BB(7910) and BB(748) . . . 59
3.4.1 Basic Features of the Code . . . 60
3.4.2 Encoding of Formulas and the WFF-Stack . . . 60
3.4.3 Proofs in the Code . . . 62
3.5 Failed Attempts to Optimize the Result . . . 65
3.5.1 Optimizing the Pairing Function . . . 66
3.5.2 Using a Different Pairing Function . . . 67
3.5.3 Choosing a Different Undecidable Formula . . . 68
3.5.4 Decompressing and Simulating the Existing Turing Machine . . . 69
3.5.5 Peephole Optimization . . . 71
3.6 Optimizing the Result: BB(745) . . . 71
3.6.1 Changing the Arguments in the Pairing Function . . . 73
3.6.2 Optimizing the Pairing Function in the Compiler . . . 73
3.6.3 Changing the Formula Definitions . . . 76

1 Introduction to Logic

The following is a brief introduction to the basic definitions and concepts that will be relevant in the coming sections. We follow the ideas of [Bar77, Chapter A1, §2–§4] and adopt much of Barwise's notation. A good introduction to first-order logic with similar definitions is also given in [Pro21] and [Pro22].

1.1 Metalogic vs. Logic

In order to determine the provability of statements within a formal theory, it is essential to examine the theory from an external perspective. This broader context, known as "metalogic," provides the logic and reasoning used in proofs, and it is crucial to distinguish it from the internal logic or theory being studied.

In this context, we will use the high-level language of English and standard logic (e.g., **ZFC**) as our metalogic. It is worth noting that the main results of this thesis can also be obtained assuming a weaker metalogic.

1.2 First-Order Languages

To talk about terms, formulas, and sentences, we need to introduce the notion of a language. Similar to natural languages like English or German, we will introduce a logical language as a set of symbols and later introduce rules how to form expressions and formulas with them (syntax).

Definition 1.2.1. (First-order language with equality) A first-order language $\mathcal{L}$ is a set of symbols:

- logical symbols: $\{\neg, \vee, \forall, =\}$,
- constant symbols and variables: $\mathcal{C} \subseteq \{c_i \mid i \in \mathbb{N}\}$, $\mathcal{V} = \{x_i \mid i \in \mathbb{N}\}$ respectively,
- function symbols: $\mathcal{F} \subseteq \{f_i \mid i \in \mathbb{N}\}$,
- relation symbols $\mathcal{R} \subseteq \{r_i \mid i \in \mathbb{N}\}$,
- and brackets: {"(", ")"}

together with a function $\# : \mathcal{F} \cup \mathcal{R} \to \mathbb{N}_{>0}$. We call $\#(f)$ and $\#(r)$ the arity of f and r, respectively, and define it as the number of elements to be given as input.

For better readability, we shall permit symbols not of the form x_i, f_i, r_i for variables, function symbols, and relation symbols, respectively (for example, we shall permit $y, z, \dots$ to denote variables). The reader may regard this as metamathematical notation, or may extend our definition to include these symbols. We will also use ξ, ζ, and ϑ to denote meta-variables. These variables are not part of the language, but can be understood as placeholders for any variable of the language.

Note that the sets $\mathcal{C}, \mathcal{F}$ and $\mathcal{R}$ are subsets and may be finite or empty. To make this definition more compact, one could allow $\#(f) = 0$ and interpret such functions as constants. The fact that "=" is a logical and not a relational symbol is not a mistake. As we will see later, equality can not be formulated as a relation in first-order logic.

Example 1.2.2. (Peano arithmetic) An important example is the language we will use for Peano arithmetic (**PA**). In this system we use the language $\mathcal{L}_{\text{PA}}$ given by: $\mathcal{C} = \{0\}, \mathcal{F} = \{\text{S} = f_0, + = f_1, \times = f_2\}$, and $\mathcal{R} = \{<= r_1\}$, where $\#(\text{S}) = 1$ and $\#(+), \#(\times), \#(<) = 2$. We will discuss **PA** in more detail in the next chapter.

Definition 1.2.3. (Terms) We define the notion of terms inductively:

- All variables x_i and constants c_i are terms
- If f is a function symbol with $\#(f) = n$ and $t_1, ..., t_n$ are terms then $f(t_1, ..., t_n)$ is a term

Definition 1.2.4. (Formulas) We define formulas inductively:

- For a given n-ary relation symbol r and terms $t_1, ..., t_n$ we define
$$(t_1 = t_2), \qquad (r(t_1, ..., t_n))$$
as (atomic) formulas.
- If φ, ψ are formulas then
$$(\varphi \vee \psi), \quad (\neg\varphi), \quad (\forall\xi\ \varphi)$$
are formulas of the language $\mathcal{L}$.

Note that φ and ψ are not language symbols. Similar to how ξ, ζ and ϑ describe variables, we use symbols like φ, ψ and χ in our metamathematical notation to refer to formulas.

Example 1.2.5. The following formulas of the language $\mathcal{L}_{\mathrm{PA}}$ are syntactically correct:

$$\begin{aligned} &\varphi := (\neg(0 = 0))\,, \\ &\chi := (\forall x\,(\neg(\forall y\,(\neg(<(x, y))))))\,, \\ &\psi := (\neg(\forall x\,(\neg(\forall y\,((\neg(<(x, y))) \vee (\neg(\forall\,(\neg(\neg(\forall z_2\,(\\ &\qquad\quad \neg(\neg((\neg(\neg((\neg(\times(z_2, y))) \vee (\neg(<(z_1, y)))))) \vee (\neg(<(z_2, y)))))))))))))))))\,, \end{aligned} \tag{1}$$

whereas, strictly speaking, those are not:

$$\begin{aligned} &\varphi := \neg(0 = 0)\,, \\ &\chi := \forall x\, \exists y\, (x < y)\,, \\ &\psi := \exists x\, \forall y\, (x < y \to \exists z_1\, \exists z_2\, (z_1 \cdot z_2 = y \wedge z_1 < y \wedge z_2 < y))\,. \end{aligned} \tag{2}$$

This is because they use undefined symbols or incorrect bracketing.

Notation 1. *We introduce a notation in the metalanguage for more readability:*

- *We define the (familiar) abbreviations:*
  - $(\exists\xi\ \varphi) := (\neg(\forall\xi\ (\neg\varphi)))$,
  - $(\varphi \wedge \psi) = (\neg((\neg\varphi) \vee (\neg\psi))$,
  - $(\varphi \to \psi) := ((\neg\varphi) \vee \psi)$,
  - $(\varphi \leftrightarrow \psi) := ((\varphi \to \psi) \wedge (\psi \to \varphi))$.
- *In addition, we set* $\xi\ f\ \zeta := f(\xi, \zeta)$ *or* $\xi\ r\ \zeta := r(\xi, \zeta)$ *when we see fit for some* 2*-ary function symbols or relation symbols of the language (e.g.* $f = +,\ r = <$ *in* $\mathcal{L}_{\mathrm{PA}}$*).*
- *As long as the parsing order is clear, we may omit brackets. We also allow brackets of different sizes, as well as square brackets.*

With this notation the formulas φ, χ, ψ of Equation (1) turn into the formulas of Equation (2).

Definition 1.2.6. (Free variables) Let φ be a formula in a language $\mathcal{L}$. The set $\mathbf{FV}(\varphi)$ is defined recursively:

- If φ is an atomic formula, then $\mathbf{FV}(\varphi)$ is defined as the set of variables that occur in φ,
- $\mathbf{FV}(\neg\varphi) := \mathbf{FV}(\varphi)$ and $\mathbf{FV}(\varphi \vee \psi) := \mathbf{FV}(\varphi) \cup \mathbf{FV}(\psi)$,

- $\mathbf{FV}(\forall\xi\ \varphi) := \mathbf{FV}(\varphi)\backslash\{\xi\}$.

It describes the set of free variables (i.e. variables that are not bound by quantifiers) in φ. If $\mathbf{FV}(\varphi) = \emptyset$, we call φ a closed formula or sentence in $\mathcal{L}$.

1.3 Axiomatic Systems and Their Models

The goal of axiomatic systems is to generalize an example by extracting the key features of interest. Models are structures that satisfy these axioms. To be more precise:

Definition 1.3.1. (Axiomatic system) An axiomatic system $\mathcal{A}$ in a language $\mathcal{L}$ is a (not necessarily finite) set of sentences in the language $\mathcal{L}$.

Definition 1.3.2. (Formal theory) The combination of a first-order predicate language $\mathcal{L}$ together with an axiomatic system $\mathcal{A}$ is called a *formal theory* of first-order. We will denote such a theory by the letter $\mathcal{T}$.

Definition 1.3.3. (Structures for a language $\mathcal{L}$) A (set-theoretic) structure $\mathcal{S} = \langle M, I\rangle$ for a given language $\mathcal{L}$ is a pair of a nonempty set M and a function I with domain $\mathcal{L}$ such that:

- $I(\mathcal{C}) \subseteq M$,
- $I(r) \subseteq M^{\#(r)}$ for every relation symbol $r \in \mathcal{R}$,
- $I(f)$ is a function with $I(f)\colon M^{\#(f)} \to M$.

So far, terms, formulas, and sentences are just meaningless finite strings of symbols. Now we introduce semantics to our language. To do so, we introduce a way to insert values into variables:

Definition 1.3.4. (Variable assignment) Let $\mathcal{L}$ be a first-order language and $s : \mathcal{V} \to M$ be a function. We call s an assignment for $\mathcal{S}$ and define the notion $t\,\big|_{\mathcal{V}\leftarrow s}$ for terms t recursively:

- If t is a constant, then $t\,\big|_{\mathcal{V}\leftarrow s} := I(t)$.
- If t is a variable, then $t\,\big|_{\mathcal{V}\leftarrow s} := s(t)$.
- $f(t_1, ..., t_n)\,\big|_{\mathcal{V}\leftarrow s} := I(f)(t_1\,\big|_{\mathcal{V}\leftarrow s}, \dots, t_n\,\big|_{\mathcal{V}\leftarrow s})$, for every function symbol $f \in \mathcal{F}$ and terms $t_1, ..., t_n$.

Definition 1.3.5. (Semantics of a language $\mathcal{L}$) Let $\mathcal{L}$ be a language, $\mathcal{S}$ a corresponding structure and s an assignment for $\mathcal{S}$. We define recursively for terms $t_1, t_2, \dots, t_n$ and formulas φ, ψ:

- $\mathcal{S} \models (t_1 = t_2)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if $t_1\,\big|_{\mathcal{V}\leftarrow s} = t_2\,\big|_{\mathcal{V}\leftarrow s}$,
- $\mathcal{S} \models r(t_1, ..., t_n)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if $\left(t_1\,\big|_{\mathcal{V}\leftarrow s}, ..., t_n\,\big|_{\mathcal{V}\leftarrow s}\right) \in I(r)$,
- $\mathcal{S} \models (\varphi \vee \psi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if $\mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s}$ or $\mathcal{S} \models \psi\,\big|_{\mathcal{V}\leftarrow s}$,
- $\mathcal{S} \models (\neg\varphi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if not $\mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s}$,
- $\mathcal{S} \models (\forall\xi\ \varphi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if for all $a \in M$, $\mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s[\xi/a]}$.

By $s[\xi/a]$ we denote the assignment for $\mathcal{S}$ that results from s by setting $\xi \mapsto a \in M$ and extending this change to the recursive definition for terms in Definition 1.3.4 (i.e., in each term ξ is sent to a).

Some remarks on Definition 1.3.5:

- If φ is a sentence, i.e. a closed formula, we simply write $\mathcal{S} \models \varphi$, because the specific choice of s does not affect the interpretation. Because of our recursive definition of formulas and because we adopt a classical metatheory, this uniquely determines whether for a sentence $\mathcal{S} \models \varphi$ or $\mathcal{S} \not\models \varphi$ holds. If $\mathcal{S} \models \varphi$ holds, we say that $\mathcal{S}$ satisfies φ.

- Note that the above definition mixes expressions of $\mathcal{L}$ and the meta language. For instance in the first bullet point the "=" on the left side is part of a formula in $\mathcal{L}$ whereas the right "=" is part of the meta language. Similarly for the other bullet points.

- Also note that because of the fourth bullet point, for an arbitrary formula φ it is never the case that $\mathcal{S} \not\models \varphi$ and $\mathcal{S} \not\models \neg\varphi$. The same generally cannot be said about its counterpart in formal theories. This is the result of Gödels first incompleteness theorem and we will explore its proof in detail in Chapter 2.

- Furthermore, note that in Definition 1.2.4 we allow the same variable symbol to be bound by different quantifiers. This can be understood semantically using the shadowing principle. The reader may be familiar with a similar concept used in many programming languages that distinguish between local and global variables. Although they may be defined with the same name, they are accessed differently depending on the context and within a given scope. Similarly, in formal languages, only the innermost binding is valid in a semantic interpretation. This concept is captured by Definition 1.3.5. For instance, the formula

$$\exists x\, [(\forall x\, r_1(x)) \to r_2(x)]$$

  is syntactically correct (after replacing abbreviations and correcting brackets). It has the following semantic interpretation: There exists one particular $a \in M$ such that $b \notin I(r_1)$ for at least one $b \in M$ or $a \in I(r_2)$. For the sake of clarity, we will avoid such notation whenever possible. Nevertheless, it is useful to reduce restrictions for some definitions, and we need it in Chapter 3 to define the set of axioms more efficiently.

Definition 1.3.6. (Models of an axiomatic system) Let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ an axiomatic system in $\mathcal{L}$. A structure $\mathcal{M}$ for $\mathcal{L}$ such that $\mathcal{M} \models \varphi$ for each axiom $\varphi \in \mathcal{A}$ is called a model of $\mathcal{A}$ and is denoted as $\mathcal{M} \models \mathcal{A}$.

Definition 1.3.7. (Semantic consequence) Let $\mathcal{L}$ be a first-order language, $\mathcal{A}$ an axiomatic system and φ a formula in $\mathcal{L}$. We define:

$$\begin{aligned} \mathcal{A} \models \varphi \;&:\Longleftrightarrow\; \{\mathcal{S} \text{ structure for } \mathcal{L} \mid \forall \psi \in \mathcal{A} : \mathcal{S} \models \psi\} \subseteq \{\mathcal{S} \text{ structure for } \mathcal{L} \mid \mathcal{S} \models \psi\} \\ &\Longleftrightarrow\; \text{Every model of } \mathcal{A} \text{ is also a model of } \{\varphi\} \end{aligned}$$

In this case we call φ a semantic consequence of $\mathcal{A}$. If $\mathcal{A} = \emptyset$, we write $\models \varphi$.

Example 1.3.8. (Axioms of Robinson arithmetic **Q**) We set $\mathcal{L}_{\mathrm{Q}} = \mathcal{L}_{\mathrm{PA}}$. The axiomatic system $\mathcal{A}_{\mathrm{Q}}$ of $\mathcal{L}_{\mathrm{Q}}$ is the set containing the following formulas:

Q1. $\forall x\, \neg(0 = \mathrm{S}(x))$

Q2. $\forall x\, \forall y\, (\mathrm{S}(x) = \mathrm{S}(y) \to x = y)$

Q3. $\forall x\, (x + 0 = x)$

Q4. $\forall x\, \forall y\, (x + \mathrm{S}(y) = \mathrm{S}(x + y))$

Q5. $\forall x\, (x \times 0 = 0)$

Q6. $\forall x\, \forall y\, (x \times \mathrm{S}(y) = (x \times y) + x)$

Q7. $\forall x\, \forall y\, (x < y \leftrightarrow \neg\exists z\, (x = y + z))$

We write $\bar{n} := \overbrace{\mathrm{S}(\ldots\mathrm{S}(0))}^{n\text{-times}}$ as an abbreviation. Obviously, these axioms try to extract central features of the natural numbers and their basic arithmetic. The following counterexample shows that **Q** is not sufficient to characterize $\mathbb{N}$

up to isomorphism: Set $M = \mathbb{N}[\diamond] = \{a_n \cdot \diamond^n + ... + a_1 \cdot \diamond + a_0 \mid a_i \in \mathbb{N}\}$. We define $I(+) : \mathbb{N}[\diamond]^2 \to \mathbb{N}[\diamond]$ as the addition in the semiring $\mathbb{N}[\diamond]$ and likewise $I(\times) : \mathbb{N}[\diamond]^2 \to \mathbb{N}[\diamond]$ as the multiplication. Furthermore, we set:

$$I(\mathrm{S}) : \mathbb{N}[\diamond] \to \mathbb{N}[\diamond]\,, p \mapsto p+1\,, \quad I(0) = 0 \in \mathbb{N}[\diamond]\,, \quad I(<) = \{(p,q) \in \mathbb{N}[\diamond]^2 \mid \neg\exists r \in \mathbb{N}[\diamond] : (p = q + r)\}.$$

According to Definition 1.3.3 this uniquely defines a structure $\mathcal{S}$ for $\mathcal{L}_\mathrm{Q}$. It is easy to prove that $\langle M, I\rangle \models \mathcal{A}_\mathrm{Q}$ but clearly this model is not isomorphic to $\mathbb{N}$ since $\diamond \neq 0$ has no predecessor.

The problem with the above system is the missing axiom of induction. A naive attempt to formalize the induction principle as a sentence would be:

$$\forall\varphi \left(\varphi\left(\bar{0}\right) \wedge \forall\bar{n}\left(\varphi\left(\bar{n}\right) \to \varphi\left(\overline{n+1}\right)\right) \to \forall\bar{n}\,\varphi\left(\bar{n}\right)\right). \tag{3}$$

There are two issues with Equation (3). First, in Definition 1.3.5 we agreed that, when interpreted, quantifiers always range over the set M, i.e. over objects. This limitation gives *first* order logic its name. Equation (3) on the other hand implies that it is possible to quantify over properties of objects, i.e. sets of objects.

Second, the above equation also implicitly quantifies over the set $\{x \mid \exists n \in \mathbb{N} : x = \bar{n}\}$, but this is only possible if the set can be formalized within $\mathcal{L}_\mathrm{Q}$, which in particular would require that we can uniquely axiomatize $\mathbb{N}$ within $\mathcal{L}_\mathrm{Q}$ – a vicious circle.

The second problem can be fixed by changing Equation (3) into:

$$\left[\varphi\left(\bar{0}\right) \wedge \forall\xi\left(\varphi(\xi) \to \varphi(\mathrm{S}(\xi))\right)\right] \to \forall\xi\,\varphi(\xi)\,. \tag{4}$$

We will define later exactly what we mean by the substitutions $\varphi\left(\bar{0}\right)$ and $\varphi(\mathrm{S}(\xi))$. Next, let us fix the first problem by adding the sentence in Equation (4) to our axiomatic system, for each formula φ and variable ξ. This idea is called axiom schema. Initially, it may seem like a disadvantage to have an infinite number of axioms. However, our axiom set is primitively recursive. In other words, one could write a computer program that checks whether a given formula is an axiom in a finite number of computational steps. We will study primitive recursive sets in more detail in Section 2.2. Thus, despite the infinite number of axioms, our axiom scheme is no less efficient than a finite list. The changes to $\mathcal{A}_\mathrm{Q}$ result in:

Example 1.3.9. (Axiomatic system of Peano arithmetic (**PA**)) The axiomatic system of Peano arithmetic is defined as the set $\mathcal{A}_\mathrm{Q}$ together with the axiom schema of induction defined above. Although stronger than $\mathcal{A}_\mathrm{Q}$, the system $\mathcal{A}_\mathrm{PA}$ is still not able to uniquely characterize the natural numbers. To prove this, we will take a look at two important theorems of model theory.

One of the interesting questions in model theory is whether there are models for a given set of axioms. The following theorem by Kurt Gödel allows us to always reduce this problem to a finite matter.

Theorem 1.3.10. *(Compactness Theorem [Gödel, 1930]) Let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ an axiomatic system in $\mathcal{L}$. There exists a model $\mathcal{M} \models \mathcal{A}$ of $\mathcal{A}$ if and only if every finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$ has a model $\mathcal{M}_0 \models \mathcal{A}_0$.*

Proof. For a detailed proof see [Bar77, pp. 23–33]. □

A different formulation of this theorem is as follows:

Theorem 1.3.11. *(Compactness Theorem - alternative formulation) Let $\mathcal{L}$ be a first-order language and $\mathcal{A} \cup \{\varphi\}$ a set of axioms in $\mathcal{L}$. If and only if $\mathcal{A} \models \varphi$, there exists a finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$ such that $\mathcal{A}_0 \models \varphi$.*

Proof. The key step in this proof is taken from [Bar77]. We will show that this statement is an immediate corollary of Theorem 1.3.10. One direction is trivial. For the other direction let us assume $\mathcal{A} \models \varphi$. We want to show that there exists a finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$ such that $\mathcal{A}_0 \models \varphi$. If $\mathcal{A}$ does not have a model, then by Theorem 1.3.10 there exists a finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$ that also does not have a model. For this subset it is trivial that $\mathcal{A}_0 \models \varphi$ and we are done. For the other case, let us assume that $\mathcal{A}$ has at least one model $\mathcal{M}$. By hypothesis, $\mathcal{M} \models \varphi$. Set $\mathcal{A}_2 := \mathcal{A} \cup \{\neg\varphi\}$. It is obvious that $\mathcal{A}_2$ does not have a model since $\mathcal{M}_2 \models \mathcal{A}_2$ would imply $\mathcal{M}_2 \models \varphi$ and $\mathcal{M}_2 \models \neg\varphi$. Therefore, by Theorem 1.3.10, it follows that there exists a finite subset $\mathcal{A}_1 \subseteq \mathcal{A}_2$ which also does not have a model. Since $\mathcal{A}_1 \cup \{\neg\varphi\}$ does not have a model, there does not exist a structure $\mathcal{S}$ for $\mathcal{L}$ such that $\mathcal{S} \models \mathcal{A}_0 := \mathcal{A}_1 \backslash \{\neg\varphi\} \subseteq \mathcal{A}$ and $\mathcal{S} \models \neg\varphi$. Thus, every model of $\mathcal{A}_0$ must satisfy φ which, by definition, implies $\mathcal{A}_0 \models \varphi$. □

In fact, both theorems are trivially equivalent:

Lemma 1.3.12. *Theorem 1.3.11 implies Theorem 1.3.10.*

Proof. One direction of Theorem 1.3.10 is trivial. For the other direction let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ a corresponding axiomatic system such that every finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$ has a model. Set $\varphi := \neg\forall x\,(x = x)$. This is always a sentence in $\mathcal{L}$ and every structure $\mathcal{S}$ for $\mathcal{L}$ will trivially satisfy $\neg\varphi$. The contrapositive of one direction of Theorem 1.3.11 states that $\mathcal{A} \not\models \varphi$ holds if $\mathcal{A}_0 \not\models \varphi$ holds for every finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$. By hypothesis every finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$ has a model and every such model will satisfy $\neg\varphi$. Hence, $\mathcal{A}_0 \not\models \varphi$ holds for every finite subset. By definition, $\mathcal{A} \not\models \varphi$ implies that there exists a model of $\mathcal{A}$ and the claim follows. □

The compactness theorem shows the existence of interesting models. For example, suppose a first-order language $\mathcal{L}_{\mathbb{R}}$ and an axiomatic system $\mathcal{A}_{\mathbb{R}}$ that attempts to axiomatize the real numbers. Note that $\mathcal{L}_{\mathbb{R}}$ is first-order, so as we are about to prove, there will be models of $\mathcal{A}_{\mathbb{R}}$ that are not isomorphic to $\mathbb{R}$. However, a famous result of one dimensional real analysis states that $\mathbb{R}$ is the only field (up to isomorphism) satisfying the usual axioms of the real numbers. Thus, at least one of those axioms must not be expressible in first-order languages. It is easy to spot the only non first-order axiom: The axiom assuring completeness, i.e. the axiom that every non-empty *subset* of $\mathbb{R}$ with a lower bound has a greatest lower bound. Instead of using the usual axioms, we choose $\mathcal{A}_{\mathbb{R}}$ to be the set of all first-order sentences true in $\mathbb{R}$. In order to do so, we introduce a constant for every real number $\{c_r \mid r \in \mathbb{R}\}$. Next, we extend this system by once again introducing a new constant $\varepsilon \in \mathcal{C}_{\mathbb{R}}$ and adding the set of formulas $\{\varepsilon < c_r \mid r > 0\} \cup \{c_0 < \varepsilon\}$ to our axiomatic system. Let us call this extension $\mathcal{A}^*_{\mathbb{R}}$. Since every finite subset of $\mathcal{A}^*_{\mathbb{R}}$ is modeled by the real numbers $\mathbb{R}$ (when choosing a suitable interpretation $\varepsilon \mapsto \frac{1}{n}, n \in \mathbb{N}$ big enough), the compactness theorem implies the existence of a model which satisfies every axiom in $\mathcal{A}^*_{\mathbb{R}}$. This justifies the idea of an infinitesimal number ε that is smaller than every positive real number, yet bigger than 0. By construction, every true first-order sentence of the real numbers still applies in such a model. In a similar manner one can achieve a model of the natural numbers that satisfies every axiom of $\mathcal{A}_{\mathrm{PA}}$ but furthermore features a non-standard number α_0 that is bigger than every standard natural number $\bar{n} < \alpha_0$.

Theorem 1.3.13. *(Upward Löwenheim–Skolem Theorem (weak version)) Let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ a corresponding axiomatic system. If there exists an infinite model $\mathcal{M} = \langle M, I\rangle \models \mathcal{A}$ of $\mathcal{A}$, then there exist arbitrarily large models of $\mathcal{A}$.*

Proof. This proof follows [Ges13, pp. 16–19]. Let $\kappa := |M|$ be the cardinality of M, and let N be an arbitrarily large set such that $|N| \geq \kappa$. We start by extending our language $\mathcal{L}_1 := \mathcal{L} \cup \{c_n \mid n \in N\}$ with a set of new constants – one for each element in N. Next, we extend our axiomatic system $\mathcal{A}_1 := \mathcal{A} \cup \{\neg(c_n = c_m) \mid n, m \in N,\ n \neq m\}$ accordingly to an axiomatic system in $\mathcal{L}_1$.

In order to apply the completeness theorem, let $\mathcal{A}_0 \subseteq \mathcal{A}_1$ be a finite subset of $\mathcal{A}_1$. Then there exists a finite subset $N_0 \subseteq N$ of N such that $\mathcal{A}_0 \subseteq \mathcal{A} \cup \{\neg(c_n = c_m) \mid n, m \in N_0,\ n \neq m\}$. We can extend $\mathcal{M}$ to a structure $\mathcal{M}_0$ for the language $\mathcal{L}_1$ by setting $M_0 = M$ and extending I to I_0 as follows: We assign each $c_n, n \in N_0$, to pairwise different elements in M_0 and extend this change to the recursive rules of Definition 1.3.3. It is possible to assign pairwise different elements of M_0 to the finitely many constants c_n because M_0 is infinite by hypothesis. This yields a model $\mathcal{M}_0 := \langle M_0, I_0 \rangle$ of $\mathcal{A}_0$.

We have shown that every finite subset of $\mathcal{A}_1$ has a model, which, according to the compactness theorem, implies the existence of a model $\mathcal{M}_1 := \langle M_1, I_1 \rangle \models \mathcal{A}_1$ of $\mathcal{A}_1$. By construction, we have $\{I_1(c_n) \mid n \in N\} \subseteq M_1$ as well as $I_1(c_n) \neq I_1(c_m)$ for all $n \neq m$. This implies $|M_1| \geq |N|$. By setting $\mathcal{M}' := \langle M' := M_1, I' := I_1|_{\mathcal{L}} \rangle$ we obtain a model of $\mathcal{A}$ for the language $\mathcal{L}$. Since $M' = M_1$ has a cardinality bigger than or equal to $|N|$ and because N was chosen of arbitrary size, the claim of this theorem follows. □

Corollary 1.3.14. *Let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ an axiomatic theory in $\mathcal{L}$. If $\mathcal{A}$ is satisfiable, then there exist non-isomorphic models of $\mathcal{A}$.*

This proves our claim that we cannot axiomatize the natural numbers – or any infinitary concept – within first-order logic in a way that characterizes that concept up to isomorphism. We also have a related result:

Theorem 1.3.15. *(Downward Löwenheim–Skolem Theorem) Let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ a corresponding satisfiable axiomatic system. Let κ be an infinite cardinal such that $|\mathcal{A}| \leq \kappa$. Then there exists a model $\mathcal{M} = \langle M, I \rangle$ of cardinality $|M| \leq \kappa$.*

Proof. For a detailed proof see [Bar77, pp. 23 – 33]. The proof is a corollary of the same lemma that Barwise uses to prove the compactness theorem, see Theorem 1.3.10. □

The downward Löwenheim–Skolem theorem shows a seemingly paradox side of model theory. Let $\mathcal{L}_{\mathrm{ZFC}}$ be the first-order language and $\mathcal{A}_{\mathrm{ZFC}}$ the corresponding axiomatic system that formulates set theory for first-order languages. Set theory is constructed in such a way, that one can find the natural numbers $\mathbb{N}$ as well as their power set $\mathcal{P}(\mathbb{N})$ in every model. Since the set $\mathcal{P}(\mathbb{N})$ has uncountably many elements, it seems that there cannot be countable models of $\mathcal{A}_{\mathrm{ZFC}}$. Theorem 1.3.15 on the other hand states that if $\mathcal{A}_{\mathrm{ZFC}}$ is satisfiable at all, it also has a countable model. We have every reason to believe in the existence of a model of $\mathcal{A}_{\mathrm{ZFC}}$, which thereby seems to lead to a contradiction known as Skolem's paradox. It can be resolved by understanding that the notions of cardinality can be different from inside the model and when looked at from the outside.

By combining the upward and downward version of the Löwenheim–Skolem theorem we get:

Corollary 1.3.16. *(Löwenheim–Skolem Theorem) Let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ a corresponding satisfiable axiomatic system. Let κ be an infinite cardinal such that $|\mathcal{A}| \leq \kappa$. Then for every infinite cardinal $\eta \geq \kappa$ there exists a model $\mathcal{M} = \langle M, I \rangle$ of cardinality $|M| = \eta$.*

Proof. Let us revisit the proof of the upward Löwenheim–Skolem theorem, see Theorem 1.3.13. We use the same notation and choose N as a set of cardinality η. In the proof, we showed that the extended axiomatic system $\mathcal{A}_1 \supseteq \mathcal{A}$ in $\mathcal{L}_1$ has a model $\mathcal{M}_1$. By applying the downward Löwenheim–Skolem theorem to $\mathcal{A}_1$, we get a model $\mathcal{M}_1' = \langle M_1', I_1' \rangle$ with cardinality $\left|M_1'\right| \leq |\mathcal{A}_1|$. Note that since $\kappa \leq \eta = |N|$, we have:

$$|N| \leq |\mathcal{A}_1| = |\mathcal{A} \cup \{\neg(c_n = c_m) \mid n, m \in N,\ n \neq m\}| \leq \kappa + |N|^2 = |N| .$$

The last equality uses the axiom of choice. For a reference, see e.g. [Zer10]. Thus, the axiomatic system $\mathcal{A}_1$ has cardinality $|N| = \eta$. On the other hand we also proved that, by construction, every model of $\mathcal{A}_1$ has at least cardinality $|N| = \eta$. It follows that the so obtained model $\mathcal{M}_1'$ of $\mathcal{A}_1$ has exactly the desired cardinality. The claim follows, by reducing $\mathcal{M}_1'$ to a model of $\mathcal{A}$ for the language $\mathcal{L}$ in the same way we did at the end of the proof of Theorem 1.3.13. □

1.4 Logical Axioms and Rules of Inference

So far, we have introduced the idea of languages, along with the rules for forming terms, formulas, and sentences correctly. Then we defined a way to interpret these statements by using metalogics to study models of certain systems. In this section we will define the notion of proofs.

Notation 2. *(Rules of inference) To define which deductions can be made, we agree on a new notation: Let $\mathcal{L}$ be a first-order language and $\varphi_1, \dots, \varphi_n, \psi_{\varphi_1,\dots,\varphi_n}$ some formulas in $\mathcal{L}$, where the metavariable $\psi_{\varphi_1,\dots,\varphi_n}$ denotes a formula dependent on $\varphi_1, \dots, \varphi_n$. We call*

$$\frac{\begin{array}{c}\varphi_1\\ \vdots\\ \varphi_n\end{array}}{\psi_{\varphi_1,\dots,\varphi_n}}$$

an inference rule and say that the formula $\psi_{\varphi_1,\dots,\varphi_n}$ can be obtained by $\varphi_1, \dots, \varphi_n$.

Rules of inference are the justification for any kind of deduction in proofs. Without them, proving anything but axioms would be impossible.

Definition 1.4.1. The following list of inference rules will be denoted as $\mathfrak{I}^{\mathrm{H}}$:

$$\text{Modus ponens (\textbf{MP}):}\quad \frac{\begin{array}{c}\varphi\\ \varphi \to \psi\end{array}}{\psi}, \qquad\qquad \text{Generalisation (\textbf{G}):}\quad \frac{\varphi}{\forall\xi\ \varphi}. \tag{5}$$

The variable ξ must not occur freely in any formula $\chi \in \mathcal{A}$ of the corresponding axiomatic system $\mathcal{A}$.

Modus ponens formalizes the intuitive idea that it should be a proof of ψ to prove φ as well as $\varphi \to \psi$. The generalization rule formalizes the idea that if something is proved for any x (" Let x be an arbitrary...") this should be a proof that the statement holds for all x. The restriction that x cannot be a free variable in any formula $\varphi \in \mathcal{A}$ is redundant in our case, since we only allowed sentences as axioms in Definition 1.3.1. If we were to extend this definition to general formulas, this restriction would be necessary, since the occurrence of a free variable x in an axiom would prevent us from choosing x arbitrarily.

For example, imagine the formula $(x = 0)$ as an additional axiom in $\mathcal{A}_{\mathrm{PA}}$ and $\varphi = \exists y\ (x = \bar{2} \times y)$. Of course φ holds because we can use the axiom to prove that x will always be $\bar{0}$ and therefore φ trivially holds with $y = 0$. But this should not prove that the formula $\forall x\ \exists y\ (x = \bar{2} \times y)$ holds.

In addition to inference rules, we allow some logical axioms. For this purpose we introduce a new notation.

Notation 3. *Let φ be a formula, ξ a variable and t a term. By*

$$\varphi\{\xi \mapsto t\} =: \varphi(t),$$

we denote the substitution of all free occurrences of ξ in φ by t. Note that if ξ is not free in φ, $\varphi\{\xi \mapsto t\} = \varphi$. We can use the abbreviation $\varphi(t)$ if it is clear which variable we will replace with t. If t contains a variable bound by a quantifier in φ, we say that there is a collision in the substitution. A substitution is called collision-free if it has no collision. For example, the following substitution is collision-free:

$$\varphi(x) = \exists y\ (x < y)\,,\ t = \mathrm{S}(\mathrm{S}(z)) \quad \rightsquigarrow \quad \varphi\{x \mapsto t\} = \exists y\ (\mathrm{S}(\mathrm{S}(z)) < y)\,,$$

while this on is not:

$$\varphi(x) = \exists y\ (x < y)\,,\ t = \mathrm{S}(\mathrm{S}(y)) \quad \rightsquigarrow \quad \varphi\{x \mapsto t\} = \exists y\ (\mathrm{S}(\mathrm{S}(y)) < y)\,.$$

If we want to substitute several terms $t_1, \dots, t_k$ for variables $\xi_1, \dots, \xi_k$, we write

$$\varphi\{\xi_1, \dots, \xi_k \mapsto t_1, \dots, t_k\} := \varphi\{\xi_1 \mapsto t_1\} \dots \{\xi_k \mapsto t_k\} \ ,$$

for subsequently applying k substitutions.

We will use the following scheme as logical axioms:

Definition 1.4.2. Let $\mathcal{L}$ be a first-order language. The following list of logical axioms will be denoted as $\mathcal{A}^{\mathrm{H}}_{\mathrm{Lo}}$:

L1.	$\varphi \lor \varphi \to \varphi$	**L**2.	$\psi \to \varphi \lor \psi$
L3.	$\varphi \lor \psi \to \psi \lor \varphi$	**L**4.	$\varphi \lor (\psi \lor \chi) \to \psi \lor (\varphi \lor \chi)$
L5.	$(\psi \to \chi) \to (\varphi \lor \psi \to \varphi \lor \chi)$	**L**6.	$\forall \xi\, \varphi \to \varphi\{\xi \mapsto t\}$
L7.	$\forall x\, (\varphi \lor \psi) \to (\varphi \lor \forall x\, \psi)$	**L**8.	$\forall \xi\ (\xi = \xi)$
L9.	$\forall \xi\, \forall \zeta\ \left[\xi = \zeta \to \left(\varphi(\xi) \to \varphi(\zeta)\right)\right]$	**L**10.	$[\forall \xi\, (\varphi \to \psi)] \to [\forall \xi\, \varphi \to \forall \xi\, \psi]$

For Axiom **L**6, the substitution has to be collision-free and for **L**7, the variable ξ must not appear (freely) in φ. As usual the above axioms are to be interpreted as axiom schemes, i.e. the actual list of axioms is infinite.

Two remarks on Definition 1.4.1 and 1.4.2:

- The first five axioms were introduced by Russell and Whitehead in their famous work "Principia Mathematica", see [WR10, Axioms *1.2–*1.6, pp. 100–101]. Axiom **L**4 has been shown not to be independent of the other axioms in [Ber26], so we could easily omit it. Note that the choice of logical axioms (**L**1 - **L**3, **L**5 - **L**9) and rules of inference were inspired by Gödel as the basis of an important theorem which we will discuss in the next section, see [Göd29]. However, Gödel used a different formulation of first-order languages which technically does not agree with our definitions. He allowed second-order variables with the restriction that they cannot be quantified over. For the definition of second-order formulas, we refer to Section 1.8 and for a detailed introduction to Gödel's version of first-order logic to Hilbert and Ackermann, see [HA38]. The last axiom is adopted from [Rau09, Chapter 3.6].
- There are numerous variants of first-order predicate logic that employ different inference rules or logical axioms. The theorems we will discuss in this thesis will be so robust that the precise formulation is mostly a technical matter. As a result, the reader may regard the definitions used as an historical example.

Now that we have established the basic building blocks, we can define what exactly a *formal proof* of a formula φ should be. Intuitively, it should explain exactly how to derive φ by applying inference rules to some axioms. Furthermore, this process should be algorithmically verifiable. There are several ways to define such systems. For our purposes, Hilbert-style calculus is well suited:

Notation 4. *(Hilbert-style proof). A Hilbert-style proof is a finite list of formulas, where each line is either an axiom or the result of applying an inference rule to previous lines. The last line is the statement we proved. A proof of the formula φ with n steps has the following form:*

Proof of: φ		
1.	*Some formula*	*Reference to the axiom used*
2.	*Some formula*	*Reference to the axiom or inference rule used*

Proof of: φ *(continuation)*		
$\vdots$	$\dots$	$\dots$
n.	φ	*Reference to the axiom or inference rule used*

To keep these proofs concise, we will introduce abbreviations. In addition, we will use extra lines to translate between our formula notations, such as the definitions for $\wedge, \to$ (see Notation 1). This translation, however, occurs only in the meta language, and these lines are omitted in the actual proof.

Definition 1.4.3. A set of inference rules $\mathcal{I}$ together with a set of logical Axioms $\mathcal{A}_{\mathrm{Lo}}$ in the context of a proof system is called deductive system.

Our set of inference rules and logical axioms defines a deductive system. We denote it as $\mathcal{H} = \left(\mathcal{A}^{\mathrm{H}}_{\mathrm{Lo}}, \mathcal{I}^{\mathrm{H}}\right)$ and call it Hilbert-style deductive system.

Notation 5. *(Provability) Let $\mathcal{T} = (\mathcal{L}, \mathcal{A})$ be a formal theory, $\mathcal{D}$ a deductive system and φ a formula in $\mathcal{L}$. We call φ* provable *in $\mathcal{T}$ iff there exists a (finite) proof of φ using $\mathcal{D}$. In this case we write:*

$$\mathcal{T} \vdash \varphi\,.$$

Unless stated otherwise, we will consider a formal proof to be a proof in Hilbert-style. Nonetheless, there are many other proof systems. Another common method is natural deduction. In this system, a proof of φ would be a tree of formulas, where each leaf is an axiom and the only root is φ. Each formula in this tree can be derived by applying a rule of inference to its children. The method of natural deduction arose because Hilbert-style proofs can quickly feel unnatural. On the other hand, the Hilbert system is designed to be very minimalistic, which can be useful for proving metatheorems.

To conclude this section, we highlight a famous result that will be relevant later:

Theorem 1.4.4. *(Deduction theorem) Let $\mathcal{T} = (\mathcal{L}, \mathcal{A})$ be a theory and φ, ψ formulas in $\mathcal{L}$ such that φ is a sentence. Assume the Hilbert-style calculus $\mathcal{H}$ as a deductive system. We have:*

$$\mathcal{T} + \varphi := \left(\mathcal{L}, \mathcal{A} \cup \{\varphi\}\right) \vdash \psi \qquad \Longrightarrow \qquad \mathcal{T} \vdash (\varphi \to \psi)$$

Proof. The theorem can be shown by induction over the number of proof steps and case distinction in the inductive step. An explicit proof can be found, for example, in [Men97]. □

1.5 Examples of Formal Proofs

Recall the definition of Peano arithmetic in Example 1.3.9. Let's call the corresponding theory $\mathcal{T}_{\mathrm{PA}}$. To conduct formal proofs, we will use the Hilbert-style deductive system $\mathcal{H}$. Throughout the section, note that there is still a significant difference between the formal proofs and our versions due to abbreviations and notation.

Lemma 1.5.1. *(Existential generalization) For any formula φ, any variable ξ and for any term t such that substituting t for ξ in φ will not cause any collisions, we have:*

$$\varphi\{\xi \mapsto t\} \to \exists \xi\, \varphi\,. \tag{6}$$

Proof. This follows immediately from logical axiom **L**6:

Proof of: $\varphi\{\xi \mapsto t\} \to \exists \xi\, \varphi$		
1.	$\forall \xi\, \neg\varphi \to \neg\varphi\{\xi \mapsto t\}$	logical axiom **L**6

Proof of: $\varphi\{\xi \mapsto t\} \to \exists\xi\,\varphi$ (continuation)		
2.	$\neg(\forall\xi\,\neg\varphi) \vee \neg\varphi\{\xi \mapsto t\}$	Notation ($\to$)
3.	$[\neg(\forall\xi\,\neg\varphi) \vee \neg\varphi\{\xi \mapsto t\}] \to [\neg\varphi\{\xi \mapsto t\} \vee \neg(\forall\xi\,\neg\varphi)]$	Logical axiom **L**3
4.	$\neg\varphi\{\xi \mapsto t\} \vee \neg(\forall\xi\,\neg\varphi)$	**MP**(2,3)
5.	$\varphi\{\xi \mapsto t\} \to \neg(\forall\xi\,\neg\varphi)$	Notation ($\to$)
6.	$\varphi\{\xi \mapsto t\} \to \exists\xi\,\varphi$	Notation ($\exists$)

□

Lemma 1.5.2. *(Transitivity of implication) Implications are transitive, i.e. we have:*

$$\textbf{TR}1.\quad (\varphi \to \psi) \to \big((\psi \to \chi) \to (\varphi \to \chi)\big) \qquad\qquad \textbf{TR}2.\quad (\psi \to \chi) \to \big((\varphi \to \psi) \to (\varphi \to \chi)\big)$$

Proof. The following list gives a proof of **TR**2:

Proof of: $(\psi \to \chi) \to \big((\varphi \to \psi) \to (\varphi \to \chi)\big)$		
1.	$(\psi \to \chi) \to \big((\neg\varphi \vee \psi) \to (\neg\varphi \vee \chi)\big)$	Logic ax. **L**5
2.	$(\psi \to \chi) \to \big((\varphi \to \psi) \to (\varphi \to \chi)\big)$	Not. ($\to$)

This immediately gives a proof of **TR**1:

Proof of: $(\varphi \to \psi) \to \big((\psi \to \chi) \to (\varphi \to \chi)\big)$		
1.	$(\psi \to \chi) \to \big((\varphi \to \psi) \to (\varphi \to \chi)\big)$	**TR**2
2.	$\neg(\psi \to \chi) \vee \big(\neg(\varphi \to \psi) \vee (\varphi \to \chi)\big)$	Not. ($\to$)
3.	$\Big[\neg(\psi \to \chi) \vee \big(\neg(\varphi \to \psi) \vee (\varphi \to \chi)\big)\Big] \to \Big[\neg(\varphi \to \psi) \vee \big(\neg(\psi \to \chi) \vee (\varphi \to \chi)\big)\Big]$	Logic ax. **L**4
4.	$\neg(\varphi \to \psi) \vee \big(\neg(\psi \to \chi) \vee (\varphi \to \chi)\big)$	**MP**(2, 3)
5.	$(\varphi \to \psi) \to \big((\psi \to \chi) \to (\varphi \to \chi)\big)$	Not. ($\to$)

□

Recall that for a correct proof in Hilbert style, every line should be an axiom or the result of an inference rule applied to lines above. For that reason, the proof of **TR**1 is an abbreviation of the actual proof in which we had to first repeat the proof of **TR**2 before stating its result. To make proofs more readable, we will allow this kind of abbreviation in our proofs. We will also allow the following rule of inference:

$$\text{Specialisation } (\mathbf{S}): \qquad \frac{\forall\xi_1\,\forall\xi_2\ldots\forall\xi_k\;\;\varphi(\xi_1,\ldots,\xi_k)}{\varphi\{\xi_1,\ldots,\xi_k \mapsto t_1,\ldots,t_k\}}, \tag{7}$$

for successive collision-free substitutions, which can be easily proved for any k, by repeatedly applying logical axiom **L**6. In a similar vein, we will allow this rule of inference:

$$\text{Generalisation } (\mathbf{G}): \qquad \frac{\varphi}{\forall\xi_1\,\forall\xi_2\ldots\forall\xi_k\;\;\varphi(\xi_1,\ldots,\xi_k)} \tag{8}$$

This is simply an abbreviation of repeatedly applying the generalisation rule **G**, introduced in Equation (5).

Lemma 1.5.3. *(Symmetry of "=") The equals sign is a symmetric relation. That is, we have:*

$$\forall x \forall y\;\;(x = y \to y = x)\,.$$

Proof. For simplicity, define $\varphi_x(z) := \varphi(z, y) := (z = x)$. We have:

Proof of: $\forall x\, \forall y\ \ (x = y \to y = x)$		
1.	$\forall x\, \forall y\ \big(x = y \to (\varphi_x(x) \to \varphi_x(y))\big)$	Logical ax. **L**9
2.	$x = y \to (\varphi_x(x) \to \varphi_x(y))$	**S**(1)
3.	$\neg(x = y) \vee (\neg\varphi_x(x) \vee \varphi_x(y))$	Not. ($\to$)
4.	$\Big(\neg(x = y) \vee \big(\neg\varphi_x(x) \vee \varphi_x(y)\big)\Big) \to \Big(\neg\varphi_x(x) \vee \big(\neg(x = y) \vee \varphi_x(y)\big)\Big)$	Logical ax. **L**4
5.	$\neg\varphi_x(x) \vee \big(\neg(x = y) \vee \varphi_x(y)\big)$	**MP**(3, 4)
6.	$\varphi_x(x) \to (x = y \to \varphi_x(y))$	Not. ($\to$)
7.	$x = x \to (x = y \to y = x)$	Def. (φ_x)
8.	$\forall x\ \ (x = x)$	Logical ax. **L**8
9.	$x = x$	**S**(8)
10.	$x = y \to y = x$	**MP**(9, 7)
11.	$\forall x\, \forall y\ \ (x = y \to y = x)$	**G**(10)

□

Lemma 1.5.4. *(Extensionality of* S(•)*) The successor function is extensional, i.e. we have:*

$$\forall x\, \forall y\ \ (x = y \to \mathrm{S}(x) = \mathrm{S}(y)).$$

Proof. We use the same trick as in Lemma 1.5.3 by defining: $\varphi_x(z) := (\mathrm{S}(x) = \mathrm{S}(z))$ and get:

Proof of: $\forall x\, \forall y\ \ (x = y \to \mathrm{S}(x) = \mathrm{S}(y))$		
1.	$\forall x\, \forall y\ \big(x = y \to (\varphi_x(x) \to \varphi_x(y))\big)$	Logical ax. **L**9
2.	$x = y \to (\varphi_x(x) \to \varphi_x(y))$	**S**(1)
3.	$\neg(x = y) \vee (\neg\varphi_x(x) \vee \varphi_x(y))$	Not. ($\to$)
4.	$\big(\neg(x = y) \vee (\neg\varphi_x(x) \vee \varphi_x(y))\big) \to \big(\neg\varphi_x(x) \vee (\neg(x = y) \vee \varphi_x(y))\big)$	Logical ax. **L**4
5.	$\neg\varphi_x(x) \vee (\neg(x = y) \vee \varphi_x(y))$	**MP**(3,4)
6.	$\varphi_x(x) \to (x = y \to \varphi_x(y))$	Not. ($\to$)
7.	$\mathrm{S}(x) = \mathrm{S}(x) \to (x = y \to \mathrm{S}(x) = \mathrm{S}(y))$	Def. (φ_x)
8.	$\forall x\ \ (x = x)$	Logical ax. **L**8
9.	$\mathrm{S}(x) = \mathrm{S}(x)$	**S**(8)
10.	$x = y \to \mathrm{S}(x) = \mathrm{S}(y)$	**MP**(9, 7)
11.	$\forall x\, \forall y\ \ (x = y \to \mathrm{S}(x) = \mathrm{S}(y))$	**G**(10)

□

Lemma 1.5.5. *(Double negation) Negations are self inverse operations, i.e. we have:*

$$\mathbf{DN}1.\quad \varphi \to \neg(\neg\varphi) \qquad\qquad \mathbf{DN}2.\quad \neg(\neg\varphi) \to \varphi$$

Proof. We start with **DN**1:

Proof of: $\varphi \to \neg(\neg\varphi)$		
1.	$\neg\varphi \to \neg(\neg\varphi) \vee \neg\varphi$	Logical ax. **L**2
2.	$\neg(\neg\varphi) \vee \neg\varphi \to \neg\varphi \vee \neg(\neg\varphi)$	Logical ax. **L**3
3.	$\big[\neg\varphi \to \neg(\neg\varphi) \vee \neg\varphi\big] \to \Big[\big(\neg(\neg\varphi) \vee \neg\varphi \to \neg\varphi \vee \neg(\neg\varphi)\big) \to \big(\neg\varphi \to \neg\varphi \vee \neg(\neg\varphi)\big)\Big]$	**TR**1
4.	$\big(\neg(\neg\varphi) \vee \neg\varphi \to \neg\varphi \vee \neg(\neg\varphi)\big) \to \big(\neg\varphi \to \neg\varphi \vee \neg(\neg\varphi)\big)$	**MP**(1,3)
5.	$\neg\varphi \to \neg\varphi \vee \neg(\neg\varphi)$	**MP**(2, 4)
6.	$\neg(\neg\varphi) \vee \big(\neg\varphi \vee \neg(\neg\varphi)\big)$	Not. ($\to$)
7.	$\Big(\neg(\neg\varphi) \vee \big(\neg\varphi \vee \neg(\neg\varphi)\big)\Big) \to \Big(\neg\varphi \vee \big(\neg(\neg\varphi) \vee \neg(\neg\varphi)\big)\Big)$	Logical ax. **L**4

Proof of: $\varphi \to \neg(\neg\varphi)$ (continuation)		
8.	$\neg\varphi \vee \big(\neg(\neg\varphi) \vee \neg(\neg\varphi)\big)$	**MP**(6, 7)
9.	$\varphi \to \neg(\neg\varphi) \vee \neg(\neg\varphi)$	Not. ($\to$)
10.	$\neg(\neg\varphi) \vee \neg(\neg\varphi) \to \neg(\neg\varphi)$	Logical ax. **L**1
11.	$\big[\varphi \to \neg(\neg\varphi) \vee \neg(\neg\varphi)\big] \to \Big[\big(\neg(\neg\varphi) \vee \neg(\neg\varphi) \to \neg(\neg\varphi)\big) \to \big(\varphi \to \neg(\neg\varphi)\big)\Big]$	**TR**1
12.	$\big(\neg(\neg\varphi) \vee \neg(\neg\varphi) \to \neg(\neg\varphi)\big) \to \big(\varphi \to \neg(\neg\varphi)\big)$	**MP**(9, 11)
13.	$\varphi \to \neg(\neg\varphi)$	**MP**(10, 12)

Using the above result we can prove **DN**2:

Proof of: $\neg(\neg\varphi) \to \varphi$		
1.	$\neg\varphi \to \neg(\neg(\neg\varphi))$	**DN**1
2.	$\big(\neg\varphi \to \neg(\neg(\neg\varphi))\big) \to \big((\varphi \vee \neg\varphi) \to (\varphi \vee \neg(\neg(\neg\varphi)))\big)$	Logical ax. **L**5
3.	$(\varphi \vee \neg\varphi) \to (\varphi \vee \neg(\neg(\neg\varphi)))$	**MP**(1,2)
4.	$\varphi \to \varphi \vee \varphi$	Logical ax. **L**2
5.	$\varphi \vee \varphi \to \varphi$	Logical ax. **L**1
6.	$(\varphi \to \varphi \vee \varphi) \to \big((\varphi \vee \varphi \to \varphi) \to (\varphi \to \varphi)\big)$	**TR**1
7.	$(\varphi \vee \varphi \to \varphi) \to (\varphi \to \varphi)$	**MP**(4, 6)
8.	$\varphi \to \varphi$	**MP**(5,7)
9.	$\neg\varphi \vee \varphi$	Not. ($\to$)
10.	$\neg\varphi \vee \varphi \to \varphi \vee \neg\varphi$	Logical ax. **L**3
11.	$\varphi \vee \neg\varphi$	**MP**(9, 10)
12.	$\varphi \vee \neg(\neg(\neg\varphi))$	**MP**(11, 3)
13.	$\varphi \vee \neg(\neg(\neg\varphi)) \to \neg(\neg(\neg\varphi)) \vee \varphi$	Logical ax. **L**3
14.	$\neg(\neg(\neg\varphi)) \vee \varphi$	**MP**(12, 13)
15.	$\neg(\neg\varphi) \to \varphi$	Not. ($\to$)

$\square$

Note that we also proved the law of excluded middle as an interim result:

$$\textbf{LEM}: \quad \neg\varphi \vee \varphi\,.$$

A pattern that wastes a lot of space in the proof above is to formulate **TR**1 or **TR**2 and then apply modus ponens twice. Let us introduce an inference rule to abbreviate this in the future:

$$\text{Transitivity } (\textbf{TR}): \quad \frac{\begin{array}{c}\varphi \to \psi \\ \psi \to \chi\end{array}}{\varphi \to \chi} \tag{9}$$

This can easily be proved:

Proof of: **TR**		
1.	$\varphi \to \psi$	precondition
2.	$\psi \to \chi$	precondition
3.	$(\varphi \to \psi) \to ((\psi \to \chi) \to (\varphi \to \chi))$	**TR**1
4.	$(\psi \to \chi) \to (\varphi \to \chi)$	**MP**(1,3)
5.	$\varphi \to \chi$	**MP**(2,4)

In a formally correct proof we would still have to do all the steps, but for the coming proofs we will use the above abbreviation. In a similar vein, we can introduce the following rules of inference:

$$\text{Associativity } (\mathbf{AS})\text{:} \quad \frac{\varphi \lor (\psi \lor \chi)}{\psi \lor (\varphi \lor \chi)},$$

which, of course, is just an abbreviation for applying logical axiom **L**4 and modus ponens. This also yields

$$\text{Associativity } (\mathbf{AS})\text{:} \quad \frac{\varphi \to (\psi \to \chi)}{\psi \to (\varphi \to \chi)},$$

by applying the definition of "→".

Lemma 1.5.6. *The AND operation satisfies:*

$$\varphi \to (\psi \to (\varphi \land \psi))$$

Proof.

Proof of: $\varphi \to (\psi \to (\varphi \land \psi))$		
1.	$\neg(\neg\varphi \lor \neg\psi) \lor (\neg\varphi \lor \neg\psi)$	**LEM**
2.	$\neg\varphi \lor \big(\neg(\neg\varphi \lor \neg\psi) \lor \neg\psi\big)$	**AS**(1)
3.	$\varphi \to \big(\neg(\neg\varphi \lor \neg\psi) \lor \neg\psi\big)$	Not. (→)
4.	$\big(\neg(\neg\varphi \lor \neg\psi) \lor \neg\psi\big) \to \big(\neg\psi \lor \neg(\neg\varphi \lor \neg\psi)\big)$	Logical ax. **L**3
5.	$\varphi \to \big(\neg\psi \lor \neg(\neg\varphi \lor \neg\psi)\big)$	**TR**(3, 4)
6.	$\varphi \to \big(\psi \to \neg(\neg\varphi \lor \neg\psi)\big)$	Not. (→)
7.	$\varphi \to (\psi \to (\varphi \land \psi))$	Not. (∧)

□

This can be helpful in proofs to state $\varphi \land \psi$ when having already proved φ and ψ. To conclude this chapter, we will show that addition in the natural numbers is commutative. We need the following auxiliary lemma to do so:

Lemma 1.5.7. *The following formula holds:*

$$\forall k\, \forall n \;\; (k + \mathrm{S}(n) = \mathrm{S}(k) + n)$$

Proof.

Proof of: $\forall k\, \forall n \;\; (k + \mathrm{S}(n) = \mathrm{S}(k) + n)$		
1.	$\forall x\, \forall y \;\; (x + \mathrm{S}(y) = \mathrm{S}(x + y))$	Peano ax. **PA**4
2.	$k + \mathrm{S}(0) = \mathrm{S}(k + 0)$	**S**(1)
3.	$\forall x \;\; (x + 0 = x)$	Peano ax. **PA**3
4.	$k + 0 = k$	**S**(3)
5.	$\forall x\, \forall y \;\; \big[x = y \to \big(k + \mathrm{S}(0) = \mathrm{S}(x) \to k + \mathrm{S}(0) = \mathrm{S}(y)\big)\big]$	Logical ax. **L**9
6.	$k + 0 = k \to \big(k + \mathrm{S}(0) = \mathrm{S}(k + 0) \to k + \mathrm{S}(0) = \mathrm{S}(k)\big)$	**S**(5)
7.	$k + \mathrm{S}(0) = \mathrm{S}(k + 0) \to k + \mathrm{S}(0) = \mathrm{S}(k)$	**MP**(4,6)
8.	$k + \mathrm{S}(0) = \mathrm{S}(k)$	**MP**(2,7)
9.	$\forall x\, \forall y \;\; (x = y \to y = x)$	Lemma 1.5.3
10.	$k + \mathrm{S}(0) = \mathrm{S}(k) \to \mathrm{S}(k) = k + \mathrm{S}(0)$	**S**(9)
11.	$\mathrm{S}(k) = k + \mathrm{S}(0)$	**MP**(8,10)

Proof of: $\forall k\, \forall n\ \ (k + \mathrm{S}(n) = \mathrm{S}(k) + n)$ (continuation)		
12.	$\mathrm{S}(k) + 0 = \mathrm{S}(k)$	**S**(3)
13.	$\mathrm{S}(k) + 0 = \mathrm{S}(k) \to \mathrm{S}(k) = \mathrm{S}(k) + 0$	**S**(9)
14.	$\mathrm{S}(k) = \mathrm{S}(k) + 0$	**MP**(12, 13)
15.	$\forall x\, \forall y\ \left[x = y \to \left(x = \mathrm{S}(k) + 0 \to y = \mathrm{S}(k) + 0\right)\right]$	Logical ax. **L**9
16.	$\mathrm{S}(k) = k + \mathrm{S}(0) \to \left(\mathrm{S}(k) = \mathrm{S}(k) + 0 \to k + \mathrm{S}(0) = \mathrm{S}(k) + 0\right)$	**S**(15)
17.	$\mathrm{S}(k) = \mathrm{S}(k) + 0 \to k + \mathrm{S}(0) = \mathrm{S}(k) + 0$	**MP**(11,16)
18.	$k + \mathrm{S}(0) = \mathrm{S}(k) + 0$	**MP**(14,17)
19.	$\forall x\, \forall y\ \ (x = y \to \mathrm{S}(x) = \mathrm{S}(y))$	Lemma 1.5.4
20.	$k + \mathrm{S}(n) = \mathrm{S}(k) + n \to \mathrm{S}(k + \mathrm{S}(n)) = \mathrm{S}(\mathrm{S}(k) + n)$	**S**(19)
21.	$k + \mathrm{S}(\mathrm{S}(n)) = \mathrm{S}(k + \mathrm{S}(n))$	**S**(1)
22.	$k + \mathrm{S}(\mathrm{S}(n)) = \mathrm{S}(k + \mathrm{S}(n)) \to \mathrm{S}(k + \mathrm{S}(n)) = k + \mathrm{S}(\mathrm{S}(n))$	**S**(9)
23.	$\mathrm{S}(k + \mathrm{S}(n)) = k + \mathrm{S}(\mathrm{S}(n))$	**MP**(21,22)
24.	$\mathrm{S}(k) + \mathrm{S}(n) = \mathrm{S}(\mathrm{S}(k) + n)$	**S**(1)
25.	$\mathrm{S}(k) + \mathrm{S}(n) = \mathrm{S}(\mathrm{S}(k) + n) \to \mathrm{S}(\mathrm{S}(k) + n) = \mathrm{S}(k) + \mathrm{S}(n)$	**S**(9)
26.	$\mathrm{S}(\mathrm{S}(k) + n) = \mathrm{S}(k) + \mathrm{S}(n)$	**MP**(24, 25)
27.	$\forall x\, \forall y\ \left[x = y \to \left(x = k + \mathrm{S}(\mathrm{S}(n)) \to y = k + \mathrm{S}(\mathrm{S}(n))\right)\right]$	Logical ax. **L**9
28.	$\mathrm{S}(k + \mathrm{S}(n)) = \mathrm{S}(\mathrm{S}(k) + n) \to \left(\mathrm{S}(k + \mathrm{S}(n)) = k + \mathrm{S}(\mathrm{S}(n)) \to \mathrm{S}(\mathrm{S}(k) + n) = k + \mathrm{S}(\mathrm{S}(n))\right)$	**S**(27)
29.	$\mathrm{S}(k + \mathrm{S}(n)) = k + \mathrm{S}(\mathrm{S}(n)) \to \left(\mathrm{S}(k + \mathrm{S}(n)) = \mathrm{S}(\mathrm{S}(k) + n) \to \mathrm{S}(\mathrm{S}(k) + n) = k + \mathrm{S}(\mathrm{S}(n))\right)$	**AS**(28)
30.	$\mathrm{S}(k + \mathrm{S}(n)) = \mathrm{S}(\mathrm{S}(k) + n) \to \mathrm{S}(\mathrm{S}(k) + n) = k + \mathrm{S}(\mathrm{S}(n))$	**MP**(23 ,29)
31.	$\forall x\, \forall y\ \left[x = y \to \left(x = \mathrm{S}(k) + \mathrm{S}(n) \to y = \mathrm{S}(k) + \mathrm{S}(n)\right)\right]$	Logical ax. **L**9
32.	$\mathrm{S}(\mathrm{S}(k) + n) = k + \mathrm{S}(\mathrm{S}(n)) \to \left(\mathrm{S}(\mathrm{S}(k) + n) = \mathrm{S}(k) + \mathrm{S}(n) \to k + \mathrm{S}(\mathrm{S}(n)) = \mathrm{S}(k) + \mathrm{S}(n)\right)$	**S**(31)
33.	$\mathrm{S}(k + \mathrm{S}(n)) = \mathrm{S}(\mathrm{S}(k) + n) \to \left(\mathrm{S}(\mathrm{S}(k) + n) = \mathrm{S}(k) + \mathrm{S}(n) \to k + \mathrm{S}(\mathrm{S}(n)) = \mathrm{S}(k) + \mathrm{S}(n)\right)$	**TR**(30,32)
34.	$\mathrm{S}(\mathrm{S}(k) + n) = \mathrm{S}(k) + \mathrm{S}(n) \to \left(\mathrm{S}(k + \mathrm{S}(n)) = \mathrm{S}(\mathrm{S}(k) + n) \to k + \mathrm{S}(\mathrm{S}(n)) = \mathrm{S}(k) + \mathrm{S}(n)\right)$	**AS**(33)
35.	$\mathrm{S}(k + \mathrm{S}(n)) = \mathrm{S}(\mathrm{S}(k) + n) \to k + \mathrm{S}(\mathrm{S}(n)) = \mathrm{S}(k) + \mathrm{S}(n)$	**MP**(26,34)
36.	$k + \mathrm{S}(n) = \mathrm{S}(k) + n \to k + \mathrm{S}(\mathrm{S}(n)) = \mathrm{S}(k) + \mathrm{S}(n)$	**TR**(20,35)
37.	$\varphi_k(n) := (k + \mathrm{S}(n) = \mathrm{S}(k) + n)$	Def.
38.	$\varphi_k(0)$	see line 18
39.	$\varphi_k(n) \to \varphi_k(\mathrm{S}(n))$	see line 36
40.	$\forall n\ \left(\varphi_k(n) \to \varphi_k(\mathrm{S}(n))\right)$	**G**(39)
41.	$\varphi_k(0) \to \left[\forall n\ \left(\varphi_k(n) \to \varphi_k(\mathrm{S}(n))\right) \to \left(\varphi_k(0) \land \forall n\ \left(\varphi_k(n) \to \varphi_k(\mathrm{S}(n))\right)\right)\right]$	Lemma 1.5.6
42.	$\forall n\ \left(\varphi_k(n) \to \varphi_k(\mathrm{S}(n))\right) \to \left(\varphi_k(0) \land \forall n\ \left(\varphi_k(n) \to \varphi_k(\mathrm{S}(n))\right)\right)$	**MP**(38, 41)
43.	$\varphi_k(0) \land \forall n\ \left(\varphi_k(n) \to \varphi_k(\mathrm{S}(n))\right)$	**MP**(40, 42)
44.	$\left[\varphi_k(0) \land \forall n \left(\varphi_k(n) \to \varphi_k(\mathrm{S}(n))\right)\right] \to \forall n\ \ \varphi_k(n)$	**IND**
45.	$\forall n\ \ \varphi_k(n)$	**MP**(43, 44)
46.	$\forall n\ \ (k + \mathrm{S}(n) = \mathrm{S}(k) + n)$	Def. (φ_k)
47.	$\forall k\, \forall n\ \ (k + \mathrm{S}(n) = \mathrm{S}(k) + n)$	**G**(46)

□

With these results it is now easy to prove commutativity of addition:

Lemma 1.5.8. *(Commutativity of addition) We have, as expected:*

$$\forall n\, \forall k\ \ (n + k = k + n)$$

Proof.

Proof of: $\forall n\, \forall k\ (n + k = k + n)$		
1.	$\forall x\ (x + 0 = x)$	Peano ax. **PA**3
2.	$0 + 0 = 0$	**S**(1)
3.	$\forall x\, \forall y\ (x = y \to S(x) = S(y))$	Lemma 1.5.4
4.	$0 + r = r \to S(0 + r) = S(r)$	**S**(3)
5.	$\forall x\, \forall y\ (x + S(y) = S(x + y))$	Peano ax. **PA**4
6.	$0 + S(r) = S(0 + r)$	**S**(5)
7.	$\forall x\, \forall y\ \left[x = y \to \left(0 + S(r) = x \to 0 + S(r) = y\right)\right]$	Logical ax. **L**9
8.	$S(0 + r) = S(r) \to \left(0 + S(r) = S(0 + r) \to 0 + S(r) = S(r)\right)$	**S**(7)
9.	$0 + S(r) = S(0 + r) \to \left(S(0 + r) = S(r) \to 0 + S(r) = S(r)\right)$	**AS**(8)
10.	$S(0 + r) = S(r) \to 0 + S(r) = S(r)$	**MP**(6, 9)
11.	$0 + r = r \to 0 + S(r) = S(r)$	**TR**(4, 10)
12.	$\varphi(r) := (0 + r = r)$	Def.
13.	$\varphi(0)$	see line 2
14.	$\varphi(r) \to \varphi(S(r))$	see line 11
15.	$\forall r\ (\varphi(r) \to \varphi(S(r)))$	**G**(14)
16.	$\varphi(0) \to \left[\forall r\ \left(\varphi(r) \to \varphi(S(r))\right) \to \left(\varphi(0) \wedge \forall r\ \left(\varphi(r) \to \varphi(S(r))\right)\right)\right]$	Lemma 1.5.6
17.	$\forall r\ \left(\varphi(r) \to \varphi(S(r))\right) \to \left(\varphi(0) \wedge \forall r\ \left(\varphi(r) \to \varphi(S(r))\right)\right)$	**MP**(13,16)
18.	$\varphi(0) \wedge \forall r\ \left(\varphi(r) \to \varphi(S(r))\right)$	**MP**(15,17)
19.	$\left[\varphi(0) \wedge \forall r\ \left(\varphi(r) \to \varphi(S(r))\right)\right] \to \forall r\ \ \varphi(r)$	**IND**
20.	$\forall r\ \ \varphi(r)$	**MP**(18, 19)
21.	$\varphi(n)$	**S**(20)
22.	$0 + n = n$	Def. (φ)
23.	$n + 0 = n$	**S**(1)
24.	$\forall x\, \forall y\ (x = y \to y = x)$	Lemma 1.5.3
25.	$0 + n = n \to n = 0 + n$	**S**(24)
26.	$n + 0 = n \to n = n + 0$	**S**(24)
27.	$n = 0 + n$	**MP**(22 ,25)
28.	$n = n + 0$	**MP**(23, 26)
29.	$\forall x\, \forall y\ \left[x = y \to \left(x = 0 + n \to y = 0 + n\right)\right]$	Logical ax. **L**9
30.	$n = n + 0 \to \left(n = 0 + n \to n + 0 = 0 + n\right)$	**S**(29)
31.	$n = 0 + n \to n + 0 = 0 + n$	**MP**(28, 30)
32.	$n + 0 = 0 + n$	**MP**(27, 31)
33.	$k + S(n) = S(k + n)$	**S**(5)
34.	$n + S(k) = S(n + k)$	**S**(5)
35.	$k + S(n) = S(k + n) \to S(k + n) = k + S(n)$	**S**(24)
36.	$n + S(k) = S(n + k) \to S(n + k) = n + S(k)$	**S**(24)
37.	$S(k + n) = k + S(n)$	**MP**(33, 35)
38.	$S(n + k) = n + S(k)$	**MP**(34, 36)
39.	$n + k = k + n \to S(n + k) = S(k + n)$	**S**(3)
40.	$\forall x\, \forall y\ \left[x = y \to \left(x = n + S(k) \to y = n + S(k)\right)\right]$	Logical ax. **L**9
41.	$S(n + k) = S(k + n) \to \left(S(n + k) = n + S(k) \to S(k + n) = n + S(k)\right)$	**S**(40)
42.	$S(n + k) = n + S(k) \to \left(S(n + k) = S(k + n) \to S(k + n) = n + S(k)\right)$	**AS**(41)
43.	$S(n + k) = S(k + n) \to S(k + n) = n + S(k)$	**MP**(38, 42)
44.	$n + k = k + n \to S(k + n) = n + S(k)$	**TR**(39, 43)

Proof of: $\forall n\,\forall k\;\;(n+k=k+n)$ (continuation)		
45.	$\mathrm{S}(k+n)=k+\mathrm{S}(n)\to\big(\mathrm{S}(k+n)=n+\mathrm{S}(k)\to k+\mathrm{S}(n)=n+\mathrm{S}(k)\big)$	**S**(40)
46.	$\mathrm{S}(k+n)=n+\mathrm{S}(k)\to k+\mathrm{S}(n)=n+\mathrm{S}(k)$	**MP**(37, 45)
47.	$n+k=k+n\to k+\mathrm{S}(n)=n+\mathrm{S}(k)$	**TR**(44, 46)
48.	$k+\mathrm{S}(n)=n+\mathrm{S}(k)\to n+\mathrm{S}(k)=k+\mathrm{S}(n)$	**S**(24)
49.	$n+k=k+n\to n+\mathrm{S}(k)=k+\mathrm{S}(n)$	**TR**(47, 48)
50.	$\forall k\,\forall n\;\;(k+\mathrm{S}(n)=\mathrm{S}(k)+n)$	Lemma 1.5.7
51.	$k+\mathrm{S}(n)=\mathrm{S}(k)+n$	**S**(50)
52.	$\forall x\,\forall y\;\big[x=y\to\big(n+\mathrm{S}(k)=x\to n+\mathrm{S}(k)=y\big)\big]$	Logical ax. **L9**
53.	$k+\mathrm{S}(n)=\mathrm{S}(k)+n\to\big(n+\mathrm{S}(k)=k+\mathrm{S}(n)\to n+\mathrm{S}(k)=\mathrm{S}(k)+n\big)$	**S**(52)
54.	$n+\mathrm{S}(k)=k+\mathrm{S}(n)\to n+\mathrm{S}(k)=\mathrm{S}(k)+n$	**MP**(51,53)
55.	$n+k=k+n\to n+\mathrm{S}(k)=\mathrm{S}(k)+n$	**TR**(49, 54)
56.	$\psi(k):=(n+k=k+n)$	Def.
57.	$\psi(0)$	see line 32
58.	$\psi(k)\to\psi(\mathrm{S}(k))$	see line 55
59.	$\forall k\;\big(\psi(k)\to\psi(\mathrm{S}(k))\big)$	**G**(58)
60.	$\psi(0)\to\Big[\forall k\,\big(\psi(k)\to\psi(\mathrm{S}(k))\big)\to\Big(\psi(0)\land\forall k\,\big(\psi(k)\to\psi(\mathrm{S}(k))\big)\Big)\Big]$	Lemma 1.5.6
61.	$\forall k\;\big(\psi(k)\to\psi(\mathrm{S}(k))\big)\to\Big(\psi(0)\land\forall k\;\big(\psi(k)\to\psi(\mathrm{S}(k))\big)\Big)$	**MP**(57, 60)
62.	$\psi(0)\land\forall k\;\big(\psi(k)\to\psi(\mathrm{S}(k))\big)$	**MP**(59, 61)
63.	$\Big[\psi(0)\land\forall k\;\big(\psi(k)\to\psi(\mathrm{S}(k))\big)\Big]\to\forall k\;\;\psi(k)$	**IND**
64.	$\forall k\;\;\psi(k)$	**MP**(62, 63)
65.	$\forall k\;\;(n+k=k+n)$	Def.
66.	$\forall n\,\forall k\;\;(n+k=k+n)$	**G**(65)

□

The formally correct proof, that is, if we write out the abbreviations and omit lines for notations and definitions, has *342* steps. However, it is probably possible to find a shorter proof. Therefore, it would be an interesting metamathematical question to ask for the minimum number of steps needed to prove this theorem.

1.6 Gödel's Completeness Theorem

So far we have not discussed the connection between proof and truth. Given a theory $\mathcal{T}=(\mathcal{L},\mathcal{A})$ and a deductive system $\mathcal{D}=(\mathcal{A}_{\mathrm{Lo}},\mathcal{I})$, a proof of φ should imply that φ is true in every realization that satisfies the axioms of $\mathcal{A}$. In other words, φ should be a semantic consequence of $\mathcal{A}$.

Definition 1.6.1. (Soundness of deductive systems) Let $\mathcal{L}$ be a fixed first-order language and $\mathcal{D}=(\mathcal{A}_{\mathrm{Lo}},\mathcal{I})$ a deductive system. We call $\mathcal{D}$ *sound* iff for every theory $\mathcal{T}=(\mathcal{L},\mathcal{A})$ and every sentence φ in $\mathcal{L}$, the following holds:

$$\mathcal{T}\vdash\varphi\implies\mathcal{T}\models\varphi\,.$$

We define $\mathcal{T}\models\varphi:\iff\mathcal{A}\models\varphi$ and call φ a semantic consequence of $\mathcal{T}$.

Conversely, one could ask if every semantic consequence φ of $\mathcal{T}$ is provable in $\mathcal{T}$:

Definition 1.6.2. (Completeness of deductive systems and formal theories) Let $\mathcal{T}=(\mathcal{L},\mathcal{A})$ be a formal theory and $\mathcal{D}=(\mathcal{A}_{\mathrm{Lo}},\mathcal{I})$ a deductive system. We call $\mathcal{D}$ *complete* iff for every sentence φ in $\mathcal{L}$,

$$\models\varphi\implies\mathcal{A}_{\mathrm{Lo}}\vdash\varphi\,.$$

Moreover, we call $\mathcal{T}$ *complete* iff for every formula φ in $\mathcal{L}$ the following holds:

$$\mathcal{T} \models \varphi \implies \mathcal{T} \vdash \varphi\,.$$

It is desirable to have a sound and complete system. That way, proven statements will be true, and true statements will be provable. In the case of first-order logic, both are possible:

Lemma 1.6.3. (Soundness Lemma) *Let* $\mathcal{D} = (\mathcal{A}_{\text{Lo}}, \mathcal{I})$ *be a deductive system with the following properties:*

- *all logical axioms are tautologies:* $\models \Psi$*, for every* $\Psi \in \mathcal{A}_{\text{Lo}}$
- *the inference rules are truth preserving: Given a structure S with assignment s,* $IR(\varphi_1, \ldots, \varphi_n)\big|_{\mathcal{V}\leftarrow s}$ *is guaranteed to be true for any inference rule* $IR \in \mathcal{I}$ *with n inputs, as long as* $\varphi_i\big|_{\mathcal{V}\leftarrow s}$ *is true for all i.*

In this case, $\mathcal{D}$ *is sound.*

Proof. Let $\mathcal{T} = (\mathcal{L}, \mathcal{A})$ be some theory and the sequence $(\Psi_1, \ldots, \Psi_n)$ a proof in $\mathcal{T}$. We want to show that $\mathcal{T} \models \Psi_n$. Our meta-proof is by induction over n: For the base case $n = 1$, $\Psi_n = \Psi_1$ must be an axiom, so either $\Psi_n \in \mathcal{A}_{\text{Lo}}$ or $\Psi_n \in \mathcal{A}$. In the first case, $\mathcal{T} \models \Psi_n$, since $\Psi_n \in \mathcal{A}_{\text{Lo}}$ is a tautology. In the second case, note that $\mathcal{T} \models \Psi_n \iff \mathcal{A} \models \Psi_n$. The right side is trivial: If all axioms of $\mathcal{A}$ (including Ψ_n) are true in some structure, then Ψ_n is also true in that structure. For the step case $n \mapsto n+1$, consider some proof $(\Psi_1, \ldots, \Psi_n, \Psi_{n+1})$. By the induction hypothesis, $\mathcal{T} \models \Psi_i$ for all $i \leq n$. Either $\Psi_{n+1} \in \mathcal{A}_{\text{Lo}} \cup \mathcal{A}$ and $\mathcal{T} \models \Psi_{n+1}$ like we showed in the base case. Otherwise, Ψ_{n+1} is the result of applying a truth-preserving inference rule to some logical consequences of $\mathcal{T}$ and thus $\mathcal{T} \models \Psi_{n+1}$. □

Therefore, to ensure soundness, it is sufficient to verify that all logical axioms are tautologies (true in every possible world) and the rules of inference cannot lead to a false conclusion if only true propositions have been used. This is the best we can hope for, and it is easy to confirm that $\mathcal{H}$ satisfies these requirements.

Theorem 1.6.4. *(Gödel's Completeness Theorem, Original Version) Let* $\mathcal{L}$ *be a formal language. The Hilbert-style deductive system* $\mathcal{H}$ *is complete. In signs:*

$$\models \varphi \implies \mathcal{A}_{\text{Lo}} \vdash \varphi\,,$$

for every formula in $\mathcal{L}$.

Proof. Gödel originally proved the theorem in 1929 as part of his dissertation. For a modern proof of this theorem, see e.g. [Rau09, pp. 121-126]. □

Theorem 1.6.5. *(Gödel's Completeness Theorem, Strong Version) Let* $\mathcal{L}$ *be a formal language and* $\mathcal{A}$ *a corresponding axiomatic system. Together with the Hilbert-style system* $\mathcal{H}$*, the theory* $\mathcal{T} = (\mathcal{L}, \mathcal{A})$ *is complete. In signs:*

$$\mathcal{T} \models \varphi \implies \mathcal{T} \vdash \varphi\,,$$

for every formula in $\mathcal{L}$.

Proof. Let φ be some formula in $\mathcal{L}$ such that $\mathcal{A} \models \varphi$. We have to provide a proof for φ. By Theorem 1.3.11 (compactness theorem), there exists a finite subset $\{\chi_1, \ldots, \chi_k\} = \mathcal{A}_0 \subseteq \mathcal{A}$ such that $\mathcal{A}_0 \models \varphi$. Define

$$\tilde{\varphi} := (\chi_1 \wedge \ldots \wedge \chi_k) \to \varphi\,.$$

Since $\mathcal{A}_0 \models \varphi$, a given structure satisfies φ or does not satisfy all axiom of $\mathcal{A}_0$. In both cases $\tilde{\varphi}$ is true, i.e. $\models \tilde{\varphi}$. Thus, by the weak formulation of Gödel's completeness theorem, $\mathcal{A}_{\text{Lo}} \vdash \tilde{\varphi}$. Let us denote this proof as the sequence $(\Psi_1, \ldots, \Psi_n)$, where $\Psi_n = \tilde{\varphi}$. Using Lemma 1.5.6 and applying modus ponens k times, we can construct a proof $(\Psi_{n+1}, \ldots, \Psi_{n+s})$ of $\chi_1 \wedge \ldots \wedge \chi_k$. The sequence $(\Psi_1, \ldots, \Psi_{n+s}, \varphi)$ is a valid proof of φ in $\mathcal{T}$. To derive φ in the last step, we applied modus ponens to $\Psi_n = \tilde{\varphi}$ and $\Psi_{n+s} = \chi_1 \wedge \ldots \wedge \chi_k$. □

Both theorems are tailored to our deductive system $\mathcal{H}$. We could have chosen a different system, and there are many possible combinations of inference rules and logical axioms that are in fact complete. Since the main idea remains the same, the proofs can be transferred directly.

1.7 Consequences of Gödel's Completeness Theorem

We used the compactness theorem to prove the strong version of the completeness theorem. One can also derive the compactness theorem as a corollary of the completeness theorem:

Lemma 1.7.1. *Theorem 1.6.5 (completeness theorem) implies Theorem 1.3.11 (compactness theorem).*

Proof. Let $\mathcal{L}$ be a first-order language and $\mathcal{A} \cup \{\varphi\}$ a set of axioms in $\mathcal{L}$. Assume $\mathcal{A} \models \varphi$. We have to show that there exists a finite subset $\mathcal{A}_0 \subseteq \mathcal{A}$ such that $\mathcal{A}_0 \models \varphi$. The completeness theorem states that $\mathcal{T} = (\mathcal{L}, \mathcal{A}) \vdash \varphi$. Write $(\Psi_1, \ldots, \Psi_n)$ for the corresponding proof. Let $\mathcal{A}_0 := \{\Psi_1, \ldots, \Psi_n\} \cap \mathcal{A}$ be the (finite) set of axioms used in the proof. This proof will also be valid in $\mathcal{T}_0 := (\mathcal{L}, \mathcal{A}_0)$, thus $\mathcal{T}_0 \vdash \varphi$. Since $\mathcal{H}$ is sound, $\mathcal{T}_0 \models \varphi$. By definition, this is equivalent to $\mathcal{A}_0 \models \varphi$. □

In fact, both theorems are closely related. The compactness theorem states that only a finite part of the axiomatic system contributes to a *semantic* consequence. Similarly, only a finite part of the axiomatic system contributes to a *syntactic* consequence (proof). The completeness theorem shows that syntactic and semantic consequences are not only similar, but even equivalent. It creates a bridge between the world of model theory and the world of formal deduction. Moreover, it shows that the definition of *provability* depends only on the formal theory and is independent of the (complete and sound) deduction system we choose.

Definition 1.7.2. (Consistency of theories) Let $\mathcal{T}$ be a theory and $\mathcal{D}$ a fixed deduction system. We call $\mathcal{T}$ inconsistent if there exists a formula φ such that

$$\mathcal{T} \vdash \varphi \qquad \text{and} \qquad \mathcal{T} \vdash \neg\varphi\,.$$

In this case we write $\mathcal{T} \vdash \bot$. A theory is called consistent if it is not inconsistent.

The *principle of explosion*, "ex falso quodlibet", asserts that any proposition can be deduced from a false statement. This fact is sometimes integrated as a logical rule:

$$\mathcal{T} \vdash (\bot \to \varphi)\,.$$

In such systems, the symbol $\bot$ is included in the language $\mathcal{L}$ and is called bottom. It is understood as a generic "false" assertion. In our deduction system, we can prove this fact from the existing axioms. Let φ, ϕ be any two formulas:

Proof of: $(\varphi \wedge \neg\varphi) \to \phi$		
1.	$\varphi \to \neg(\neg\varphi)$	**DN**1
2.	$\neg\varphi \vee \neg(\neg\varphi)$	Not. $(\to)$
3.	$(\neg\varphi \vee \neg(\neg\varphi)) \to \neg(\neg(\neg\varphi \vee \neg(\neg\varphi)))$	**DN**1
4.	$\neg(\neg(\neg\varphi \vee \neg(\neg\varphi)))$	**MP**(2, 3)
5.	$\neg(\varphi \wedge \neg\varphi)$	Not. $(\wedge)$
6.	$\neg(\varphi \wedge \neg\varphi) \to \phi \vee \neg(\varphi \wedge \neg\varphi)$	Logical ax. **L**2
7.	$\phi \vee \neg(\varphi \wedge \neg\varphi)$	**MP**(5, 6)
8.	$\phi \vee \neg(\varphi \wedge \neg\varphi) \to \neg(\varphi \wedge \neg\varphi) \vee \phi$	Logical ax. **L**3
9.	$\neg(\varphi \wedge \neg\varphi) \vee \phi$	**MP**(7, 8)
10.	$(\varphi \wedge \neg\varphi) \to \phi$	Not. $(\to)$

If our theory is inconsistent, there exists some formula φ such that $\mathcal{T} \vdash \varphi$ and $\mathcal{T} \vdash \neg\varphi$. We can combine these two proofs to get a proof for $\varphi \wedge \neg\varphi$. Then, by combining this proof with the above argument and applying modus ponens, we can construct a proof of ϕ. Since ϕ was an arbitrary formula, our theory proves anything. This is why inconsistent theories are not useful to work with: Every statement is a theorem. Furthermore, since the underlying deductive system is sound, they cannot have a realization.

Conversely, one can ask whether *consistent* theories always have a model. This is indeed the case:

Theorem 1.7.3. (Model existence theorem) *Let $\mathcal{L}$ be a first-order language and $\mathcal{A}$ a corresponding axiomatic system. Let $\mathcal{T} = (\mathcal{L}, \mathcal{A})$ be the resulting theory and let $\mathcal{D}$ be a sound and complete deductive system, e.g. $\mathcal{D} = \mathcal{H}$. Then, the following two statements are equivalent:*

$$\mathcal{T} \text{ has a model} \qquad \Longleftrightarrow \qquad \mathcal{T} \text{ is consistent}$$

Proof. One direction is trivial: If $\mathcal{T}$ has a model, it must be consistent since $\mathcal{D}$ is sound. For the other direction, we will show its contrapositive. Suppose $\mathcal{T}$ has no models. Pick any formula φ in $\mathcal{L}$. Trivially, φ and $\neg\varphi$ will be true in every model of $\mathcal{T}$, hence $\mathcal{T} \models \varphi$ and $\mathcal{T} \models \neg\varphi$. By the completeness theorem, this implies $\mathcal{T} \vdash \varphi$ and $\mathcal{T} \vdash \neg\varphi$, which is equivalent to the inconsistency of $\mathcal{T}$. ◻

Another consequence of the completeness theorem is that we can easily construct an algorithm that lists all the semantic consequences of a theory $\mathcal{T}$ by listing all the theorems of $\mathcal{T}$. In Chapter 3 we will examine and refine a specific Turing machine designed for this purpose. Sets that can be enumerated algorithmically are called *recursively enumerable*. The original definition of semantic consequence (formulas true in any model of $\mathcal{T}$) did not suggest the existence of an algorithm for this situation. Consequently, the completeness theorem showed that the set of semantic consequences is, indeed, recursively enumerable.

1.8 Exploring Second-Order and Higher-Order Logic

So far, we only allowed to quantify over individual variables. Second- and higher-order logic extends on the ideas of first-order logic, to allow more general quantification. We follow [Vää21].

Second-order languages are defined in a similar way as first-order languages. They use the same alphabet as before (see Definition 1.2.1), except that the set of variables $\mathcal{V} = \{x_i, R_i, F_i \mid i \in \mathbb{N}\}$ distinguishes between individual variables x_i, relation variables R_i, and function variables F_i. Accordingly, we extend the domain of the arity function $\#(\bullet)$ to include relation and function variables. Furthermore, the symbol "=" may be excluded. As before, we allow the use of other symbols as part of our meta-logic if it improves readability.

Definition 1.8.1. (Terms) We define the notion of terms inductively:

- all individual variables x_i and constants c_i are terms,
- if f is a function symbol with $\#(f) = n$ and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term,
- if F_i is a function variable with $\#(F_i) = n$ and $t_1, ..., t_n$ are terms, then $F_i(t_1, ..., t_n)$ is a term.

Definition 1.8.2. (Formulas) We define formulas inductively:

- For a given n-ary relation symbol r, n-ary relation variable R_i and terms $t_1, ..., t_n$, we define

$$(r(t_1, ..., t_n)), \qquad (R_i(t_1, ..., t_n))$$

  as (atomic) formulas.

- If φ, ψ are formulas, x_i is an individual variable, R_i a relational variable and F_i a function variable, then

$$(\varphi \vee \psi), \quad (\neg\varphi), \quad (\forall x_i\, \varphi), \quad (\forall R_i\, \varphi), \quad (\forall F_i\, \varphi)$$

  are formulas of the language $\mathcal{L}$.

The notion of free and bound variables is defined like in first-order logic. Sentences are formulas without free variables. Note that we did not include the equal sign in our language. Instead we can give the following definition:

$$t_1 = t_2 \quad := \quad \forall R \;\; (R(t_1) \rightarrow R(t_2)) \wedge (R(t_2) \rightarrow R(t_1)) \tag{10}$$

Intuitively, this formula defines two terms to be equal if they share the same truth value on each relation. It is a formula that explicitly has to range over all relations and therefore cannot be stated in first-order logic. The principle that two objects are identical if they share the same properties is called *identity of indiscernibles*. The converse is called *principle of the indiscernibility of identicals*, or *Leibniz's Law*. For an interesting discussion of the philosophy of equality, we recommend [NC22].

Model theory is similar to that for first-order languages. We define structures $\mathcal{S} = \langle M, I \rangle$ as in Definition 1.3.3, but adopt variable assignment to include the extended notion of terms:

Definition 1.8.3. (Variable assignment) Let $\mathcal{L}$ be a second-order language. Let s be a function with domain $\mathcal{V}$ such that individual variables are mapped to an element in M, relation variables with arity n to a subset of M^n and function variables with arity n to a function $M^n \rightarrow M$. We call s an assignment for $\mathcal{S}$ and define the notion $t\,\big|_{\mathcal{V}\leftarrow s}$ for a term t recursively by the following rules:

- If t is a constant, $t = c_i$, then $t\,\big|_{\mathcal{V}\leftarrow s} := I(t) \in M$,
- If t is an individual variable, $t = x_i$, then $t\,\big|_{\mathcal{V}\leftarrow s} := s(t) \in M$,
- $f(t_1, ..., t_n)\,\big|_{\mathcal{V}\leftarrow s} := I(f)(t_1\,\big|_{\mathcal{V}\leftarrow s}, \dots, t_n\,\big|_{\mathcal{V}\leftarrow s})$ for every function symbol $f \in \mathcal{F}$ and terms $t_1, ..., t_n$,
- $F_i(t_1, ..., t_n)\,\big|_{\mathcal{V}\leftarrow s} := s(F_i)(t_1\,\big|_{\mathcal{V}\leftarrow s}, \dots, t_n\,\big|_{\mathcal{V}\leftarrow s})$ for every function variable $F_i \in \mathcal{V}$ and terms $t_1, ..., t_n$.

As expected, we will interpret formulas in a structure as follows:

Definition 1.8.4. (Semantics of a language $\mathcal{L}$) Let $\mathcal{L}$ be a second-order language, $\mathcal{S}$ a corresponding structure and s an assignment for $\mathcal{S}$. We define recursively for terms $t_1, t_2, \dots, t_n$, formulas φ, ψ and variables x_i, R_i, F_i:

- $\mathcal{S} \models r(t_1, ..., t_n)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if $\left(t_1\,\big|_{\mathcal{V}\leftarrow s}, ..., t_n\,\big|_{\mathcal{V}\leftarrow s}\right) \in I(r)$,
- $\mathcal{S} \models R_i(t_1, ..., t_n)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if $\left(t_1\,\big|_{\mathcal{V}\leftarrow s}, ..., t_n\,\big|_{\mathcal{V}\leftarrow s}\right) \in s(R_i)$,
- $\mathcal{S} \models (\varphi \vee \psi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if $\mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s}$ or $\mathcal{S} \models \psi\,\big|_{\mathcal{V}\leftarrow s}$,
- $\mathcal{S} \models (\neg\varphi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if not $\mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s}$,
- $\mathcal{S} \models (\forall x_i\; \varphi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if for all $a \in M : \mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s[x_i/a]}$,
- $\mathcal{S} \models (\forall R_i\; \varphi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if for all $A \subseteq M^{\#R_i} : \mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s[R_i/A]}$,
- $\mathcal{S} \models (\forall F_i\; \varphi)\,\big|_{\mathcal{V}\leftarrow s}$ if and only if for all functions $g\colon M^{\#F_i} \rightarrow M : \mathcal{S} \models \varphi\,\big|_{\mathcal{V}\leftarrow s[F_i/g]}$.

The notion $s[\;/\;]$ is defined like in Definition 1.3.5.

With this definition (syntactic) relation variables correspond to properties (or sets) in the semantic context. Consequently, quantifying over them can be interpreted as quantifying over properties. We define axiomatic systems and models of these systems in the usual way.

Example 1.8.5. (Natural numbers in second-order logic) Let $\mathcal{L}_{\mathbb{N}}$ be the second-order language given by $\mathcal{C} = \{0\}$, $\mathcal{R} = \emptyset$ and $\mathcal{F} = \{G\}$. Define $\mathcal{A}$ as $\{\Psi_1, \Psi_2, \Psi_3\}$, where:

$$\begin{aligned}
\Psi_1 &= \forall x\, \neg[G(x) = 0] \\
\Psi_2 &= \forall x\, \forall y\, [G(x) = G(y) \to x = y] \\
\Psi_3 &= \forall R\, \Big[\Big(R(0) \wedge \forall x\, [R(x) \to R(G(x))]\Big) \to \forall x\, R(x)\Big]
\end{aligned}$$

Then, every model $\mathcal{M} = \langle M, I\rangle \models \mathcal{A}$ of $\mathcal{A}$ is isomorphic to $\mathbb{N}$. More precisely, $(M, I(0), I(G)) \cong (\mathbb{N}, 0, +_1)$. Thus, $\mathcal{A}$ defines the natural numbers up to isomorphism. This was not possible in first-order logic due to Theorem 1.3.13 (Löwenheim–Skolem), see Example 1.3.9.

Note that for a fixed structure $\mathcal{S}$, every formula with n free variables corresponds to a set

$$\Big\{(a_1, \dots, a_n) \in M^n \;\Big|\; \mathcal{S} \models \varphi\, \big|_{\mathcal{V} \leftarrow s[x_1/a_1] \circ \dots \circ s[x_n/a_n]}\Big\} \subseteq M^n$$

Thus, if given an n-ary relation variable R such that $s(R)$ is the above set, we have:

$$\mathcal{S} \models \forall x_1 \dots \forall x_n\, [R(x_1, \dots, x_n) \leftrightarrow \varphi(x_1, \dots, x_n)]\,.$$

In this sense, formulas can be interpreted as relations. The *comprehension principle* states that such a variable always exists:

$$\exists R\, \forall x_1 \dots \forall x_n\, [R(x_1, \dots, x_n) \leftrightarrow \varphi(x_1, \dots x_n)] \tag{11}$$

Here, φ is any formula in which R is not free. The comprehension principle is usually integrated as one of the axioms of second-order predicate logic.

Similar to second-order logic, one can define third-, fourth-, ... order logic by allowing variables for properties of properties or for properties of properties of properties, and so on. At the semantic level, this nesting carries over to $\mathfrak{P}(\mathfrak{P}(M))$, $\mathfrak{P}(\mathfrak{P}(\mathfrak{P}(M)))$, and so on – $\mathfrak{P}(\bullet)$ denoting the power set. Although second- and higher-order languages are more expressive than first-order languages, this advantage comes at a high price: For instance, with the result of Gödel's first incompleteness theorem, it is apparent that the completeness theorem for first-order logic cannot hold in higher-order logic. Moreover, as we will see in Section 3.3, modern set theory (**ZFC**) can be defined in first-order logic and is strong enough to formalize vast areas of modern mathematics.

1.9 Type Theory

We follow [Coq22]. One formulation of Russell's paradox for second-order systems goes as follows: Define the formula $\varphi(S) := \neg S(S)$. Then, find a relation R corresponding to this formula and ask whether $\varphi(R)$ or $\neg\varphi(R)$ hold. Both lead to a contradiction. The system in which Russell discovered this antinomy was that of Gottlob Frege at the beginning of the last century. Although Frege's system did not allow for the exact construction we have provided, he was able to derive the contradiction in different ways from one of his basic axioms. In each derivation, the paradox arises by self-reference. To avoid such vicious circles for his own project, Principia Mathematica, Russell invented type theory. A specific type is assigned to each object:

1. i is the type of terms,
2. if φ is a formula with $n \in \mathbb{N}_0$ free variables of types $\tau_1, ..., \tau_n$, then $(\tau_1, ..., \tau_n)$ is the type of φ.

Functions are interpreted as $(n + 1)$-ary relations. For example, the following formulas have these types:

$$\begin{array}{rcc} x & \rightsquigarrow & i \\ \varphi := \forall x\, \exists y\; y = x & \rightsquigarrow & (\,) \\ \psi(x) := \exists y\; y = x & \rightsquigarrow & (i) \\ \chi(R, x) := \forall y\; R(y) \lor R(x) & \rightsquigarrow & ((i), i) \end{array}$$

This immediately prohibits formulas like $S(S)$, since if S accepts inputs of type τ, then the type of S is (τ). Moreover, the rule eliminates the cause of Russell's paradox and prevents its construction. This theory of types is now known as simple type theory. But Russell feared that it was not enough. The formula

$$\varphi(x) := \forall R\, (\neg R(x) \lor R(x)),$$

for example, can itself be seen as a relation – so does the quantification reference the definition it is itself part of? This property is known as *impredicativity*. Out of fear of circularity, Russell introduced a second system, the so called *ramified hierarchy*. To simplify the resulting theory, he also added an axiom, the so called *axiom of reducibility*. We will not go into the details of his construction, as it will not be relevant to the following sections. In fact, in his proof of the first incompleteness theorem, Gödel uses (a slightly different formulation of) simple type theory, which we will discuss in the next chapter. For further reading on impredicativity, we recommend [Cro18] as well as [Sch61]. For more information on type theory, we recommend [Coq22].

2 Gödel's Incompleteness Theorems

Definition 2.0.1. (Incompleteness of theories) Let $\mathcal{T} = (\mathcal{L}, \mathcal{A})$ be a theory. $\mathcal{T}$ is called incomplete if there exists a formula φ in $\mathcal{L}$ such that neither $\mathcal{T} \vdash \varphi$ nor $\mathcal{T} \vdash \neg\varphi$ hold.

This notion of incompleteness is not to be confused with the kind - used in Gödel's completeness theorem. Unless otherwise noted, we refer to this new definition throughout this chapter. In 1931, Gödel demonstrated that nearly all mathematical theories are incomplete. We follow his original proof of the first incompleteness theorem as presented in [Göd31]. To improve readability we will use modern notation.

2.1 Gödel's System P

Gödel introduced a concrete theory, the system P, for which he proved the (first) incompleteness theorem. It is the system obtained by using the Peano axioms in the logic of Principia Mathematica. In particular, it is a higher-order system using (simple) type theory. Unlike Gödel, we include the operations of addition and multiplication. This is not necessary, since they can be defined in Gödel's original language. But it makes our notation simpler.

Definition 2.1.1. (Language of P) The Language $\mathcal{L}_\mathrm{P}$ of P is defined as in Section 1.8, where $\mathcal{C} = \{0\}, \mathcal{V} = \{x_1, y_1, z_1, \dots, x_2, y_2, z_2, \dots, x_3, y_3, z_3, \dots\}, \mathcal{F} = \{\mathrm{S}, +, \times\}$ and $\mathcal{R} = \emptyset$. Variables x_i, y_i and z_i are of type i, where type-1 variables are individual variables and ones with type $i \geq 2$, relation variables. The type of relation variables dictates whether it represents properties of natural numbers, properties of properties of natural numbers, and so on. The arity of variables of type $i \geq 2$ is always set to 1.

We will write x_i, y_i, z_i to indicate a specific variable of type i, ξ_i or ζ_i to indicate any variable of type i and ξ or ζ to indicate any variable of any type. If we want to talk about multiple specific variables of type 1, we may denote them by $\alpha_1, \alpha_2, \dots$. Variables ξ_1 can be considered as individuals, namely those corresponding to a natural number. Variables ξ_2 model a set of natural numbers, ξ_3 a set of sets of natural numbers, and so on. If ξ_n is a variable of type n, we will call the corresponding modelled objects (numbers, sets of numbers, and so on), objects of type n. Note that we did not allow relation variables with arity $k \geq 2$, nor did we allow function variables. However, this is not limiting: A function f of arity k can be represented as a relation of arity $k + 1$,

$$f \equiv R(a_1, \dots, a_{k+1}) \equiv \{(a_1, \dots, a_k, a_{k+1}) \mid a_{k+1} = f(a_1, \dots, a_k)\}.$$

Furthermore, k-tuples of objects of the same type can be represented as sets,

$$(b_1, \dots, b_k) \equiv \{\{b_1\}, \{b_1, b_2\}, \{b_1, b_2, b_3\}, \dots, \{b_1, \dots, b_k\}\}.$$

This is also known as "Kuratowski's pairing". Moreover, note that an object of type n can be represented to have type $n + s$ for every $s \in \mathbb{N}$ by nesting it in sets,

$$b \equiv \underbrace{\{\dots\{}_{s\text{-times}} b\}\dots\},$$

and that we can thus represent any k-tuple of objects of different types as a set of elements of the same type. However, we can represent a set of objects of the same type as a unary relation of that type.

Definition 2.1.2. (Terms / Signs of type 1) As in Definition 1.8.1 we define the following as terms:

- Variables of type 1: $x_1, y_1, z_1, \dots,$
- Constant(s): 0,

- If t_1, t_2 are terms, then so are $\mathrm{S}(t_1), t_1 + t_2$ and $t_1 \times t_2$.

Terms will also be called signs of type 1. Signs of type n for $n \geq 2$ are defined to be the same as variables of type n.

Next, we define atomic formulas, or as Gödel calls them, *elementary formulas*.

Definition 2.1.3. (Atomic formulas / elementary formulas) Let ξ_2 be a variable of type 2 and t a sign of type 1. Let ξ_{n+1} be a variable of type $n+1$ and ζ_n a variable of type n. Then, the expressions

$$\xi_2(t)\,, \qquad \xi_{n+1}(\zeta_n)$$

are called elementary (atomic) formulas.

The fact that a sign of type $n+1$ accepts exactly a sign of type n as an input is the implementation of simple type theory in Gödel's system P. It prevents the construction of sets that contain themselves and therefore antinomies like Russell's paradox. Formulas are now defined in the same way as in Definition 1.8.2. Like earlier, we will also use abbreviations (for $\exists, \rightarrow$ and $\wedge$) as well as differently sized and square brackets. If it improves readability, we may omit brackets. Of course, these notations exist only in our meta language.

We will call a formula φ...

- ... sentential (or closed) formula, if $\mathbf{FV}(\varphi) = \emptyset$, i.e. φ has no free variable,
- ... n–place relation sign, if every free variable in φ is of type 1 and $|\mathbf{FV}(\varphi)| = n$,
- ... class sign, if every free variable in φ is of type 1 and $|\mathbf{FV}(\varphi)| = 1$.

The notion of relation signs is not to be confused with that of relation symbols given by $\mathcal{R}$ ($= \emptyset$ in $\mathcal{L}_P$). Furthermore, we will use the familiar notation $\varphi\,\{\xi_n \mapsto \sigma\}$ to indicate a substitution of free instances of the variable ξ_n in φ by σ. To conform with type theory, we demand that σ is a sign of type n. Moreover, we call a formula $\tilde{\varphi}$ a type elevation of the formula φ, if $\tilde{\varphi}$ emerges from φ by increasing the type of every variable in φ by the same value $s \in \mathbb{N}$. For instance, in the following equation, $\tilde{\varphi}$ is a type elevation of φ:

$$\varphi(x_4, x_3) = x_4(x_3) \vee \exists x_2\ \forall x_1\ x_2(x_1) \quad \rightsquigarrow \quad \tilde{\varphi}(x_6, x_5) = x_6(x_5) \vee \exists x_4\ \forall x_3\ x_4(x_3)\,.$$

The equality sign is not introduced as a basic sign of the language but can rather be defined as in Equation (10). Using type elevation, this yields a definition for equality between signs of type n, for every $n \in \mathbb{N}$.

The system P is given by defining the axiomatic system $\mathcal{A}$, the logical axioms $\mathcal{A}_{\mathrm{Lo}}$ and the rules of inference $\mathfrak{I}$. As noted above, we will use the set of Peano axioms as an axiomatic system:

Definition 2.1.4. (Definition of $\mathcal{A}$) The second-order formulation of Peano arithmetic is given by axioms **Q**1-**Q**6 in Example 1.3.8 (i.e. all except the definition of <) and a second-order version of the induction principle:

$$\forall x_2 \left[\left[x_2(0) \wedge \forall x_1 \left(x_2(x_1) \rightarrow x_2(\mathrm{S}(x_1))\right)\right] \rightarrow \forall x_1\ x_2(x_1)\right].$$

One can show that this choice of axioms in the language of P admits only models that are isomorphic to $\mathbb{N}$, see [Hof17, Section 2.2.3].

Definition 2.1.5. (Logical axioms of P) The following axioms are included:

- Axiom of extensionality: $\forall x_2\ \forall y_2 \left[\forall x_1\ (x_2(x_1) \leftrightarrow y_2(x_1)) \rightarrow x_2 = y_2\right]$ and any of its type elevations,
- Comprehension axiom: $\exists \xi_{n+1}\ \forall \zeta_n\ (\xi_{n+1}(\zeta_n) \leftrightarrow \varphi)$, where ξ_{n+1} is not free in φ.

Furthermore, we will add the same axioms as in Definition 1.4.2 with some adjustments since P is higher-order:

- Axiom **L**4 and **L**10 are omitted,
- Axiom **L**6 is replaced with the above mentioned substitution,
- Axiom **L**8 is omitted, since it (and all its type elevations) can be proved using the definition of equality,
- Axiom **L**9 can be omitted, since it (and all its type elevations) can be proved using the definition of equality and the axiom of comprehension.

Finally, the inference rules are defined as in $\mathcal{I}^{\mathrm{H}}$, see Definition 1.4.1, although we extend the generalization rule **G** to variables of any type.

Next, Gödel introduces a system that assigns a natural number to each formula and to each proof.

Definition 2.1.6. (Assignment of primitive signs) We will assign a natural number to every primitive sign:

"0"	$\rightsquigarrow$	1	"S"	$\rightsquigarrow$	3	"$\neg$"	$\rightsquigarrow$	5	"$\vee$"	$\rightsquigarrow$	7	"$\forall$"	$\rightsquigarrow$	9
"("	$\rightsquigarrow$	11	")"	$\rightsquigarrow$	13	"+"	$\rightsquigarrow$	15	"$\times$"	$\rightsquigarrow$	17	"x_n"	$\rightsquigarrow$	p_8^n
"y_n"	$\rightsquigarrow$	p_9^n	"z_n"	$\rightsquigarrow$	p_{10}^n	$\dots$								

Here, p_i describes the i-th prime number ($p_8, p_9, p_{10}, \dots$ are the prime numbers > 17). Let $\Phi\colon \mathcal{L}_{\mathrm{P}} \to \mathbb{N}$ be the function that maps primitive signs to their corresponding natural number.

Definition 2.1.7. (Gödel numbering) Let $\phi = \sigma_1, \dots, \sigma_k$ be a finite sequence (string) of primitive signs. We define:

$$\ulcorner\phi\urcorner = \ulcorner\sigma_1, \dots, \sigma_k\urcorner := 2^{\Phi(\sigma_1)} \cdot 3^{\Phi(\sigma_2)} \cdot 5^{\Phi(\sigma_3)} \cdot \dots \cdot p_k^{\Phi(\sigma_k)}.$$

We call $\ulcorner\phi\urcorner$ the *Gödel number* of ϕ. Moreover, if $\phi_1, \dots, \phi_k$ is a finite list of strings of primitive signs, we define:

$$\ulcorner\phi_1, \dots, \phi_k\urcorner := 2^{\ulcorner\phi_1\urcorner} \cdot 3^{\ulcorner\phi_2\urcorner} \cdot 5^{\ulcorner\phi_3\urcorner} \cdot \dots \cdot p_k^{\ulcorner\phi_k\urcorner},$$

and again call $\ulcorner\phi_1, \dots, \phi_k\urcorner$ the *Gödel number* of $\phi_1, \dots, \phi_k$.

Note that $\ulcorner\bullet\urcorner$ is injective, since the exponents in the prime factorization of $\ulcorner\phi\urcorner$ are all odd, while they are all even in the factorization of $\ulcorner\phi_1, \dots, \phi_k\urcorner$.

Instead of saying $\varphi_1, \dots, \varphi_k$ stand in some (metamathematical) relation R, we can say that the natural numbers $\ulcorner\varphi_1\urcorner, \dots, \ulcorner\varphi_k\urcorner$ stand in some relation R', i.e. $(\ulcorner\varphi_1\urcorner, \dots, \ulcorner\varphi_k\urcorner) \in R' \subseteq \mathbb{N}^k$. Similarly for primitive signs or lists of strings. Since our system P is able to make statements about the natural numbers, we are able to express metamathematical statements about P within P. Whenever we describe relations of natural numbers by interpreting them as the Gödel numbers of metamathematical objects (if possible), we will write in `CAPITAL LETTERS`. For example, instead of saying that n and k are natural numbers such that there is a proof of some formula φ in P where this proof has Gödel number n and φ has Gödel number k, we will say that n is a `PROOF` of k.

One might wonder about the detailed exploration of Gödel's system P that we have presented. While it is true that our upcoming discussion will not need many of the definitions outlined here, they were necessary in the original proof. One of the core theorems of his work is based on these definitions. In addition, Gödel's detailed introduction has historical significance. It shows the system Gödel used to illustrate his theorem and to convince his contemporary mathematicians – many of whom thought a complete mathematical theory was possible. Although we will take a slightly different approach in the following section, which does not require this detailed introduction, we thought it important to appreciate the historical context of this seminal work.

2.2 Primitive Recursive Functions

We shall leave the system P for the moment to introduce a subset of (computable) functions:

Definition 2.2.1. (Primitive recursive functions) The following functions $f\colon \mathbb{N}^n \to \mathbb{N}$ are called primitive recursive:

PR1. for $n = 1$ and $c \in \mathbb{N}$, the constant functions $f_c(x) = c$,

PR2. for every $i \leq n$, the projection functions $\pi_i(x_1, \dots, x_n) = x_i$,

PR3. for $n = 1$, the successor function $\mathrm{S}(x) = x + 1$,

PR4. for primitive recursive functions $h\colon \mathbb{N}^m \to \mathbb{N}$ and $g_1, \dots, g_m\colon \mathbb{N}^n \to \mathbb{N}$, the composition:

$$h(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$$

PR5. for primitive recursive functions $h\colon \mathbb{N}^{n+1} \to \mathbb{N}$ and $g\colon \mathbb{N}^{n-1} \to \mathbb{N}$, the function $f\colon \mathbb{N}^n \to \mathbb{N}$, defined by:

$$f(0, x_2, \dots, x_n) = g(x_2, \dots, x_n)$$

$$f(k+1, x_2, \dots, x_n) = h(k, f(k, x_2, \dots, x_n), x_2, \dots, x_n)$$

Definition 2.2.2. The property of a function being primitive recursive can be defined as being the last member $f_r = f$ in a finite sequence $f_1, \dots, f_r$ of functions, where each function f_i is either ...

- ... of the form **PR**1, **PR**2 or **PR**3 or
- ... the result of applying **PR**4 or **PR**5 to functions f_j with $j < i$.

We call the length of the shortest possible such sequence the *degree* of f.

Let f be as in **PR**5 for some primitive recursive functions $h\colon \mathbb{N}^{n+1} \to \mathbb{N}$ and $g\colon \mathbb{N}^{n-1} \to \mathbb{N}$. Calculating $f(k, x_1, \dots, x_n)$ can be done using a for loop like in Algorithm 1. We will use Python syntax for these

```
def $f(k, x_2, \dots, x_n)$:
    result = $g(x_2, \dots, x_n)$
    for $i$ in range($k$):
        result = $h(i, \text{result}, x_2, \dots, x_n)$
    return result
```

Algorithm 1: Primitive recursive schema as a for-loop

algorithms. Thus, the for loop ranges from $i = 0$ to $i = k - 1$. If we assume g to be constructed using **PR**5 as well, we can unfold it to Algorithm 2. Furthermore, if we assume h to be constructed using **PR**5, we can unfold it to Algorithm 3. We are seemingly restricted in the design of these for-loops since the schema specifies exactly what values i has to range over and which variables may be used inside the loop. For example, according to the schema, in Algorithm 1, i must range over $0, \dots, k-1$ and the functions g and h must not depend on k. However, this is not limiting, as a short observation shows: First, let $f(x_1, \dots, x_n)$ be a primitive recursive function and let ω: $\{1, \dots, n\} \to \{1, \dots, n\}$ be a permutation. Then, according to **PR**4, the function

$$f(x_{\omega(1)}, \dots, x_{\omega(n)}) = f(\pi_{\omega(1)}(x_1, \dots, x_n), \dots, \pi_{\omega(n)}(x_1, \dots, x_n))$$

is primitive recursive. Thus, by applying an appropriate permutation afterwards, we can assume that looping over any one of the input variables will result in a primitive recursive function. Next, note that instead of looping

```
def $f(k, x_2, \dots, x_n)$:
    result = $g_g(x_3, \dots, x_n)$
    for $i$ in range($x_2$):
        result = $h_g(i, \text{result}, x_3, \dots, x_n)$
    for $i$ in range($k$):
        result = $h(i, \text{result}, x_2, \dots, x_n)$
    return result
```

Algorithm 2: Lists of for-loops are primitive recursive

```
def $f(k, x_2, \dots, x_n)$:
    result = $g_g(x_3, \dots, x_n)$
    for $i$ in range($x_2$):
        result = $h_g(i, \text{result}, x_3, \dots, x_n)$
    for $i$ in range($k$):
        temp = $g_h(\text{result}, x_2, \dots, x_n)$
        for $j$ in range($i$):
            temp = $h_h(j, \text{temp}, \text{result}, x_2, \dots, x_n)$
        result = temp
    return result
```

Algorithm 3: Nesting of for-loops is primitive recursive

over $i = 0, \dots, k-1$, we can use $i = 0, \dots, m(k)-1$ if m is primitive recursive, by defining $f \circ (m \circ \pi_1, \pi_2, \dots, \pi_n)$. Finally, we can apply a simple trick that allows us to pass down variables in a for loop: For example, consider f as in Algorithm 1. We can allow g and h to depend on k by first defining $\tilde{f}$ in Algorithm 4 and then f as in Equation (12).

```
def $\tilde{f}(k, x_1, x_2, \dots, x_n)$:
    result = $g(x_1, \dots, x_n)$
    for $i$ in range($k$):
        result = $h(i, \text{result}, x_1, \dots, x_n)$
    return result
```

Algorithm 4: Passing down variables

$$f(k, x_2, \dots, x_n) := (\tilde{f} \circ (\pi_1, \pi_1, \pi_2, \dots, \pi_n))\,(k, x_2, \dots, x_n) = \tilde{f}(k, k, x_2, \dots, x_n) \tag{12}$$

We have just seen that functions defined by some code consisting of a list of nested for-loops using only primitive recursive functions are primitive recursive. Furthermore the process of first calculating r such lists of nested for-loop functions and then evaluating their result in a primitive recursive function yields a primitive recursive function. We will now show that we can also use *if-else* blocks, as well as *break* statements inside a loop and that we are allowed to define additional variables. To do so, we first prove that certain functions are primitive recursive:

Lemma 2.2.3. *Addition, saturated subtraction (monus), multiplication and exponentiation is primitive recursive:*

$$+\,(k,n) := k+n \qquad \dot{-}\,(k,n) := \begin{cases} k-n, & \text{for } k > n \\ 0, & \text{else} \end{cases} \qquad \bullet\,(k,n) := k \cdot n \qquad \exp(k,n) := k^n$$

Proof. See Algorithms 5, 6, 7, 8. Note that they will also work in edge cases (i.e. if $k = 0$, $n = 0$ or both). □

```
def $+(k,n)$:
    result = $n$
    for $i$ in range($k$):
        result = S(result)
    return result
```
Algorithm 5: Addition

```
def $\dot{-}(k,n)$:
    result = $k$
    for $i$ in range($n$):
        temp = 0
        for $j$ in range(result):
            temp = $j$
        result = temp
    return result
```
Algorithm 6: Saturated subtraction

```
def $\bullet(k,n)$:
    result = 0
    for $i$ in range($k$):
        result = $+(\text{result}, n)$
    return result
```
Algorithm 7: Multiplication

Definition 2.2.4. (Primitive recursive relations) Recall that an n-ary relation on $\mathbb{N}$ is a subset of $\mathbb{N}^n$. We call an n-ary relation R on $\mathbb{N}$ *primitive recursive*, if there exists a primitive recursive function $f_R\colon \mathbb{N}^n \to \mathbb{N}$ such that:

$$(x_1, \dots, x_n) \in R \iff f_R(x_1, \dots, x_n) = 0$$

Corollary 2.2.5. *The relations $\leq, \geq$ are primitive recursive. Furthermore, if R and S are primitive recursive n-ary relations, so are $R \cap S$, $R \cup S$ and $R^c := \mathbb{N}^n \setminus R$.*

Proof. Set $f_{\leq}(k,n) := \dot{-}(k,n)$ and $f_{\geq}(k,n) := \dot{-}(\pi_2(k,n), \pi_1(k,n))$. Let R, S be two primitive recursive n-ary relations and $f_R, f_S\colon \mathbb{N}^n \to \mathbb{N}$ the corresponding (primitive recursive) functions. We define:

$$f_{R\cap S} := +(f_R, f_S), \qquad f_{R\cup S} := \bullet(f_R, f_S), \qquad f_{R^c} := \leq(1, f_R).$$

These functions are primitive. It is easy to verify that they satisfy the desired relations. $\square$

Remark. Note that "∩" usually corresponds to "•" and "∪" to "+". In this case they are reversed.

Because of Corollary 2.2.5, $< := (\geq)^c$, $> := (\leq)^c$ and $= := (\leq) \cap (\geq)$ are primitive recursive relations. We can now introduce a way to implement *if-else* statements in our primitive recursive code.

```
def exp(k, n):
    result = 1
    for i in range(n):
        result = $\bullet$(result, $k$)
    return result
```
Algorithm 8: Exponentiation

Lemma 2.2.6. *Let R be a primitive recursive n-ary relation and $f_1, f_2\colon \mathbb{N}^n \to \mathbb{N}$ two primitive recursive functions. There exists a primitive recursive function $f\colon \mathbb{N}^n \to \mathbb{N}$ such that:*

$$f(x_1, \dots, x_n) = \begin{cases} f_1(x_1, \dots, x_n), & \textit{if } (x_1, \dots, x_n) \in R \\ f_2(x_1, \dots, x_n), & \textit{if } (x_1, \dots, x_n) \notin R \end{cases}$$

Proof. Let R, f_1 and f_2 be as described. The function:

$$f := +\Big(\bullet\big(\dot{-}(1, f_R), f_1\big), \bullet\big(\dot{-}(1, \dot{-}(1, f_R)), f_2\big)\Big) = (1 \dot{-} f_R)\cdot f_1 + (1 \dot{-} (1 \dot{-} f_R))\cdot f_2$$

is primitive recursive and satisfies the desired property. □

Let R be a primitive recursive relation and $h_1, h_2\colon \mathbb{N}^{n+1} \to \mathbb{N}$ two primitive recursive functions. Let $\tilde{h}$ be as in Lemma 2.2.6. If we replace h in Algorithm 1 with $\tilde{h}$, we get a function represented by Algorithm 9. The same

```
def $f(k, x_2, \dots, x_n)$:
    result = $g(x_2, \dots, x_n)$
    for i in range(k):
        if $(i, \text{result}, x_2, \dots, x_n) \in R$:
            result = $h_1(i, \text{result}, x_2, \dots, x_n)$
        else:
            result = $h_2(i, \text{result}, x_2, \dots, x_n)$
    return result
```
Algorithm 9: If-else implementation

argument holds, of course, whenever we apply a primitive recursive function in code.

Next, we will introduce a way to implement variables in our language. To do so, we use an auxiliary function:

Lemma 2.2.7. *(Cantor pairing function) The so called* Cantor pairing function *is bijective:*

$$C\colon \mathbb{N}^2 \to \mathbb{N} \quad , \quad (k, n) \mapsto k + \frac{(n+k)\cdot(n+k+1)}{2} = k + \sum_{i=1}^{n+k} i$$

Proof. For a proof, see e.g. [Wei87, Section 1.2]. □

Lemma 2.2.8. *The functions C, $C_1^{-1} := \pi_1 \circ C^{-1}$ and $C_2^{-1} := \pi_2 \circ C^{-1}$ are primitive recursive.*

Proof. The function C is obviously primitive recursive, see Algorithm 10. Moreover, Algorithm 11 proves that

$$D(c, s) := \begin{cases} C_1^{-1}(c), & \text{for } s = 1 \\ C_2^{-1}(c), & \text{for } s \neq 1 \end{cases}$$

and thus $C_1^{-1} = D(\bullet, 1)$ and $C_2^{-1} = D(\bullet, 2)$ are primitive recursive. Note that $C(r, s) > c$, if $r > c$ or $s > c$ and that $C(r, s)$ is bijective. Thus, the if condition is satisfied exactly once in the for-loop. □

```
def C(k, n):
    result = 0
    for i in range(n + k + 2):
        if i < n + k + 1:
            result = result + i
        else:
            result = result + k
    return result
```

Algorithm 10: Cantor pairing

```
def D(c, s):
    result = 0
    for i in range(c + 1):
        for j in range(c + 1):
            if c == C(i, j):
                if s == 1:
                    result = i
                else:
                    result = j
            else:
                result = result
    return result
```

Algorithm 11: Inverse of Cantor pairing

Now, it is easy to define additional variables in our code. We use this technique to prove the following result:

Lemma 2.2.9. *The* break *command for* for-loops *can be used in our code.*

Proof. Pseudocodes 12, 13 and 14 will yield equivalent results. □

We have seen that code that uses lists of nested for-loops with if-else and break blocks is primitively recursive. With this tool in mind, it is easy to show that a function is primitive recursive.

Example 2.2.10. The relation Prime := $\{n \in \mathbb{N} \mid n \text{ is prime}\}$ is primitive recursive. The function $\Pi(\mathrm{k}) := \mathrm{p_k}$, where p_k is the k-th prime number for $k > 0$ and $p_0 := 1$, is also primitive recursive.

Proof. Bertrand's postulate states that for all $n > 1$, there exists a prime number p such that $n < p < 2n$. For a proof, see [Ram19]. Thus, we know that for any $k \in \mathbb{N}$, there are prime numbers $q_1, \ldots, q_k$, where

$$2 < q_1 < 2^2 < q_2 < 2^3 < \cdots < q_k < 2^{k+1}.$$

In particular, this implies $\Pi(k) < 2^{k+1}$. Algorithm 15 proves that the relation Prime, and Algorithm 16 that the function Π is primitive recursive. □

```
def break_example($k, x_2, \ldots, x_n$):
    result = $C(1, g(x_2, \ldots, x_n))$
    for $i$ in range($k$):
        if $f_R\big(i, C_2^{-1}(\text{result}), x_2, \ldots, x_n\big) \cdot C_1^{-1}(\text{result}) == 0$:
            result = $C\big(0, C_2^{-1}(\text{result})\big)$
        else:
            result = $C\big(C_1^{-1}(\text{result}), h(i, C_2^{-1}(\text{result}), x_2, \ldots, x_n)\big)$
    return $C_2^{-1}(\text{result})$
```

Algorithm 12: Break function in primitive recursive form

```
def break_example2($k, x_2, \ldots, x_n$):
    result = $g(x_2, \ldots, x_n)$
    temp = 1
    for $i$ in range($k$):
        if $R(i, \text{result}, x_2, \ldots, x_n)$ or temp == 0:
            temp = 0
            result = result
        else:
            temp = temp
            result = $h(i, \text{result}, x_2, \ldots, x_n)$
    return result
```

Algorithm 13: Break function using variables

```
def break_example3($k, x_2, \ldots, x_n$):
    result = $g(x_2, \ldots, x_n)$
    for $i$ in range($k$):
        if $R(i, \text{result}, x_2, \ldots, x_n)$:
            break
        result = $h(i, \text{result}, x_2, \ldots, x_n)$
    return result
```

Algorithm 14: Break function

```
def $f_{\texttt{Prime}}$($k$):
    if $k < 2$:
        result = 1
    else:
        result = 0
    for $i$ in range($k$):
        for $j$ in range($k$):
            if $i \cdot j == k$:
                result = result + 1
    return result
```

Algorithm 15: Prime relation

```
def $\Pi(k)$:
    result = 0
    counter = 1
    for $i$ in range($2^{k+1}$):
        result = $i$
        if $i \in$ Prime  and  counter < $k$:
            counter = counter + 1
        else:
            if $i \in$ Prime:
                break
    return result
```

Algorithm 16: k-th prime number

Note that Bertrand's postulate is a non-trivial number-theoretic statement. Thus, proving that Π is primitive recursive is not an obvious result.

Example 2.2.11. Let $p_1, p_2, p_3, \dots$ be the sequence of prime numbers in $\mathbb{N}$. The following functions are primitive recursive:

1. $\texttt{prime-exp}\,(n,k) := \begin{cases} e_k, & \text{if } n,k > 0 \text{ and where } n = \Pi_{i=1}^{\infty} p_i^{e_i} \\ 0, & \text{else} \end{cases}$

2. $\texttt{length}\,(n) := \begin{cases} k, & \text{if } n > 1 \text{ and where } n = \Pi_{i=1}^{k} p_i^{e_i} \quad \text{for} \quad e_k \neq 0 \\ 0, & \text{if } n \leq 1 \end{cases}$

3. $\texttt{var-type}\,(n) := \begin{cases} k, & \text{if } n = p^k \text{ for a prime number } p > 17 \\ 0, & \text{else} \end{cases}$

Thus, for a sequence of primitive signs $\phi = \sigma_1, \dots, \sigma_r$ and a variable x_n of type n, we have for all $i = 1, \dots, r$:

$$\texttt{prime-exp}\,(\ulcorner\phi\urcorner, i) = \Phi(\sigma_i), \qquad \texttt{length}\,(\ulcorner\phi\urcorner) = r, \qquad \texttt{var-type}\,(\Phi(x_n)) = n$$

Proof. For `prime-exp` see code 17. Note that $p^{n-1} \geq n$ for all prime numbers p and natural numbers $n \geq 1$. For `length` see code 18. Note that $p_n \geq n$ for all n. For `var-type` see code 19. □

```
def prime-exp($n, k$):
    result = 0
    $p = \Pi(k)$
    for $i$ in range($n$):
        if $k == 0$:
            break
        for $j$ in range($n + 1$):
            if $j \cdot p^i == n$:
                result  = $i$
    return result
```

Algorithm 17: Get the k-th prime exponent

```
def length($n$):
    result = 0
    for $i$ in range($n + 1$):
        if prime-exp($(n, i)$) $!= 0$:
            result = $i$
    return result
```

Algorithm 18: Get the length of a string

```
def type($n$):
    result = 0
    for $i$ in range($n$):
        for $j$ in range($n$):
            if $\Pi(i + 7)^{j} == n$:
                result = $j$
    return result
```

Algorithm 19: Get the type of a variable

At this point we end our study of primitive recursive functions. We have seen that they can be represented as a function in a programming language with the restriction of only allowing for-loops. Well aware of the expressiveness of these functions, we are not surprised that there is a primitive recursive relation $\mathrm{B} \subseteq \mathbb{N}^2$ consisting of all tuples (n, k) such that n is a PROOF of k. The reader can easily imagine how such an algorithm would be constructed using the tools provided in this section. First, the algorithm would have to verify that k is a FORMULA. To do so, it should check whether k is a STRING and whether it is COMPOSED of ATOMIC FORMULAS according to Definition 1.8.2. Furthermore, the algorithm should check whether these atomic formulas conform to the syntax in Definition 2.1.3. The functions from the last example give a first indication of how to achieve this. They can resolve, for example, the LENGTH and INDIVIDUAL CHARACTERS of the STRING k. In a similar vein, the algorithm should verify that n is a PROOF and that its LAST FORMULA is the FORMULA k. We will not provide a concrete algorithm, since the construction does not cause any difficulties and the modern reader immediately believes this result. Gödel, on the other hand, could not base his reasoning on the intuition of his contemporaries in 1931. The concept of a (theoretical) calculating machine and the related idea of algorithms was introduced only later by Alan Turing, see Chapter 3. Thus, in his original proof, we find a long technical section in which Gödel constructs the relation B from scratch, using Definition 2.2.1, see [Göd31].

Theorem 2.2.12. *The relation $B := \{(n, k) \mid n \text{ is a PROOF of the FORMULA } k\} \subseteq \mathbb{N}^2$ is primitive recursive. We will also denote this relation by:*

$$n\,B\,k :\Longleftrightarrow (n, k) \in B$$

Based on this relation, we can define the relation of *provability* (regarding FORMULAS n):

$$\text{Bew }(n) :\Longleftrightarrow n \in \text{Bew}, \quad \text{Bew} := \{n \in \mathbb{N} \mid \exists\, r : r \text{ B } n\}$$

The relation Bew is not primitive recursive. Recall that $\bar{k}$ is an abbreviation of the term $\underbrace{\mathrm{S}(\ldots\mathrm{S}(k)\ldots)}_{k\text{-times S}}$. We define the following primitive recursive functions:

- $\mathrm{Z}(k) := \ulcorner \bar{k} \urcorner$ for all natural numbers k,
- $\mathrm{Sb}\,(\ulcorner\varphi\urcorner, \Phi(\xi_{i_1 1}), \ulcorner t_{i_1 1}\urcorner, \ldots, \Phi(\xi_{i_n n}), \ulcorner t_{i_n n}\urcorner) := \ulcorner\varphi(\xi_{i_1 1}, \ldots, \xi_{i_n n} \mapsto t_{i_1 1}, \ldots, t_{i_n n})\urcorner$ for every formula φ, variables $\xi_{i_1 1}, \ldots, \xi_{i_n n}$ and signs $t_{i_1 1}, \ldots, t_{i_n n}$ of types i_j,

- $\mathrm{Neg}(\ulcorner\varphi\urcorner) := \ulcorner\neg\varphi\urcorner$ for every formula φ,
- $\mathrm{Gen}(\Phi(\xi), \ulcorner\varphi\urcorner) := \ulcorner\forall\xi\,\varphi\urcorner$ for every formula φ and variable ξ.

The following theorem bridges our knowledge of primitive recursive relations and the system P from Section 2.1.

Theorem 2.2.13. *Let $R \subseteq \mathbb{N}^n$ be an n-ary relation on the natural numbers. If R is primitive recursive, there exists an n-ary relation sign φ in P with free variables $\alpha_1, \ldots, \alpha_n$ such that:*

$$(k_1, k_2, \ldots, k_n) \in R \implies \mathit{Bew}\left(\mathit{Sb}\left(\ulcorner\varphi\urcorner, \Phi(\alpha_1), \ulcorner\overline{k_1}\urcorner, \ldots, \Phi(\alpha_n), \ulcorner\overline{k_n}\urcorner\right)\right) \iff P \vdash \varphi\left(\overline{k_1}, \ldots, \overline{k_n}\right)$$

$$(k_1, k_2, \ldots, k_n) \notin R \implies \mathit{Bew}\left(\mathit{Neg}\left(\mathit{Sb}\left(\ulcorner\varphi\urcorner, \Phi(\alpha_1), \ulcorner\overline{k_1}\urcorner, \ldots, \Phi(\alpha_n), \ulcorner\overline{k_n}\urcorner\right)\right)\right) \iff P \vdash \neg\varphi\left(\overline{k_1}, \ldots, \overline{k_n}\right)$$

Proof. The theorem states that P is able to simulate the code corresponding to R, i.e. prove for a specific choice of $\overline{k_1}, \ldots, \overline{k_n}$ what the evaluation in f_R will be. The formal proof is rather technical and we will only state the proof idea: The theorem follows if we can show for any n-ary primitive recursive function f:

$$y = f(k_1, \ldots, k_n) \implies P \vdash \bar{y} = f\left(\overline{k_1}, \ldots, \overline{k_n}\right) \quad \text{and} \quad y \neq f(k_1, \ldots, k_n) \implies P \vdash \bar{y} \neq f\left(\overline{k_1}, \ldots, \overline{k_n}\right) \qquad (13)$$

It is important to note that Equation (13) is not formally correct, as the right-hand sides of $\vdash$ are not formulas according to Definition 2.1.2. It is to be interpreted as a notation that refers to a formula φ that expresses this (in)equality in terms of the formal syntax of P. We prove by induction over the degree of f. For $r = 1$, f is either a constant function, a projection or $f = \mathrm{S}$. In all cases it is trivial that $f\left(\overline{k_1}, \ldots, \overline{k_n}\right)$ is a term and that Equation (13) holds. Assume the statement has been shown for all $j < r$ and let $f = f_r$ be given by the sequence $f_1, \ldots, f_r$. By the induction hypothesis, Equation (13) holds for all $f_1, \ldots, f_{r-1}$ and since $r > 1$, f is the result of applying **PR**4 or **PR**5 to functions $f_{i_1}, \ldots, f_{i_m}$. One has to verify that in both cases the resulting statement can be formally represented in P. It is then possible to show that P will be able to prove the corresponding result. Thus, Equation 13 holds for $f_r = f$, which concludes the proof. For a formal proof of this theorem, see [HB34] and [HB39]. In addition, we will later improve this result and give insight into how to construct these formulas. □

When expressed in terms of natural numbers, Theorem 2.2.13 states: For any primitive recursive relation $R \subseteq \mathbb{N}^n$, there exists an n-`ARY RELATION SIGN` $r \in \mathbb{N}$ with `FREE VARIABLES` $19, 23, \ldots, p_{n+7}$ such that:

$$(k_1, \ldots, k_n) \in R \implies \mathrm{Bew}\left(\mathrm{Sb}\left(r, 19, Z(k_1), \ldots, p_{n+7}, Z(k_n)\right)\right)$$

$$(k_1, \ldots, k_n) \notin R \implies \mathrm{Bew}\left(\mathrm{Neg}\left(\mathrm{Sb}\left(r, 19, Z(k_1), \ldots, p_{n+7}, Z(k_n)\right)\right)\right)$$

When a `RELATION SIGN` r is assigned a primitive recursive relation as above, we call r *primitive recursive*.

2.3 The First Incompleteness Theorem for Extensions of P

Let $\chi \subseteq \mathbb{N}$ be a set of `FORMULAS` in P. We refer to $P \cup \chi$ as the theory that arises from P by extending its axioms:

$$\mathcal{A}_{P \cup \chi} := \{\varphi \mid \varphi \in \mathcal{A}_P \text{ or } \ulcorner\varphi\urcorner \in \chi\}.$$

Furthermore, we define $\mathrm{Flg}(\chi)$ (for "Folgerungen") to be the set of `THEOREMS` of $P \cup \chi$:

$$\mathrm{Flg}(\chi) := \{\ulcorner\varphi\urcorner \mid P \cup \chi \vdash \varphi\}.$$

Definition 2.3.1. $P \cup \chi$ is called ω-consistent if there is no `CLASS SIGN` a with `FREE VARIABLE` v such that:

$$\forall n \left[\mathrm{Sb}(a, v, Z(n)) \in \mathrm{Flg}(\chi)\right] \quad \text{and} \quad \mathrm{Neg}(\mathrm{Gen}(v, a)) \in \mathrm{Flg}(\chi)$$

If we use our standard notation for P, interpret a as a formula φ and v as free variable ξ_1, the equation in Definition 2.3.1 can be read as:

$$\forall n: \quad P \cup \chi \vdash \varphi(\bar{n}) \quad \text{and} \quad P \cup \chi \vdash \neg \forall \xi_1 \, \varphi(\xi_1) \tag{14}$$

Note that the universal quantifier on the left is part of our metalogic, while on the right it is part of $\mathcal{L}_P$. Suppose $P \cup \chi$ is inconsistent as in Definition 1.7.2. Then, by the principle of explosion, any formula is provable, including those in Equation (14) (for any class sign φ). Thus $P \cup \chi$ is ω-inconsistent. The contrapositive of this argument states that an ω-consistent system $P \cup \chi$ is always also consistent. However, the converse does not hold. Using Gödel's second incompleteness theorem, we will later construct a consistent but ω-inconsistent theory, see Example 2.5.4.

Theorem 2.3.2. *(Incompleteness of $P \cup \chi$) Let $\chi \subseteq \mathbb{N}$ be a primitive recursive set of* `FORMULAS`*. If $P \cup \chi$ is ω-consistent, then it is incomplete.*

Proof. Suppose $P \cup \chi$ is ω-consistent. We will show that there exists a primitive recursive class sign φ with free variable x_1 such that

$$P \cup \chi \nvdash \forall x_1 \, \varphi(x_1) \qquad \text{and} \qquad P \cup \chi \nvdash \neg \forall x_1 \, \varphi(x_1) \, .$$

This will be an immediate corollary of Theorem 2.3.3. We can easily translate the results between our standard notation for P and the notation using natural numbers. □

Theorem 2.3.3. *(Incompleteness of $P \cup \chi$, $\mathbb{N}$ notation) Let $\chi \subseteq \mathbb{N}$ be a set of* `FORMULAS`*. If $P \cup \chi$ is ω-consistent and χ primitive recursive, then there exists a primitive recursive* `CLASS SIGN` *r with* `FREE VARIABLE` *v such that*

$$Gen\,(v, r) \notin \operatorname{Flg}(\chi) \qquad \textit{and} \qquad Neg\,(Gen\,(v, r)) \notin \operatorname{Flg}(\chi) \, .$$

Proof. Similar to Theorem 2.2.12, there exists a primitive recursive relation $\mathrm{B}_\chi \subseteq \mathbb{N}^2$ defined by:

$$n \, \mathrm{B}_\chi \, k \; :\Longleftrightarrow \; (n, k) \in \mathrm{B}_\chi \, , \quad \mathrm{B}_\chi := \{(n, k) \mid n \text{ is a } \texttt{PROOF} \text{ in } P \cup \chi \text{ of the } \texttt{FORMULA } k\}$$

Again, we set $\mathrm{Bew}_\chi\,(n) :\Longleftrightarrow n \in \mathrm{Bew}_\chi := \{n \in \mathbb{N} \mid \exists r \; r \, \mathrm{B}_\chi \, n\}$. Since any proof in P is a proof in $P \cup \chi$, we get:

$$\mathrm{Bew} \subseteq \mathrm{Bew}_\chi = \operatorname{Flg}(\chi) \, . \tag{15}$$

Let us define the following relation $Q \subseteq \mathbb{N}^2$:

$$(k_1, k_2) \in Q \;:\Longleftrightarrow\; \Big(k_1, \mathrm{Sb}\,(k_2, 23, Z(k_2))\Big) \notin \mathrm{B}_\chi \, . \tag{16}$$

Since Sb and Z are primitive recursive, so is $\mathbb{N}^2 \setminus Q$ and therefore, by Corollary 2.2.5, Q. Thus, Theorem 2.2.13 assures the existence of some `FORMULA` q with `FREE VARIABLES` 19, 23 such that the following holds:

$$\begin{aligned}
\Big(k_1, \mathrm{Sb}\,(k_2, 23, Z(k_2))\Big) \notin \mathrm{B}_\chi \;&\Longrightarrow\; (k_1, k_2) \in Q \\
&\Longrightarrow\; \mathrm{Bew}\,\Big(\mathrm{Sb}\Big(q, 19, Z(k_1), 23, Z(k_2)\Big)\Big) \\
&\Longrightarrow\; \mathrm{Bew}_\chi\,\Big(\mathrm{Sb}\Big(q, 19, Z(k_1), 23, Z(k_2)\Big)\Big)
\end{aligned} \tag{17}$$

$$\begin{aligned}
\Big(k_1, \mathrm{Sb}\,(k_2, 23, Z(k_2))\Big) \in \mathrm{B}_\chi \;&\Longrightarrow\; (k_1, k_2) \notin Q \\
&\Longrightarrow\; \mathrm{Bew}\,\Big(\mathrm{Neg}\Big(\mathrm{Sb}\,(q, 19, Z(k_1), 23, Z(k_2))\Big)\Big) \\
&\Longrightarrow\; \mathrm{Bew}_\chi\,\Big(\mathrm{Neg}\Big(\mathrm{Sb}\,(q, 19, Z(k_1), 23, Z(k_2))\Big)\Big)
\end{aligned} \tag{18}$$

Next, we define two `FORMULAS` p, r by:

$$p := \mathrm{Gen}(19, q) \in \mathbb{N} \qquad \text{and} \qquad r := \mathrm{Sb}(q, 23, Z(p)) \in \mathbb{N} \,. \tag{19}$$

Let us denote by φ_n the formula corresponding to $n = \ulcorner\varphi_n\urcorner$. Then, the definition of p and q can be read as follows:

$$\varphi_p = \forall x_1\ \varphi_q(x_1, y_1) \qquad \text{and} \qquad \varphi_r = \varphi_q\left\{y_1 \mapsto \overline{\ulcorner\varphi_p\urcorner}\right\} .$$

Note that φ_p is a class sign with the free variable y_1. Thus, given any natural number n, we can substitute y_1 for $\bar{n}$. In particular, we are allowed to choose $n = p = \ulcorner\varphi_p\urcorner$. For all $k_1 \in \mathbb{N}$, Equation (20) and 21 hold:

$$\mathrm{Sb}(p, 23, Z(p)) = \mathrm{Sb}\Big(\mathrm{Gen}(19, q), 23, Z(p)\Big) = \mathrm{Gen}\Big(19, \mathrm{Sb}(q, 23, Z(p))\Big) = \mathrm{Gen}(19, r) \tag{20}$$

$$\mathrm{Sb}\Big(q, 19, Z(k_1), 23, Z(p)\Big) = \mathrm{Sb}\Big(\mathrm{Sb}(q, 23, Z(p)), 19, Z(k_1)\Big) = \mathrm{Sb}\Big(r, 19, Z(k_1)\Big) \tag{21}$$

In Equation (20), we can switch the functions Gen and Sb, since the `SUBSTITUTION` did not affect the `VARIABLE` 19. In Equation (21) we are allowed to split up Sb due to Notation 3 and change the order of substitution since this does not affect its outcome. If we combine Equations (20), (17) and (21), we get:

$$\begin{aligned}\Big(k_1, \mathrm{Gen}(19, r)\Big) \notin \mathrm{B}_\chi \iff \Big(k_1, \mathrm{Sb}(p, 23, Z(p))\Big) \notin \mathrm{B}_\chi \implies &\mathrm{Bew}_\chi\ \Big(\mathrm{Sb}\Big(q, 19, Z(k_1), 23, Z(p)\Big)\Big) \\ \iff &\mathrm{Bew}_\chi\ \Big(\mathrm{Sb}\Big(r, 19, Z(k_1)\Big)\Big)\end{aligned} \tag{22}$$

Applying the same argument to Equations (20), (18) and (21) yields:

$$\Big(k_1, \mathrm{Gen}(19, r)\Big) \in \mathrm{B}_\chi \implies \mathrm{Bew}_\chi\ \Big(\mathrm{Neg}\Big(\mathrm{Sb}(r, 19, Z(k_1))\Big)\Big) \tag{23}$$

Equations (22) and (23) have the following meaning:

$$\begin{aligned} k_1 \text{ does not encode a proof of } \forall x_1\ \varphi_r(x_1) &\implies P \cup \chi \vdash \varphi_r\left(\overline{k_1}\right) \\ k_1 \text{ does encode a proof of } \forall x_1\ \varphi_r(x_1) &\implies P \cup \chi \vdash \neg\varphi_r\left(\overline{k_1}\right)\end{aligned}$$

In this form we can easily see a kind of self-reference of $\forall x_1\ \varphi_r(x_1)$, which helps us to show that neither $\forall x_1\ \varphi_r(x_1)$ nor $\neg\forall x_1\ \varphi_r(x_1)$ are provable. We will demonstrate the corresponding statement in our notation in $\mathbb{N}$:

1. $\underline{\mathrm{Gen}(19, r) \notin \mathrm{Flg}(\chi)}$

   Since $P \cup \chi$ is ω-consistent and thus, in particular, consistent, the following holds for all $m \in \mathbb{N}$:

   $$\neg\,\mathrm{Bew}_\chi\ (\mathrm{Sb}(r, 19, Z(m))) \qquad \text{or} \qquad \neg\,\mathrm{Bew}_\chi\ (\mathrm{Neg}(\mathrm{Sb}(r, 19, Z(m)))) \,.$$

   If there is no `PROOF` (in $P \cup \chi$) for the `FORMULA` $\mathrm{Sb}(r, 19, Z(m))$ for a specific $m \in \mathbb{N}$, then there cannot be a `PROOF` of $\mathrm{Gen}(19, r)$. The contrapositive of this statement can easily be shown using the substitution rule (see **L**6). Furthermore, if we apply the contrapositive of Equation (23) to the right side, we get:

   $$\neg\,\mathrm{Bew}_\chi\ (\mathrm{Gen}(19, r)) \qquad \text{or} \qquad \text{for all } m \in \mathbb{N}\text{: } \Big(m, \mathrm{Gen}(19, r)\Big) \notin \mathrm{B}_\chi \,. \tag{24}$$

   But now the left and the right side are, by definition, each equivalent to $\mathrm{Gen}(19, r) \notin \mathrm{Flg}(\chi)$.

2. $\underline{\mathrm{Neg}\,(\mathrm{Gen}\,(19, r)) \notin \mathrm{Flg}\,(\chi)}$

   Combining Equation (22) and the right side of Equation (24) proves:

$$\text{for all } m \in \mathbb{N}\text{: } \mathrm{Bew}_\chi \left(\mathrm{Sb}\left(r, 19, Z(m)\right)\right) .$$

   On the other hand, the ω-consistency of $P \cup \chi$ requires that the following implication holds:

$$\text{for all } m \in \mathbb{N}\text{: } \left[\mathrm{Sb}\,(r, 19, Z(m)) \in \mathrm{Flg}\,(\chi)\right] \quad \Longrightarrow \quad \mathrm{Neg}\,(\mathrm{Gen}\,(19, r)) \notin \mathrm{Flg}\,(\chi) .$$

   Thus, combining the last two equations proves $\mathrm{Neg}(\,\mathrm{Gen}\,(19, r)) \notin \mathrm{Flg}\,(\chi)$. □

Gödel finished his proof in a different way. He showed that $\mathrm{Gen}\,(19, r)$ is undecidable by assuming the opposite and then deriving a contradiction in the obvious way. However, as we have shown, this is not necessary, since a direct argument can be used.

An immediate corollary of Theorem 2.3.2 is the incompleteness of P (if we assume P to be ω-consistent).

Example 2.3.4. (ω-consistent and complete theory) Let us assume that P is a sound and ω-consistent theory. We denote the standard model of P as $\mathbb{N}$. Define χ by

$$\chi := \{\ulcorner\varphi\urcorner \mid P \models \varphi\} .$$

Note that some structure $\mathcal{M}$ is a model of $P \cup \chi$ if and only if it is a model of P (by definition of χ and the fact that $P \subseteq P \cup \chi$). The second-order version of Peano arithmetic (as found in P) uniquely defines the natural numbers up to isomorphism. As a result, exactly those structures equivalent to $\mathbb{N}$ are models of P and $P \cup \chi$. We will show that $P \cup \chi$ is ω-consistent and complete: First suppose it is ω-inconsistent. Then, there would be a class sign φ such that

$$\forall n: \; P \cup \chi \vdash \varphi(\bar{n}) \quad \text{and} \quad P \cup \chi \vdash \neg\forall x_1 \; \varphi(x_1) .$$

By definition, P is sound if and only if $P \cup \chi$ is sound. Thus, since $\mathbb{N}$ is a model of $P \cup \chi$, we would have:

$$\forall n: \; \mathbb{N} \models \varphi(\bar{n}) \quad \text{and} \quad \mathbb{N} \models \neg\forall x_1 \; \varphi(x_1) .$$

The right side of this equation states that there exists a natural number n such that $\mathbb{N} \models \neg\varphi(\bar{n})$. But this contradicts the left side of the equation, since for this specific n, $\mathbb{N} \models \varphi(\bar{n})$. Thus, our assumption was wrong and $P \cup \chi$ is in fact ω-consistent. Moreover, $P \cup \chi$ is complete, for if it were not, there would be a sentence φ such that

$$P \cup \chi \nvdash \varphi \quad \text{and} \quad P \cup \chi \nvdash \neg\varphi .$$

In particular, φ would not be an axiom of $P \cup \chi$. Then, by the definition of χ, we would have

$$P \not\models \varphi \quad \text{and} \quad P \not\models \neg\varphi .$$

However, since all models of P are isomorphic, this is would yield a contradiction. As claimed, $P \cup \chi$ is ω-consistent and complete. Therefore, by Gödel's first incompleteness theorem, χ cannot be primitive recursive.

Remark. The standard example in the literature for an ω-consistent and complete theory is *true arithmetic*. This first-order system is defined by selecting all true first-order statements about the arithmetic of $\mathbb{N}$ as axioms. However, this example is not applicable to our situation, since we need an extension of P which involves higher-order logic.

So far, we have seen that (under some weak conditions) there exists an undecidable proposition $\forall x_1 \; \varphi(x_1)$ in $P \cup \chi$. A priori, its corresponding semantic interpretation might be very abstract. We will now improve the result of Theorem 2.3.2 by showing that there are undecidable arithmetic formulas, i.e. basic statements about the natural numbers that cannot be proved nor refuted. To do so, we first prove an auxiliary lemma:

Definition 2.3.5. (Gödel's β-function) For any given natural numbers $n, p \in \mathbb{N}$, $p > 0$, there exist uniquely defined natural numbers $k, r \in \mathbb{N}$ such that $n = k \cdot p + r$ and $0 \leq r < p$. We set $[n]_p = r$ and call

$$\beta : \mathbb{N}^3 \to \mathbb{N}, \quad (n, d, k) \mapsto [n]_{1+(k+1)\cdot d}$$

the Gödel β-function.

Lemma 2.3.6. *Let $(f_i)_{i\in\mathbb{N}}$ be a sequence of natural numbers and let $k \in \mathbb{N}$ be a natural number. There are natural numbers n, d such that*

$$\beta(n, d, 0) = f_0\,, \qquad \beta(n, d, 1) = f_1\,, \quad \ldots\,, \quad \beta(n, d, k-1) = f_{k-1}\,.$$

Proof. Let $l := \max\{k, f_0, \ldots, f_{k-1}\}$. We start the proof by showing that the numbers

$$1 + (i+1) \cdot l!\ , \quad \text{for } 0 \leq i < k\,, \tag{25}$$

are pairwise coprime. Suppose not, then there is a prime number p and some $0 \leq i < j < k$ such that

$$p \mid (1 + (i+1) \cdot l!\,) \quad \text{and} \quad p \mid (1 + (j+1) \cdot l!\,)\,,$$

hence,

$$p \mid [(1 + (j+1) \cdot l!\,) - (1 + (i+1) \cdot l!\,)]\,.$$

This can be simplified to $p \mid (j - i) \cdot l!$. Since $0 < j - i \leq j < k \leq l$, it follows that $p \mid l!$ and thus $p \mid (i+1) \cdot l!$. Together with $p \mid (1 + (i+1) \cdot l!\,)$ we can conclude that $p \mid 1$ which is not possible. Therefore, the assumption was wrong and all numbers in Equation (25) are pairwise coprime. Thus, by the Chinese remainder theorem, there exists a natural number $n \in \mathbb{N}$ such that

$$n \equiv f_i \mod (1 + (i+1) \cdot l!\,)\,, \quad \text{for } 0 \leq i < k\,.$$

In other words, there are natural numbers $s_0, \ldots, s_{k-1} \in \mathbb{Z}$ such that

$$n = f_i + s_i \cdot (1 + (i+1) \cdot l!\,)\,, \quad \text{for } 0 \leq i < k\,.$$

Note, that $n, (1 + (i+1) \cdot l!\,) \geq 0$ for all i and $0 \leq f_i \leq l < (1 + (i+1) \cdot l!\,)$. Therefore we must have $s_i \in \mathbb{N}$ and $f_i = [n]_{1+(i+1)\cdot l!} = \beta(n, l!\,, i)$ for all $i = 0, \ldots, k-1$, which was to be demonstrated. □

Definition 2.3.7. We call a relation of natural numbers $R \subseteq \mathbb{N}^n$ arithmetical if it is of the form

$$R = \{(x_1, \ldots, x_n) \in \mathbb{N}^n \mid \Phi_R(x_1, \ldots, x_n)\}\,,$$

for some formula Φ_R (in our meta language) consisting only of ...

- variables, the equal sign and quantifiers (if they solely apply to natural numbers),
- addition and multiplication in $\mathbb{N}$,
- and the logical connectives "$\vee$" and "$\neg$".

Example 2.3.8. The following relations are arithmetical:

$$\begin{aligned}
x < y \quad &:\Longleftrightarrow \quad (x,y) \in \Big\{(a,b) \in \mathbb{N}^2 \mid \forall c \in \mathbb{N} : \neg(a = b + c)\Big\} \\
\text{Prime}(x) \quad &:\Longleftrightarrow \quad x \in \Big\{a \in \mathbb{N} \mid \forall k_1, k_2 \in \mathbb{N} : \big((a = k_1) \vee (a = k_2) \vee (\neg(k_1 \cdot k_2 = a))\big) \wedge (a > 1)\Big\} \\
x \equiv y \mod n \quad &:\Longleftrightarrow \quad (x,y,n) \in \Big\{(a,b,c) \in \mathbb{N}^3 \mid (\exists d \in \mathbb{N} : a = b + d \cdot c) \vee (\exists d \in \mathbb{N} : b = a + d \cdot c)\Big\}
\end{aligned}$$

Theorem 2.3.9. *Every primitive recursive relation is arithmetical.*

Proof. We will prove that for any primitive recursive function $f\colon \mathbb{N}^n \to \mathbb{N}$ there exists an arithmetical relation $R \subseteq \mathbb{N}^{n+1}$ such that $(x_1, \dots, x_n, x_{n+1}) \in R$ if and only if $x_{n+1} = f(x_1, \dots, x_n)$. This implies the theorem since the relation

$$(x_1, \dots, x_n) \in R_f \quad :\Longleftrightarrow \quad 0 = f(x_1, \dots, x_n) \quad \Longleftrightarrow \quad (x_1, \dots, x_n, 0) \in R$$

is arithmetical. Let $f\colon \mathbb{N}^n \to \mathbb{N}$ be primitive recursive. We prove by induction over the degree of f. The base case is trivial: If f is constant, a projection (π_i) or the successor function (S), the formula $x_{n+1} = f(x_1, \dots, x_n)$ is obviously arithmetical. Now, suppose f is of degree $s > 1$ and that the theorem has been shown for all $i < s$. By Definition 2.2.1, f is of the form **PR**4 or **PR**5:

1. $f(x_1, \dots, x_n) = h(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$
2. $f(0, x_2, \dots, x_n) = g(x_2, \dots, x_n)$,
   $f(k+1, x_2, \dots, x_n) = h(k, f(k, x_2, \dots, x_n), x_2, \dots, x_n)$

All functions other than f must be primitive recursive with degree $< s$. In the first case, by the inductive hypothesis, there exist arithmetical and primitive recursive relations U and S_i, $i = 1, \dots, m$, such that:

$$\begin{aligned}
z_{m+1} = h(z_1, \dots, z_m) \quad &:\Longleftrightarrow \quad (z_1, \dots, z_m, z_{m+1}) \in U \quad \Longleftrightarrow \quad \Phi_U(z_1, \dots, z_m, z_{m+1}) \\
x_{n+1} = g_i(x_1, \dots, x_n) \quad &:\Longleftrightarrow \quad (x_1, \dots, x_n, x_{n+1}) \in S_i \quad \Longleftrightarrow \quad \Phi_{S_i}(x_1, \dots, x_n, x_{n+1})
\end{aligned}$$

Let us define:

$$\Phi(x_1, \dots, x_{n+1}) := \exists z_1, \dots, z_m \in \mathbb{N} : \Phi_U(z_1, \dots, z_m, x_{n+1}) \wedge \Phi_{S_1}(x_1, \dots, x_n, z_1) \wedge \dots \wedge \Phi_{S_m}(x_1, \dots, x_n, z_m). \tag{26}$$

The set $\{(x_1, \dots, x_n, x_{n+1}) \in \mathbb{N}^{n+1} \mid \Phi(x_1, \dots, x_{n+1})\}$ then has the desired property.
In the latter case, by the inductive hypothesis, there are arithmetical and primitive recursive relations Q, T such that

$$\begin{aligned}
x_{n+1} = g(x_2, \dots, x_n) \quad &:\Longleftrightarrow \quad (x_2, \dots, x_n, x_{n+1}) \in Q \quad \Longleftrightarrow \quad \Phi_Q(x_2, \dots, x_n, x_{n+1}) \\
x_{n+1} = h(x_1, z, x_2, \dots, x_n) \quad &:\Longleftrightarrow \quad (x_1, z, x_2, \dots, x_n, x_{n+1}) \in T \quad \Longleftrightarrow \quad \Phi_T(x_1, z, x_2, \dots, x_n, x_{n+1})
\end{aligned} \tag{27}$$

We define:

$$\begin{aligned}
\Phi(x_1, \dots, x_{n+1}) := \exists m, d, y \in \mathbb{N} : \Big[y = \beta(m,d,0) \wedge \Phi_Q(x_2, \dots, x_n, y) \wedge \forall k \in \mathbb{N}\ \Big(k < x_1 \implies \exists y_1, y_2 \in \mathbb{N} : \\
(y_1 = \beta(m,d,k+1)) \wedge (y_2 = \beta(m,d,k)) \wedge \Phi_T(k, y_2, x_2, \dots, x_n, y_1)\Big) \wedge x_{n+1} = \beta(m,d,x_1)\Big].
\end{aligned}$$

Let $s_1, \dots, s_{n+1} \in \mathbb{N}$ be arbitrary fixed natural numbers. If $\Phi(s_1, \dots, s_{n+1})$ holds, we know that there are natural numbers $f_0, f_1, \dots, f_{s_1}$ as well as m, d such that:

1. $f_k = \beta(m,d,k)$ for all $k = 0, \dots, s_1$,
2. $f_0 = \beta(m,d,0) = g(s_2, \dots, s_n)$,

3. $f_{k+1} = \beta(m,d,k+1) = h(k, \beta(m,d,k), s_2, \ldots, s_n) = h(k, f_k, s_2, \ldots, s_n)$ for all $k = 0, \ldots, s_1 - 1$,

4. $s_{n+1} = \beta(m,d,s_1) = f_{s_1}$.

Thus, $s_{n+1} = f(s_1, s_2, \ldots, s_n)$. Conversely, if $s_{n+1} = f(s_1, s_2, \ldots, s_n)$ holds, by Lemma 2.3.6 there are natural numbers $m, d \in \mathbb{N}$ such that $\beta(m,d,k) = f(k, s_2, \ldots, s_n)$ for all $k = 0, \ldots, s_1$. By defining $f_i := \beta(m,d,i)$ for $i \leq s_1$, all of the points in the above list and thus $\Phi(s_1, \ldots, s_{n+1})$ hold. Moreover, note that the equation

$$y = \beta(m,d,i) \quad \Longleftrightarrow \quad \big(y \equiv m \mod (1 + (i+1) \cdot d)\big) \land y < (1 + (i+1) \cdot d)$$

can be expressed arithmetically. In other words, the relation $\{(x_1, \ldots, x_n, x_{n+1}) \in \mathbb{N}^{n+1} \mid \Phi(x_1, \ldots, x_{n+1})\}$ is arithmetical and has the desired property. □

Gödel briefly points out that one can repeat this proof in P for any particular primitive recursive relation, and thus concludes the existence of unprovable arithmetic formulas. In order to fully follow his argument, we will give a detailed proof of this implicit step:

Theorem 2.3.10. *In any ω-consistent system $P \cup \chi$ with primitive recursive set $\chi \subseteq \mathbb{N}$, there are undecidable arithmetical propositions.*

Proof. We will show that for any given primitive recursive function $f\colon \mathbb{N}^n \to \mathbb{N}$, there exists an arithmetical formula in P with free variables $\alpha_1, \ldots, \alpha_{n+1}$ such that

$$k_{n+1} = f(k_1, \ldots, k_n) \implies P \vdash \varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right) \quad \text{and} \quad k_{n+1} \neq f(k_1, \ldots, k_n) \implies P \vdash \neg\varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right) \qquad (28)$$

The theorem then follows like in Theorem 2.3.2 using this improved version of Theorem 2.2.13. Let $f \in \mathbb{N}^n$ be primitive recursive. By Theorem 2.3.9 there exists an arithmetical formula Φ such that

$$k_{n+1} = f(k_1, \ldots, k_n) \quad \Longleftrightarrow \quad \Phi(k_1, \ldots, k_{n+1}).$$

Since P can formalize properties of the natural numbers (it contains the axioms of **PA**), there exists a corresponding formula φ in P for which we will prove Equation (28). To do so we will revisit the proof of Theorem 2.3.9. In each step of the induction, we will show that for any fixed $k_1, \ldots, k_{n+1} \in \mathbb{N}$, Equation (28) as well as Equation (29) holds:

$$P \vdash \forall \alpha_{n+1} \left[\neg\left(\alpha_{n+1} = \overline{k_{n+1}}\right) \land \varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right) \to \neg\varphi\left(\overline{k_1}, \ldots, \overline{k_n}, \alpha_{n+1}\right)\right]. \qquad (29)$$

In the base case, φ is of the form $\varphi(\alpha_1, \ldots, \alpha_{n+1}) \equiv (\alpha_{n+1} = \overline{m})$ for a fixed $m \in \mathbb{N}$, $\varphi(\alpha_1, \ldots, \alpha_{n+1}) \equiv (\alpha_{n+1} = \alpha_i)$ for some fixed $0 < i \leq n$, or $\varphi(\alpha_1, \alpha_2) \equiv (\alpha_2 = \mathrm{S}(\alpha_1))$. The reader can easily verify that, for a specific choice of natural numbers $k_1, \ldots, k_{n+1} \in \mathbb{N}$, one can syntactically prove $\varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right)$ if $\Phi(k_1, \ldots, k_{n+1})$ holds and $\neg\varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right)$ if not. Moreover, Equation (29) is trivial for these cases and can easily be proved in P. In Chapter 1 we have studied in detail how one can construct such formal proofs. Next, for the inductive step, suppose that f has degree s for some $s > 1$ and that Equations (28) and (29) have been proved for all functions with corresponding degree $< s$. Let $k_1, \ldots, k_{n+1}$ be a specific choice of natural numbers. The formula in Equation (29) is semantically trivial by construction, since it asserts the right-uniqueness (or functionality) of the relation $R_\Phi \subseteq \mathbb{N}^n \times \mathbb{N}$. It can be proved for both cases (**PR**4 and **PR**5) using the induction hypothesis. However, formalizing this proof in P would require considerable effort, so we omit this step. Regarding Equation (28): If we are in the case of **PR**4 (i.e. Equation (26)), define $g_1, \ldots, g_m \in \mathbb{N}$ by $g_i := g_i(k_1, \ldots, k_n)$. By the induction hypothesis, we have:

$$\begin{aligned}
\Phi(k_1, \ldots, k_{n+1}) &\implies \Phi_U(g_1, \ldots, g_m, k_{n+1}) \wedge \Phi_{S_1}(k_1, \ldots, k_n, g_1) \wedge \ldots \wedge \Phi_{S_m}(k_1, \ldots, k_n, g_m) \\
&\implies P \vdash \varphi_U\left(\overline{g_1}, \ldots, \overline{g_m}, \overline{k_{n+1}}\right) \wedge \varphi_{S_1}\left(\overline{k_1}, \ldots, \overline{k_n}, \overline{g_1}\right) \wedge \ldots \wedge \varphi_{S_m}\left(\overline{k_1}, \ldots, \overline{k_n}, \overline{g_m}\right) \\
&\implies P \vdash \underbrace{\exists z_1, \ldots, z_m\, \varphi_U\left(z_1, \ldots, z_m, \overline{k_{n+1}}\right) \wedge \varphi_{S_1}\left(\overline{k_1}, \ldots, \overline{k_n}, z_1\right) \wedge \ldots \wedge \varphi_{S_m}\left(\overline{k_1}, \ldots, \overline{k_n}, z_m\right)}_{=\varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right)}
\end{aligned}$$

This proves the left side of Equation (28). Next, suppose $k_1, \ldots, k_{n+1}$ are such that $\Phi(k_1, \ldots, k_{n+1})$ does not hold. Let $b = f(k_1, \ldots, k_n)$. By definition, we have $\Phi(k_1, \ldots, k_n, b)$ and $b \neq k_{n+1}$ and therefore:

$$P \vdash \varphi\left(\overline{k_1}, \ldots, \overline{k_n}, \bar{b}\right) \quad \text{and} \quad P \vdash \neg\left(\overline{k_{n+1}} = \bar{b}\right)$$

Moreover, applying the specialization rule (**L**6) to Equation (29) yields:

$$P \vdash \neg\left(\overline{k_{n+1}} = \bar{b}\right) \wedge \varphi\left(\overline{k_1}, \ldots, \overline{k_n}, \bar{b}\right) \to \neg\varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right)$$

Using modus ponens in P then shows $P \vdash \neg\varphi\left(\overline{k_1}, \ldots, \overline{k_{n+1}}\right)$. This was the right side of Equation (28). We can apply the same kind of argument if we are in the case of **PR**5 (Equation (27)). Note that for a fixed k_1, the formula $\varphi\left\{\alpha_1 \mapsto \overline{k_1}\right\}$ can be rewritten to only include existential quantifiers. This concludes our proof. □

It should be noted that the diagonal lemma, often mentioned in connection with Gödel's incompleteness theorem, was not necessary for the proof itself. The famous concept of a proposition asserting its own unprovability is found only in a proof sketch that Gödel included at the beginning of his paper. However, this argument was later used in a more formal way by the American mathematician J. Barkley Rosser to refine the first incompleteness theorem.

2.4 Rosser's Trick

In 1936, Rosser improved Gödel's result by weakening the requirement of Theorem 2.3.10 from ω-consistency to consistency. Rosser used a trick where he defined a new provability relation which is equivalent to Bew_χ if the underlying theory is consistent, but which has a favorable property. In this section we will present his proof. The key ideas of this section can be found, for example, in [Pro21] and [Ros36].

Definition 2.4.1. By Theorem 2.2.13 there exists a formula φ_{B_χ} in P with free variables α_1, α_2 such that

$$(n,k) \in \mathrm{B}_\chi \iff n\, \mathrm{B}_\chi\, k \implies \mathrm{Bew}\left(\mathrm{Sb}\left(\ulcorner\varphi_{B_\chi}\urcorner, \Phi(\alpha_1), \ulcorner\bar{n}\urcorner, \Phi(\alpha_2), \ulcorner\bar{k}\urcorner\right)\right) \iff P \vdash \varphi_{B_\chi}\left(\bar{n}, \bar{k}\right)$$

$$(n,k) \notin \mathrm{B}_\chi \iff \neg\, n\, \mathrm{B}_\chi\, k \implies \mathrm{Bew}\left(\mathrm{Neg}\left(\mathrm{Sb}\left(\ulcorner\varphi_{B_\chi}\urcorner, \Phi(\alpha_1), \ulcorner\bar{n}\urcorner, \Phi(\alpha_2), \ulcorner\bar{k}\urcorner\right)\right)\right) \iff P \vdash \neg\varphi_{B_\chi}\left(\bar{n}, \bar{k}\right)$$

Furthermore, since the relation

$$(n,k) \in B_{\chi N} :\iff n\, \mathrm{B}_\chi\, \mathrm{Neg}(k)$$

is primitive recursive, there is a formula $\varphi_{B_{\chi N}}$ with free variables α_1, α_2 such that:

$$n\, \mathrm{B}_\chi\, \mathrm{Neg}(k) \implies \mathrm{Bew}\left(\mathrm{Sb}\left(\ulcorner\varphi_{B_{\chi N}}\urcorner, \Phi(\alpha_1), \ulcorner\bar{n}\urcorner, \Phi(\alpha_2), \ulcorner\bar{k}\urcorner\right)\right) \iff P \vdash \varphi_{B_{\chi N}}\left(\bar{n}, \bar{k}\right)$$

$$\neg\, n\, \mathrm{B}_\chi\, \mathrm{Neg}(k) \implies \mathrm{Bew}\left(\mathrm{Neg}\left(\mathrm{Sb}\left(\ulcorner\varphi_{B_{\chi N}}\urcorner, \Phi(\alpha_1), \ulcorner\bar{n}\urcorner, \Phi(\alpha_2), \ulcorner\bar{k}\urcorner\right)\right)\right) \iff P \vdash \neg\varphi_{B_{\chi N}}\left(\bar{n}, \bar{k}\right)$$

We define the following formulas in P:

- $\varphi_{B_\chi^R}(\alpha_1, \alpha_2) := \varphi_{B_\chi}(\alpha_1, \alpha_2) \wedge \forall\alpha_3\left((\alpha_3 \leq \alpha_1) \to \neg\varphi_{B_{\chi N}}(\alpha_3, \alpha_2)\right)$

- $\varphi_{B^R_{\chi N}}(\alpha_4, \alpha_2) := \varphi_{B_{\chi N}}(\alpha_4, \alpha_2) \wedge \forall \alpha_5 \left((\alpha_5 \le \alpha_4) \to \neg\varphi_{B_\chi}(\alpha_5, \alpha_2)\right)$
- $\psi(\alpha_2) := \exists\, \alpha_1\; \varphi_{B^R_\chi}(\alpha_1, \alpha_2)$
- $\psi_N(\alpha_2) := \exists\, \alpha_4\; \varphi_{B^R_{\chi N}}(\alpha_4, \alpha_2)$

Let $n, k \in \mathbb{N}$. The formula $\varphi_{B^R_\chi}\left(\bar{n}, \bar{k}\right)$ in some sense states that n is a PROOF of k and that there is no natural number l, smaller than or equal to n, such that l is a PROOF of the NEGATION of k. If we assume $P \cup \chi$ to be consistent, this is of course equivalent to the statement that n is a PROOF of k.

Lemma 2.4.2. *We have the following result:*

$$P \cup \chi \;\vdash\; \psi(\alpha_2) \to \neg\psi_N(\alpha_2)$$

Proof. We will show $P \vdash \neg(\psi(\alpha_2) \wedge \psi_N(\alpha_2))$. The formula $\neg((\alpha_1 > \alpha_4) \wedge (\alpha_1 < \alpha_4))$ as well as the following

- $\left[\varphi_{B_\chi}(\alpha_1, \alpha_2) \wedge \forall \alpha_5 \left((\alpha_5 \le \alpha_4) \to \neg\varphi_{B_\chi}(\alpha_5, \alpha_2)\right)\right] \to \alpha_1 > \alpha_4$
- $\left[\forall \alpha_3 \left((\alpha_3 \le \alpha_1) \to \neg\varphi_{B_{\chi N}}(\alpha_3, \alpha_2)\right) \wedge \varphi_{B_{\chi N}}(\alpha_4, \alpha_2)\right] \to \alpha_1 < \alpha_4$

are obviously provable in P. Thus, P can prove the following formula:

$$\neg\Big[\left[\varphi_{B_\chi}(\alpha_1, \alpha_2) \wedge \forall \alpha_5 \left((\alpha_5 \le \alpha_4) \to \neg\varphi_{B_\chi}(\alpha_5, \alpha_2)\right)\right] \wedge \left[\forall \alpha_3 \left((\alpha_3 \le \alpha_1) \to \neg\varphi_{B_{\chi N}}(\alpha_3, \alpha_2)\right) \wedge \varphi_{B_{\chi N}}(\alpha_4, \alpha_2)\right]\Big]$$

By rearranging terms, one can easily see that the above formula is equivalent to $\neg(\varphi_{B^R_\chi}(\alpha_1, \alpha_2) \wedge \varphi_{B^R_{\chi N}}(\alpha_4, \alpha_2))$. Applying the generalization rule **G** in P for the free variables α_1, α_4, thus yields:

$$P \;\vdash\; \neg\exists\alpha_4\; \exists\alpha_1\; \varphi_{B^R_\chi}(\alpha_1, \alpha_2) \wedge \varphi_{B^R_{\chi N}}(\alpha_4, \alpha_2)$$

Finally, by rearranging terms (see **L**7) and using the definition of ψ, ψ_N, we get $P \vdash \neg(\psi(\alpha_2) \wedge \psi_N(\alpha_2))$. □

We also have the following result.

Lemma 2.4.3. *(Diagonal lemma) Let λ be a class sign. There exists a sentence ρ in $P \cup \chi$ such that*

$$P \cup \chi \vdash \rho \leftrightarrow \lambda\left(\overline{\ulcorner \rho \urcorner}\right).$$

If λ is arithmetical, we can choose ρ to be arithmetical.

Proof. We will use an adapted version of the proof found in [BBJ02]. First, bring every class sign of $P \cup \chi$ with free variable x_1 in order, e.g. by using their Gödel number. We denote γ_k for the k-th class sign with free variable x_1. Let us define the corresponding function f_γ:

$$f_\gamma(k) := \ulcorner \gamma_k \urcorner.$$

One can easily see that the formulas $\varphi_r \equiv \forall \alpha_{r+1}\ (x_1 = \alpha_{r+1})$ are class signs with increasing Gödel numbers $\ulcorner \varphi_1 \urcorner < \ulcorner \varphi_2 \urcorner < \dots$, and thus $\ulcorner \gamma_k \urcorner \le \ulcorner \varphi_k \urcorner$. Note that the function calculating $\ulcorner \varphi_k \urcorner$ for a given k is primitive recursive. As a result, one could easily construct a primitive recursive function, that loops through all natural numbers $\le \ulcorner \varphi_k \urcorner$ and outputs the k-th number corresponding to a class sign with free variable x_1. We can conclude that f_γ and thus $\mathrm{diag}(n)$ is primitive recursive, where $\mathrm{diag}(n)$ is defined as:

$$\mathrm{diag}(n) := \mathrm{Sb}(f_\gamma(n), 19, Z(n)) = \ulcorner \gamma_n\,(\bar{n})\,\urcorner.$$

In the proof of Theorem 2.3.10 we have seen how to construct an arithmetical formula δ with free variables α_1, α_2 such that for all $k_1, k_2 \in \mathbb{N}$:

$$k_2 = \operatorname{diag}(k_1) \implies P \cup \chi \vdash \delta\left(\overline{k_1}, \overline{k_2}\right) \quad \text{and} \quad k_2 \neq \operatorname{diag}(k_1) \implies P \cup \chi \vdash \neg\delta\left(\overline{k_1}, \overline{k_2}\right)$$

$$P \cup \chi \vdash \forall \alpha_2 \left[\neg\left(\alpha_2 = \overline{k_2}\right) \wedge \delta\left(\overline{k_1}, \overline{k_2}\right) \rightarrow \neg\delta\left(\overline{k_1}, \alpha_2\right)\right] \tag{30}$$

Using this formula δ, we define a new class sign ϕ by $\phi(x_1) := \exists \alpha_2\ \delta(x_1, \alpha_2) \wedge \lambda(\alpha_2)$. Let q be the natural number such that $\gamma_q = \phi$ and set $\rho := \phi(\bar{q})$. This is the closed formula we were looking for: We have

$$\ulcorner\rho\urcorner = \ulcorner\phi(\bar{q})\urcorner = \ulcorner\gamma_q(\bar{q})\urcorner = \operatorname{diag}(q),$$

and thus,

$$P \cup \chi \vdash \delta\left(\bar{q}, \overline{\ulcorner\rho\urcorner}\right). \tag{31}$$

Furthermore, using Equations (30) and (31), one can show:

$$P \cup \chi \vdash \left[\exists \alpha_2\ \delta(\bar{q}, \alpha_2) \wedge \lambda(\alpha_2)\right] \rightarrow \delta\left(\bar{q}, \overline{\ulcorner\rho\urcorner}\right) \wedge \lambda\left(\overline{\ulcorner\rho\urcorner}\right). \tag{32}$$

Although constructing such a proof in P does not cause any technical difficulties, it is a formally extensive process and we shall skip this part. Nevertheless, the reader can easily see why this statement is plausible. Next, $P \cup \chi$ can prove $\rho \rightarrow \rho$ and thus by definition:

$$P \cup \chi \vdash \rho \rightarrow \left[\exists \alpha_2\ \delta(\bar{q}, \alpha_2) \wedge \lambda(\alpha_2)\right]. \tag{33}$$

Combining Equations (33) and (32) yields

$$P \cup \chi \vdash \rho \rightarrow \delta\left(\bar{q}, \overline{\ulcorner\rho\urcorner}\right) \wedge \lambda\left(\overline{\ulcorner\rho\urcorner}\right).$$

Hence,

$$P \cup \chi \vdash \rho \rightarrow \lambda\left(\overline{\ulcorner\rho\urcorner}\right).$$

Conversely, using Equation (31) we get:

$$P \cup \chi \vdash \lambda\left(\overline{\ulcorner\rho\urcorner}\right) \rightarrow \left[\delta\left(\bar{q}, \overline{\ulcorner\rho\urcorner}\right) \wedge \lambda\left(\overline{\ulcorner\rho\urcorner}\right)\right].$$

Moreover, by existential generalization, we have

$$P \cup \chi \vdash \left[\delta\left(\bar{q}, \overline{\ulcorner\rho\urcorner}\right) \wedge \lambda\left(\overline{\ulcorner\rho\urcorner}\right)\right] \rightarrow \left[\exists \alpha_2\ \delta(\bar{q}, \alpha_2) \wedge \lambda(\alpha_2)\right].$$

Hence,

$$P \cup \chi \vdash \lambda\left(\overline{\ulcorner\rho\urcorner}\right) \rightarrow \left[\exists \alpha_2\ \delta(\bar{q}, \alpha_2) \wedge \lambda(\alpha_2)\right].$$

But the right side in the above formula is exactly $\phi(\bar{q}) = \rho$. This was to be shown. □

We can now proof the main theorem of this section:

Theorem 2.4.4. (First incompleteness theorem, Rosser 1936) *If $P \cup \chi$ is consistent and χ is primitive recursive, then there exist undecidable arithmetical propositions in $P \cup \chi$.*

Proof. First, suppose ϕ is a provable formula in $P \cup \chi$, i.e. $P \cup \chi \vdash \phi$. By definition, there exists a natural number $n \in \mathbb{N}$ such that n is the `PROOF` of $\ulcorner\phi\urcorner$, hence $n\ \mathrm{B}_\chi \ulcorner\phi\urcorner$. Since $P \cup \chi$ is consistent, we know that $\neg\, m\ \mathrm{B}_\chi \operatorname{Neg}(\ulcorner\phi\urcorner)$ holds for all m. Thus, by Definition 2.4.1, we have:

$$P \cup \chi \vdash \varphi_{B_\chi}\left(\bar{n}, \overline{\ulcorner \phi \urcorner}\right), \qquad \text{and for all } m = 1, \dots, n: \quad P \cup \chi \vdash \neg \varphi_{B_\chi N}\left(\bar{m}, \overline{\ulcorner \phi \urcorner}\right).$$

Consequently, we get

$$P \cup \chi \vdash \varphi_{B_\chi}\left(\bar{n}, \overline{\ulcorner \phi \urcorner}\right) \wedge \forall \alpha_3 \left((\alpha_3 \leq \bar{n}) \rightarrow \neg \varphi_{B_\chi N}\left(\alpha_3, \overline{\ulcorner \phi \urcorner}\right)\right),$$

which is the same as $P \cup \chi \vdash \varphi_{B_\chi^R}\left(\bar{n}, \overline{\ulcorner \phi \urcorner}\right)$. Using existential generalization, we get $P \cup \chi \vdash \psi\left(\overline{\ulcorner \phi \urcorner}\right)$. The same argument can be applied to ψ_N if $\neg\phi$ is provable. Hence, we have seen that:

$$\text{if } P \cup \chi \vdash \phi, \text{ then: } \quad P \cup \chi \vdash \psi\left(\overline{\ulcorner \phi \urcorner}\right), \qquad \text{if } P \cup \chi \vdash \neg\phi, \text{ then: } \quad P \cup \chi \vdash \psi_N\left(\overline{\ulcorner \phi \urcorner}\right). \tag{34}$$

According to Lemma 2.4.3, let ρ be a sentence such that

$$P \cup \chi \vdash \rho \leftrightarrow \neg \psi\left(\overline{\ulcorner \rho \urcorner}\right). \tag{35}$$

We will show that neither $P \cup \chi \vdash \rho$ nor $P \cup \chi \vdash \neg\rho$ hold.

1. $\underline{P \cup \chi \nvdash \rho}$
   Suppose $P \cup \chi \vdash \rho$. Then, by Equation (34), we would have
   $$P \cup \chi \vdash \psi\left(\overline{\ulcorner \rho \urcorner}\right).$$
   By Equation (35) we have
   $$P \cup \chi \vdash \psi\left(\overline{\ulcorner \rho \urcorner}\right) \rightarrow \neg\rho.$$
   But by modus ponens in $P \cup \chi$ we could then conclude
   $$P \cup \chi \vdash \neg\rho,$$
   which contradicts the consistency of $P \cup \chi$.

2. $\underline{P \cup \chi \nvdash \neg\rho}$
   Suppose $P \cup \chi \vdash \neg\rho$. Then, by Equation (34), we would obtain
   $$P \cup \chi \vdash \psi_N\left(\overline{\ulcorner \rho \urcorner}\right).$$
   By the contraposition of Lemma 2.4.2 we get
   $$P \cup \chi \vdash \psi_N\left(\overline{\ulcorner \rho \urcorner}\right) \rightarrow \neg\psi\left(\overline{\ulcorner \rho \urcorner}\right).$$
   Thus, by modus ponens in $P \cup \chi$, we would have
   $$P \cup \chi \vdash \neg\psi\left(\overline{\ulcorner \rho \urcorner}\right).$$
   However, by Equation (35), we also have
   $$P \cup \chi \vdash \neg\psi\left(\overline{\ulcorner \rho \urcorner}\right) \rightarrow \rho,$$
   and could thus conclude
   $$P \cup \chi \vdash \rho,$$
   which again contradicts the consistency of $P \cup \chi$. □

Note that the second case is longer than the first. This is because the first case can be proved similarly using the standard proof predicate introduced by Gödel. The properties of Rosser's new proof predicate are only needed in the second case and allow us to drop the ω-consistency requirement.
We used the following properties of $P \cup \chi$ in the proof of Theorem 2.4.4:

1. **The theory should be consistent.** This is not surprising, as an inconsistent theory can prove any statement and as such is trivially complete.

2. **The Gödel numbers of the axioms in the theory should form a primitive recursive set.** This requirement is not restrictive in practice. If the set were not primitive recursive and we were presented with a supposed proof in the theory, we would have no effective way of verifying that proof. As such, theories without this property are usually not of interest.

3. **The theory should be able to speak about the natural numbers.** This is necessary, since the undecidable sentence ρ was an arithmetical formula. However, any theory that aims to provide a foundation for mathematics is naturally able to capture simple number theory.

One can show that these three conditions are indeed sufficient to prove the first incompleteness theorem. The version found in modern textbooks can be formulated as follows:

Theorem 2.4.5. (First incompleteness theorem, Robinson) *Let $\mathcal{T}$ be a formal theory that includes the Robinson arithmetic $\boldsymbol{Q}$. If $\mathcal{T}$ is consistent and the set of axioms $\mathcal{A}$ is primitive recursive, then $\mathcal{T}$ is incomplete.*

By $\mathcal{A}$ primitive recursive, we mean that there is a Gödel coding for $\mathcal{T}$ such that the set of Gödel numbers of the elements of $\mathcal{A}$ is primitive recursive. In combination with Gödel's completeness theorem (Theorem 1.6.5), we obtain the following result for theories of first-order:

Theorem 2.4.6. *Let $\mathcal{T}$ be a first-order theory containing the Robinson arithmetic $\boldsymbol{Q}$. If $\mathcal{T}$ is consistent and the set of axioms $\mathcal{A}$ is primitive recursive, then there exists a sentence ρ and two models $\mathcal{M}_1, \mathcal{M}_2$ of $\mathcal{T}$, such that*

$$\mathcal{M}_1 \models \rho \qquad \textit{and} \qquad \mathcal{M}_2 \not\models \rho \,.$$

2.5 The Second Incompleteness Theorem

In what is now known as *the second incompleteness theorem*, Gödel demonstrated a somewhat strange consequence of his first incompleteness theorem:

Theorem 2.5.1. (Second incompleteness theorem, Gödel 1931) *Let χ be primitive recursive. If $P \cup \chi$ can prove its own consistency then it is inconsistent.*

Proof. In his original work, Gödel presented the following proof sketch: Let Wid_χ be a sentence in our metalanguage stating that $P \cup \chi$ is consistent, e.g.

$$\mathrm{Wid}_\chi = \forall n \, \left(n, \ulcorner 0 = \mathrm{S}(0) \urcorner\right) \notin \mathrm{B}_\chi \,.$$

In Theorem 2.3.3 we only used consistency to prove $\neg \, \mathrm{Bew}_\chi \, (\mathrm{Gen}\,(19, r))$, thus:

$$\mathrm{Wid}_\chi \implies \neg \, \mathrm{Bew}_\chi \, (\mathrm{Gen}\,(19, r)) \,.$$

By Equation (20) and the definition of Bew_χ, this is the same as:

$$\mathrm{Wid}_\chi \implies \forall k \in \mathbb{N} : \left(k, \mathrm{Sb}\,(p, 23, Z(p))\right) \notin \mathrm{B}_\chi \,.$$

By Equation (16) this is equivalent to:

$$\mathrm{Wid}_\chi \implies \forall k \in \mathbb{N} : (k, p) \in Q\,.$$

Let us assume that this whole proof can be formalized within P. By Equation (17) and (19) we can represent the formula $\forall k \in \mathbb{N} : (k, p) \in Q$ in P as $\mathrm{Gen}\,(19, r)$. Let $w \in \mathbb{N}$ be the `SENTENTIAL FORMULA` in P describing Wid_χ. Define the primitive recursive function $\mathrm{Impl}(\bullet, \bullet)$ as follows:

$$\mathrm{Impl}\;(\ulcorner\varphi_1\urcorner, \ulcorner\varphi_2\urcorner) := \ulcorner\varphi_1 \to \varphi_2\urcorner\,.$$

Translating the above proof to P, we therefore obtain:

$$\mathrm{Impl}\;(w, \mathrm{Gen}\,(19, r)) \in \mathrm{Flg}\,(\chi)\;.$$

If $P \cup \chi$ could prove its own consistency, we would have $w \in \mathrm{Flg}\,(\chi)$ and thus $\mathrm{Gen}\,(19, r) \in \mathrm{Flg}\,(\chi)$. However, this was shown to be impossible in Theorem 2.3.3 if $P \cup \chi$ is consistent. □

To carry out this proof sketch, Gödel announced a continuation of his work, which, however, never appeared. We will instead derive Gödel's second incompleteness theorem as a corollary to Löb's theorem.

Theorem 2.5.2. (Löb's theorem, 1955) *Let $\mathcal{T}$ be a formal theory containing a representation of the natural numbers and with a Gödel coding, denoted as $\bar{n}$ and $\ulcorner\varphi\urcorner$ respectively. For any formula φ we write $\#\varphi := \overline{\ulcorner\varphi\urcorner}$. Suppose that the diagonal lemma as in Lemma 2.4.3 holds for $\mathcal{T}$. Moreover, let ψ be a formula in $\mathcal{T}$ with one free variable, so that for any two formulas φ, ϕ the following statements hold:*

A1: $\mathcal{T} \vdash \varphi \implies \mathcal{T} \vdash \psi(\#\varphi)$

A2: $\mathcal{T} \vdash \psi(\#(\varphi \to \phi)) \to (\psi(\#\varphi) \to \psi(\#\phi))$

A3: $\mathcal{T} \vdash \psi(\#\varphi) \to \psi(\#\,\psi(\,\#\varphi))$

In this case, we have:

$$\mathcal{T} \vdash \psi(\#\varphi) \to \varphi \quad \implies \quad \mathcal{T} \vdash \varphi\,.$$

Proof. We follow [Men97]. Suppose $\mathcal{T} \vdash \psi(\#\varphi) \to \varphi$ holds. By the diagonal lemma there is a formula ρ such that $\mathcal{T} \vdash \rho \leftrightarrow (\psi(\#\rho) \to \varphi)$. We can give the following proof in $\mathcal{T}$:

Proof of: φ		
1.	$\psi(\#\varphi) \to \varphi$	Assumption
2.	$(\psi(\#\rho) \to \varphi) \to \rho$	Diagonal lemma
3.	$\rho \to (\psi(\#\rho) \to \varphi)$	Diagonal lemma
4.	$\psi(\#(\rho \to (\psi(\#\rho) \to \varphi)))$	**A1** & formula 3
5.	$\psi(\#(\rho \to (\psi(\#\rho) \to \varphi))) \to [\psi(\#\rho) \to \psi(\#(\psi(\#\rho) \to \varphi))]$	**A2**
6.	$\psi(\#\rho) \to \psi(\#(\psi(\#\rho) \to \varphi))$	**MP**(4,5)
7.	$\psi(\#(\psi(\#\rho) \to \varphi)) \to [\psi(\#\psi(\#\rho)) \to \psi(\#\varphi)]$	**A2**
8.	$\psi(\#\rho) \to [\psi(\#\psi(\#\rho)) \to \psi(\#\varphi)]$	**TR**(6,7)
9.	$\psi(\#\psi(\#\rho)) \to [\psi(\#\rho) \to \psi(\#\varphi)]$	**AS**(8)
10.	$\psi(\#\rho) \to \psi(\#\psi(\#\rho))$	**A3**
11.	$\psi(\#\rho) \to [\psi(\#\rho) \to \psi(\#\varphi)]$	**TR**(10,9)
12.	$\psi(\#\rho) \to \psi(\#\varphi)$	trivial from 11
13.	$\psi(\#\rho) \to \varphi$	**TR**(12, 1)

Proof of: φ (continuation)		
14.	ρ	**MP**(13,2)
15.	$\psi(\#\rho)$	**A1** & formula 14
16.	φ	**MP**(15, 13)

□

Recall the formula ψ we constructed for Rosser's trick, see Definition 2.4.1. In the proof of Theorem 2.4.4, we showed that property **A1** holds for this ψ, see Equation (34). One can prove that **A2** and **A3** also hold. Moreover, similar to ψ for P, one can construct a formula ψ_{Bew} for the system **PA** which semantically describes whether a given `FORMULA` is provable (in **PA**). It is then again possible to show that the properties **A1**-**A3** hold for ψ_{Bew}. Furthermore, one can show that the diagonal lemma holds for **PA**. Löb's theorem thus states for any **PA**-formula φ:

$$\mathbf{PA} \vdash \psi_{\text{Bew}}\left(\overline{\ulcorner\varphi\urcorner}\right) \to \varphi \quad \Longrightarrow \quad \mathbf{PA} \vdash \varphi \tag{36}$$

Informally, Equation (36) claims: If φ is some formula in **PA** and **PA** can prove that the existence of a proof of φ implies φ, then φ is provable. Alternatively, we can use the contraposition to restate the equation as follows: If **PA** cannot prove φ, then it cannot prove that the existence of a proof of φ would imply φ. For a visually engaging explanation of this remarkable result, we recommend a "Cartoon Guide to Löb's Theorem", see [Yud08].

Theorem 2.5.3. (Second incompleteness theorem, **PA**) *If **PA** is consistent, it cannot prove its own consistency:*

$$\boldsymbol{PA} \nvdash \neg\,\psi_{Bew}\left(\overline{\ulcorner 0 = \mathrm{S}(0)\urcorner}\right).$$

Proof. Suppose $\mathbf{PA} \vdash \neg\,\psi_{\text{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0)\urcorner}\right)$. Using **L**2 in **PA**, we get:

$$\mathbf{PA} \vdash \psi_{\text{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0)\urcorner}\right) \to (0 = \mathrm{S}(0)).$$

Hence, by Equation (36):

$$\mathbf{PA} \vdash (0 = \mathrm{S}(0)).$$

But we also have $\mathbf{PA} \vdash \neg(0 = \mathrm{S}(0))$, which means that **PA** must be inconsistent. □

Example 2.5.4. Using the second incompleteness theorem, we can construct a consistent but ω-inconsistent theory:

$$\mathcal{T} := \mathbf{PA} + \left\{\psi_{\text{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0)\urcorner}\right)\right\} := \left(\mathcal{L}_{\text{PA}}, \mathcal{A}_{\text{PA}} \cup \left\{\psi_{\text{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0)\urcorner}\right)\right\}\right).$$

Let us assume the consistency of **PA**. The theory $\mathcal{T}$ is consistent, for if it were not, we would have:

$$\mathbf{PA} + \left\{\psi_{\text{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0)\urcorner}\right)\right\} \vdash 0 = \mathrm{S}(0).$$

The deduction theorem, see Theorem 1.4.4, would imply:

$$\mathbf{PA} \vdash \psi_{\text{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0)\urcorner}\right) \to 0 = \mathrm{S}(0).$$

Hence, by Löb's theorem, we would get:

$$\mathbf{PA} \vdash 0 = \mathrm{S}(0),$$

which is impossible by assumption. Thus $\mathcal{T}$ is consistent. However, the theory is ω-inconsistent. Let $\psi_{\text{B}}(n, k)$ be the formula in $\mathcal{L}_{\text{PA}}$ asserting that n is a `PROOF` of k in **PA**. Since we assumed **PA** to be consistent, there cannot exist a Gödel number of a proof of $(0 = 1)$ in **PA** and thus the primitive recursive function corresponding to $\psi_{\mathrm{B}}\left(\bar{n}, \overline{\ulcorner 0 = \mathrm{S}(0) \urcorner}\right)$ will not evaluate to 0 for any natural number n. Therefore, we have:

$$\text{for all } n \in \mathbb{N}, \quad \mathbf{PA} + \left\{\psi_{\mathrm{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0) \urcorner}\right)\right\} \vdash \neg\psi_{\mathrm{B}}\left(\bar{n}, \overline{\ulcorner 0 = \mathrm{S}(0) \urcorner}\right).$$

At the same time, using the definition of $\exists$ and ψ_{Bew}, we have:

$$\mathbf{PA} + \left\{\psi_{\mathrm{Bew}}\left(\overline{\ulcorner 0 = \mathrm{S}(0) \urcorner}\right)\right\} \vdash \neg\forall x \; \neg\psi_{\mathrm{B}}\left(x, \overline{\ulcorner 0 = \mathrm{S}(0) \urcorner}\right).$$

Since both statements are true at the same time, $\mathcal{T}$ is ω-inconsistent. Note that by the model existence theorem, see Theorem 1.7.3, there are structures that satisfy all axioms of $\mathcal{T}$. However, such a model is, in a sense, convinced that it does not exist. Because of this bizarre phenomenon, such models are also called *self-hating*.

One can show that Gödel's second incompleteness theorem holds for all common systems more expressive than **PA**. In particular, it holds for the set theoretic systems **ZF** and **ZFC**, which we will discuss in the next chapter.

3 BB(745) and Undecidability in ZFC – Exploring the Limits of Set Theory

In the last part of this paper, we will provide a concrete and accessible example of an undecidable proposition in **ZFC**, the system that serves as a possible foundation for most of contemporary mathematics. To accomplish this, we first introduce the notion of Turing machines and the Busy Beaver function, followed by the formulation of the undecidable proposition and its proof of undecidability. Finally, we will improve the result and give perspectives on unsuccessful attempts.

3.1 Computability and Turing Machines

We begin with a formal definition of Turing machines. This is continued with an illustrative interpretation. We follow [De 21] for general ideas and notation, but use our own formal definitions.

Definition 3.1.1. (Turing machines) A Turing machine $\mathtt{TM}$ is defined as a triple $\mathtt{TM} = (s, \Sigma, \delta)$ where

- s is a natural number, $s \geq 1$,
- $\Sigma \ni 0$ is a finite set of symbols (containing 0),
- $\delta \colon (\{1, \dots, s\} \times \Sigma) \to (\Sigma \times \{-1, 1\} \times \{0, 1, \dots, s\})$ is a function, called transition function.

For a given Turing machine $\mathtt{TM}$, we say that $\mathtt{TM}$ has s states (or $\mathtt{TM}$ is an s-state Turing machine). Furthermore, let $\mathtt{TP} = (a_i)_{i \in \mathbb{Z}} \subseteq \Sigma$ be a sequence indexed by the integers. If $a_i = 0$ for almost all $i \in \mathbb{Z}$, we call $\mathtt{TP}$ a tape of $\mathtt{TM}$ and define the functions $P_{\mathtt{TP},\mathtt{TM}}(k), \mathtt{TP}_{\mathtt{TM}}(k, i), O_{\mathtt{TP},\mathtt{TM}}(k)$ by mutual recursion:

$$P_{\mathtt{TP},\mathtt{TM}}(0) = 0 \quad \text{and} \quad P_{\mathtt{TP},\mathtt{TM}}(k) = P_{\mathtt{TP},\mathtt{TM}}(k-1) + O_{\mathtt{TP},\mathtt{TM}}(k)_2 \quad \text{for } k > 0$$

$$\mathtt{TP}_{\mathtt{TM}}(0, i) = a_i \quad \text{and} \quad \mathtt{TP}_{\mathtt{TM}}(k, i) = \begin{Bmatrix} O_{\mathtt{TP},\mathtt{TM}}(k)_1 & \text{if } i = P_{\mathtt{TP},\mathtt{TM}}(k-1) \\ \mathtt{TP}_{\mathtt{TM}}(k-1, i) & \text{if } i \neq P_{\mathtt{TP},\mathtt{TM}}(k-1) \end{Bmatrix} \quad \text{for } k > 0$$

$$O_{\mathtt{TP},\mathtt{TM}}(1) = \delta(1, a_0) \quad \text{and} \quad O_{\mathtt{TP},\mathtt{TM}}(k+1) = \begin{Bmatrix} \left(\mathtt{TP}_{\mathtt{TM}}(k, P_{\mathtt{TP},\mathtt{TM}}(k)), 0, 0\right) & \text{if } O_{\mathtt{TP},\mathtt{TM}}(k)_3 = 0 \\ \delta\left(O_{\mathtt{TP},\mathtt{TM}}(k)_3, \mathtt{TP}_{\mathtt{TM}}(k, P_{\mathtt{TP},\mathtt{TM}}(k))\right) & \text{if } O_{\mathtt{TP},\mathtt{TM}}(k)_3 \neq 0 \end{Bmatrix} \quad \text{for } k > 0$$

The sequence $(\mathtt{TP}_{\mathtt{TM}}(k, i))_{i \in \mathbb{N}}$ describes the tape and the function $O_{\mathtt{TP},\mathtt{TM}}(k)_3$ the state of the Turing machine $\mathtt{TM}$ after computation step k. If there is no $k \in \mathbb{N}$ such that $O_{\mathtt{TP},\mathtt{TM}}(k)_3 = 0$, we say that $\mathtt{TM}$ will *LOOP*. Otherwise, we say that $\mathtt{TM}$ will *HALT*. In this case, we define the *running time* of $\mathtt{TM}$ with input $\mathtt{TP}$ as:

$$\mathtt{rt}\,(\mathtt{TM}, \mathtt{TP}) := \min\{k \in \mathbb{N} \mid O_{\mathtt{TP},\mathtt{TM}}(k)_3 = 0\}$$

Furthermore, to talk about the behavior of a Turing machine $\mathtt{TM}$ given the input $\mathtt{TP}$, we write $\mathtt{TM}[\mathtt{TP}]$.

The above definition has the following visual interpretation: A Turing machine consists of an infinite two-sided tape divided into cells that can be written on, a head that can move along the tape, and a finite set of states that determine how the machine behaves. Based on a list of rules (the transition function δ), the machine reads the symbol in the current cell, updates its state, writes a new symbol to the tape, and moves the head one cell to the left or right. The machine halts when it enters the 0 state. The symbols that can be written to the tape form the set Σ, the input tape is defined by the sequence $\mathtt{TP}$, and the running time is the number of calculation steps before the machine halts (if it ever halts). See Figure 1 for an illustration. For clarity, we will work with this representation instead of the formal definition. Moreover, for our purposes, we will only allow the symbols $\Sigma = \{0, 1\}$. Unless otherwise specified, we will assume that the input tape $\mathtt{TP}$ is empty, i.e., is filled with 0's ($\mathtt{TP} \equiv 0$). We will describe a Turing machine by defining its transition function δ as a list of ordered tuples:

$$\left(\sigma_0^i, d_0^i, s_0^i, \sigma_1^i, d_1^i, s_1^i\right) \quad \in \quad \{0,1\} \times \{L, R\} \times \{0, 1, \dots, s\} \times \{0,1\} \times \{L, R\} \times \{0, 1, \dots, s\},$$

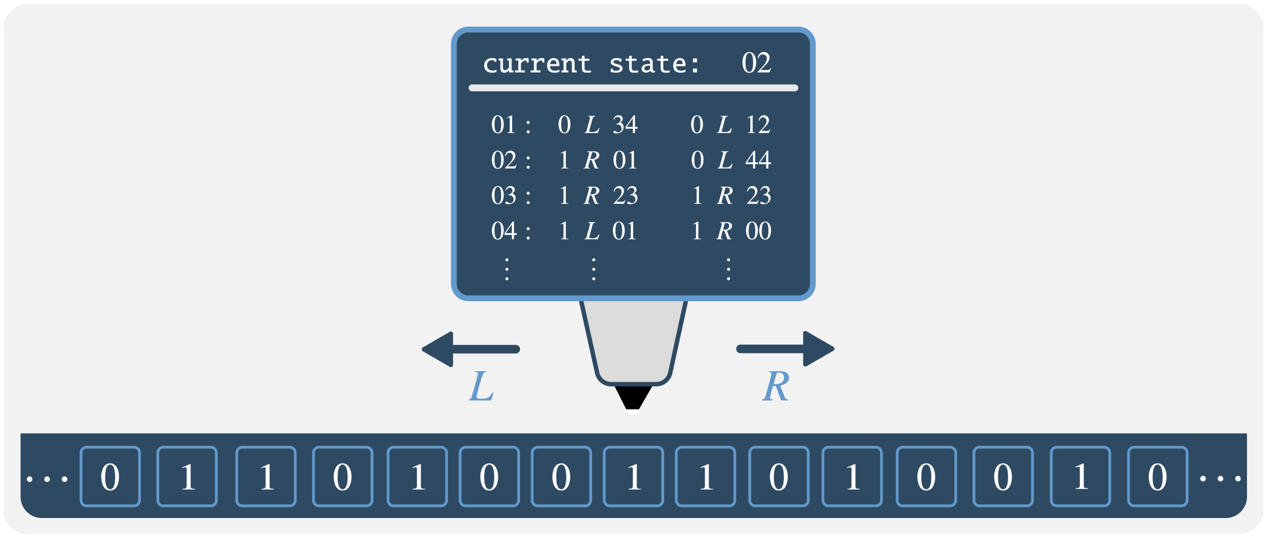

Figure 1: An illustration of a Turing machine. In this case, the head will read the current symbol (1) and, given its state (02), will decide that it should overwrite this cell with 0, move one cell to the left, and change the state to 44.

for every state $i = 1, \ldots, s$. When the machine is in state i and the current cell has the value $r \in \{0, 1\}$, the machine changes its state to s^i_r, overwrites the current symbol with σ^i_r, and moves the head one cell in the direction d^i_r. If it improves readability, we may also use a notation with leading zeros as in Figure 1.
The notion of Turing machines allows us to give a definition of computable functions:

Definition 3.1.2. (Computable functions) A function $f\colon \mathbb{N}^k \to \mathbb{N}^m$ is said to be computable if there are two effective functions I_k, I_m^{-1} that map each input $(n_1, \ldots, n_k) \in \mathbb{N}^k$ to a corresponding tape $I_k(n_1, \ldots, n_k)$ and each resulting output tape to a tuple $(n_1, \ldots, n_m) \in \mathbb{N}^m$. Moreover, there exists a Turing machine $\mathtt{TM}_f$ which halts at any input of the form $I_k(n_1, \ldots, n_k)$ and outputs a tape whose interpretation (using I_m^{-1}) is $f(n_1, \ldots, n_k)$:

$$I_m^{-1}\Big(I_k(n_1, \ldots, n_k)_{\mathtt{TM}_f}\Big(\mathtt{rt}(\mathtt{TM}_f, I_k(n_1, \ldots, n_k)), i\Big)\Big) = f(n_1, \ldots, n_k)$$

In other words, a function f is computable if there exists a Turing machine that can compute it.

We will not formally define the notion of *effective* functions as used in the above definition. It is needed to guarantee that the actual computation of f is done by the Turing machine and is not hidden in I_k or I_m^{-1}. For definiteness and for the purpose of an official definition, we fix I_k to be as follows. First we define I_1:

$$I_1(n) := (a_i)_{i\in\mathbb{Z}}\,, \quad \text{where} \quad a_1 = \cdots = a_n = 1 \quad \text{and} \quad a_i = 0\,, \quad \text{else.}$$

If I_k has already been defined for some $k \in \mathbb{N}$, we set:

$$I_{k+1}(n_1, \ldots, n_{k+1}) := \Big(I_k(n_1, \ldots, n_k)_i + I_1(n_{k+1})_{i-s-1}\Big)_{i\in\mathbb{Z}}\,, \quad \text{where} \quad s = \max\{j \in \mathbb{N} \mid I_k(n_1, \ldots, n_k)_j = 1\}.$$

In other words, we describe one natural number as a sequence of consecutive 1's, starting at the cell indexed with 1. For several natural numbers, we separate these sequences by a single 0 cell. The function I_m^{-1} is defined as the inverse function of I_m for appropriate tapes. Arguably, this method of translating natural numbers into input tapes and output tapes into natural numbers is intuitively *effective*.

The *Church–Turing thesis* states that the class of computable functions is the same as the class of intuitively computable functions. Although this is not a formal statement that can be proved or disproved, it is widely accepted by computer scientists. The thesis suggests that a function that cannot be computed by a Turing machine cannot be computed by any form of sophisticated technology.

Lemma 3.1.3. *Every primitive recursive function is computable.*

Proof. We have seen in Section 2.2 how to translate primitive recursive functions into (Python) code. Thus, by the Church–Turing thesis, the lemma follows. □

The following example shows that not every function is computable:

Theorem 3.1.4. *(Halting Problem) There is no computable function that can determine, for an arbitrary Turing machine* $\mathtt{TM}$*, whether* $\mathtt{TM}$ *will eventually halt or loop forever when given the empty tape as input.*

Proof. We will show that there is no computable function capable of determining whether a given Turing machine halts on a given tape as its input. This result is actually equivalent to our theorem, as every finite tape corresponds to some Turing machine that takes an empty tape as input and produces that tape as output. The key idea of this proof is found for example in [De 21], but we will present a more detailed argument.

The set of Turing machines and the set of tapes are countable, i.e. there are enumerations for both sets. We denote the natural numbers corresponding to the Turing machine $\mathtt{TM}$ and the tape $\mathtt{TP}$ by $n_{\mathtt{TM}}$ and $n_{\mathtt{TP}}$, respectively. We can choose the enumerations to be effective, i.e., in particular, such that

$$k \mapsto n_{I_1(k)} \tag{37}$$

is computable. Our goal is to demonstrate that the function

$$f(n_{\mathtt{TM}}, n_{\mathtt{TP}}) = \begin{cases} 1, & \text{if } \mathtt{TM} \text{ loops with input } \mathtt{TP} \\ 0, & \text{if } \mathtt{TM} \text{ halts with input } \mathtt{TP} \end{cases}$$

is not computable. Let us assume the opposite. By definition, this implies the existence of a Turing machine $\mathtt{TM}_H$ such that

$$\mathtt{TM}_H[I_2(n_{\mathtt{TM}}, n_{\mathtt{TP}})] \quad \text{outputs:} \quad \begin{cases} 1 \equiv (\dots, 0, a_1 = 1, 0, \dots), & \text{if } \mathtt{TM} \text{ loops with input } \mathtt{TP} \\ 0 \equiv (\dots, 0, 0, 0 \dots), & \text{if } \mathtt{TM} \text{ halts with input } \mathtt{TP} \end{cases}$$

By Equation (37), there exists a Turing machine $\mathtt{TM}_T$ that halts on each tape of the form $I_1(n_{\mathtt{TM}})$ and outputs

$$I_2\left(n_{\mathtt{TM}}, n_{I_1(n_{\mathtt{TM}})}\right).$$

Thus one can construct a Turing machine $\mathtt{TM}_G$ with

$$\mathtt{TM}_G[I_1(n_{\mathtt{TM}})] \quad \begin{cases} \text{halts}, & \text{if } \mathtt{TM} \text{ loops with input } I_1(n_{\mathtt{TM}}) \\ \text{loops}, & \text{if } \mathtt{TM} \text{ halts with input } I_1(n_{\mathtt{TM}}) \end{cases}$$

To obtain such a machine, first, simulate $\mathtt{TM}_T$ on the input $I_1(n_{\mathtt{TM}})$ and let $\mathtt{TM}_H$ run on the resulting output. One can add a subroutine to $\mathtt{TM}_H$ that searches the tape for a 1 and halts if and only if it finds one. Instead of allowing $\mathtt{TM}_H$ to enter the halting state directly, let it first enter the subroutine. This combination of Turing machines produces a new Turing machine with the desired property.

Since $\mathtt{TM}_G$ is a Turing machine, there exists a natural number $n_{\mathtt{TM}_G}$, and thus a well-defined tape $I_1(n_{\mathtt{TM}_G})$. If we ask whether $\mathtt{TM}_G$ halts on the input $I_1(n_{\mathtt{TM}_G})$, we obtain a contradiction: If it halts, then, by definition, $\mathtt{TM}_G$ loops with input $I_1(n_{\mathtt{TM}_G})$. On the other hand, if it loops, it must halt. □

Using the notation introduced in the last theorem, we also have the following result.

Lemma 3.1.5. *There is a primitive recursive function $f(k, n_{\mathtt{TM}}, n_{\mathtt{TP}})$ that simulates $\mathtt{TM}[\mathtt{TP}]$ for k steps and outputs the resulting tape. Furthermore, there is a primitive recursive relation $r(k, n_{\mathtt{TM}}, n_{\mathtt{TP}})$ that holds if and only if $\mathtt{TM}[\mathtt{TP}]$ halts within the first k steps.*

Proof. We will not provide a detailed demonstration. It is possible to construct a one-step Turing machine simulator using the techniques outlined in Section 2.2. This simulator executes a given machine and tape (represented as natural numbers) for precisely one step. Using this function, one can easily derive f and r. □

3.2 The Busy Beaver Function

In this section, we follow [Aar20]. For any natural number s, the set of s-state Turing machines is finite. Among them, some will eventually halt (when given the empty tape as input) after a finite number of computation steps while the others will loop forever. Hence, there exists a natural number that represents the maximum number of steps a non-looping s-state Turing machine can run before halting:

Definition 3.2.1. Let s be a natural number. If it exists, the s-th Busy Beaver number is defined as:

$$\mathbf{BB}(s) := \max\{\mathtt{rt}(\mathtt{TM}) \mid \mathtt{TM} \text{ is an } s\text{-state Turing machine that halts}\}$$

Note that the set on the right is nonempty for all $s > 0$: We can always construct a halting Turing machine with s states by setting the first state to

$$1 : \ (0, L, 0, 0, L, 0) \ ,$$

and the remaining $s - 1$ states arbitrarily. This will yield a Turing machine that halts after exactly one calculation step. Thus, we may define:

Definition 3.2.2. (Busy Beaver function) The (well-defined) function

$$\mathbf{BB}\colon \mathbb{N}_{>0} \to \mathbb{N}, \quad n \mapsto \mathbf{BB}(n) \ ,$$

is called Busy Beaver function. An s-state Turing machine with running time $\mathbf{BB}(s)$ is called s-state Busy Beaver.

Example 3.2.3. The first Busy Beaver number is equal to 1 (i.e. $\mathbf{BB}(1) = 1$).

Proof. Notice that a 1-state Turing machine $\mathtt{TM}$ of the form

$$1 : \ (*, *, 0, *, *, *)$$

will always have running time $\mathtt{rt}(\mathtt{TM}) = 1$ when given an empty tape as input. On the other hand, a 1-state Turing machine of the form

$$1 : \ \left(*, d_0^1, 1, *, *, *\right),$$

behaves as follows when given an empty tape as input: It writes down some symbol, moves the head one step in the direction d_0^1, writes down a symbol, moves the head in the direction d_0^1, writes down a symbol, and so on. Therefore, such a machine will loop whenever it receives an empty tape as input. Consequently, we get $\mathbf{BB}(1) = 1$. □

Lemma 3.2.4. *The Busy Beaver function is strictly increasing.*

Proof. Let $k > 0$ and let $\mathtt{TM}$ be a k-state Busy Beaver. We start by defining a new state:

$$k + 1 : \ (0, L, 0, 0, L, 0) \, .$$

Next, we modify `TM` so that it enters this new state instead of the halting state. To be precise, in every state

$$i: \left(\sigma_0^i, d_0^i, s_0^i, \sigma_1^i, d_1^i, s_1^i\right)$$

of `TM`, we replace s_0^i by $k+1$ if $s_0^i = 0$ and s_1^i by $k+1$ if $s_1^i = 0$. By construction, the resulting Turing machine has $k+1$ states and a running time of $\mathbf{BB}(k)+1$. Thus, $\mathbf{BB}(k+1) \geq \mathbf{BB}(k)+1 > \mathbf{BB}(k)$. □

Lemma 3.2.5. *If $f\colon \mathbb{N} \to \mathbb{N}$ is a computable function, there is an $N \in \mathbb{N}$ such that*

$$f(n) < \boldsymbol{BB}(n)$$

for all $n \geq N$. In particular, $\boldsymbol{BB}(\bullet)$ is not computable.

Proof. We will show this lemma using a similar proof idea as in [Aar20]. Let f be a fixed computable function. We define the following Turing machines:

$$\begin{aligned} \texttt{TM}_f: &\quad I_1(n) \mapsto I_1(f(n)) && [c \text{ states}] \\ \texttt{TM}_1: &\quad I_2(n,k) \mapsto I_1\left(n^2+k\right) && [d \text{ states}] \\ \texttt{TM}_{r,k}: &\quad I_1(0) \mapsto I_2(r,k) && [r+k+4 \text{ states}] \end{aligned}$$

The machine $\texttt{TM}_{r,k}$ (for $r, k > 1$) can be designed as follows: First, use $r+k+2$ states to print $I_2(r,k)$ on the tape and position the head over the cell one to the left of the last 1-cell. Next, use additional two states to move the head back to the 0-th cell by checking for two consecutive 0-cells. As with $\texttt{TM}_{r,k}$, we can assume that $\texttt{TM}_f$ and $\texttt{TM}_1$ are also designed to halt at the zeroth cell. Note that c, d are constants and do not depend on the input of the machines. For a given $n \in \mathbb{N}$, we design the following new machine:

$$\texttt{TM}_{f,n} := \texttt{TM}_f \circ \texttt{TM}_1 \circ \texttt{TM}_{r,k},$$

where $r := \left\lfloor \sqrt{n} \right\rfloor$ and $k := n - r^2$. With the composition symbol $\circ$ we indicate that the Turing machines are executed one after the other. This procedure can be summarized in an obvious way to a Turing machine with $c+d+r+k+4$ states which we call $\texttt{TM}_{f,n}$. We have the following estimations:

$$n = \left(\sqrt{n}\right)^2 \leq (r+1)^2 = r^2+2r+1 \implies k = n - r^2 \leq 2r+1\,.$$

Therefore, $\texttt{TM}_{f,n}$ is a Turing machine, with less than or equal to $c+d+5+3r < q+3r$ states, for some constant q, and that given the empty tape as an input, will output $I_1(f(n))$. In particular, this machine will run for at least $f(n)$ steps before halting. Choose $N \in \mathbb{N}$ big enough such that $n \geq q + 3\sqrt{n}$ for all $n \geq N$. We then have

$$\mathbf{BB}(n) \geq \mathbf{BB}(q+3r) > \mathbf{BB}(c+d+5+3r) \geq \mathtt{rt}\left(\texttt{TM}_{f,n}\right) \geq f(n)\,,$$

for all $n \geq N$. This was to be demonstrated. □

We can immediately improve this result:

Lemma 3.2.6. *The Busy Beaver function grows faster than any computable function. To be more precise, if f is a computable function, we have*

$$\lim_{n\to\infty} \frac{\mathbf{BB}(\mathrm{n})}{f(n)} = \infty\,.$$

Proof. Let $C \in \mathbb{N}$ be any natural number. The function $C \cdot f$ is computable and thus, by the previous lemma, there is a natural number $N(C) \in \mathbb{N}$ such that

$$\mathbf{BB}(n) > C \cdot f(n) \quad \Longleftrightarrow \quad \frac{\mathbf{BB}(n)}{f(n)} > C\,,$$

for all $n \geq N(C)$. Since C was arbitrary, the lemma follows. □

Not only can we not compute $\mathbf{BB}(n)$ for almost all $n \in \mathbb{N}$ by finite means; for sufficiently large n, even if we had the correct value $k := \mathbf{BB}(n)$, we could not prove that $\mathbf{BB}(n) = k$. This is captured by Theorem 3.2.7.

Theorem 3.2.7. *Let $\mathcal{T}$ be a consistent and primitive recursive theory. If $\mathcal{T}$ includes Peano arithmetic, there exists a natural number $N_{\mathcal{T}}$ such that for all $n \geq N_{\mathcal{T}}$,*

$$\mathcal{T} \nvdash \mathbf{BB}\,(\bar{n}) = \overline{\mathbf{BB}\,(n)}\,.$$

Proof. Since $\mathcal{T}$ is primitive recursive, we can construct a Turing machine TM that searches for contradictions in $\mathcal{T}$. More precisely, one can design a proof relation B for $\mathcal{T}$, similar to Theorem 2.2.12, and show that it is primitive recursive. Next, one can design a Turing machine $\mathcal{T}$ that loops through all $n \in \mathbb{N}$ and halts if and only if it finds one number n such that

$$n \;\mathrm{B}\; \ulcorner 0 = \mathrm{S}(0) \urcorner\,.$$

Thus, TM loops if and only if

$$\neg\, \mathrm{Bew}\left(\ulcorner 0 = \mathrm{S}(0)\urcorner\right). \tag{38}$$

Let $N \in \mathbb{N}$ be the state count of TM and $n \geq N$. Suppose we had $\mathcal{T} \vdash \mathbf{BB}\,(\bar{n}) = \overline{\mathbf{BB}\,(n)}$. Then, using an argument similar to Lemma 3.1.5, one could show that $\mathcal{T}$ could simulate TM for $\mathbf{BB}\,(n) \geq \mathbf{BB}\,(N)$ steps and prove that TM will not have halted (this can be asserted from our meta logic, since $\mathcal{T}$ is assumed to be consistent). Using the formal definition of **BB** in $\mathcal{T}$, the theory could thus conclude that TM will never halt. Moreover, $\mathcal{T}$ is expressive enough to understand (prove) that the statement that TM will never halt is equivalent to Equation (38). Thus, $\mathcal{T}$ would be able to prove its own consistency, which contradicts Gödel's second incompleteness theorem. □

We will denote by $N_{\mathcal{T}}$ the smallest possible number such that the foregoing theorem holds.

3.3 ZF and ZFC

The goal of the remainder of this work will be to find an upper bound on $N_{\mathcal{T}}$ for a specific theory. In this section, two theories are introduced: **ZF** and **ZFC**.

Unlike other first-order systems we have seen thus far, set theory allows to quantify over sets. This is achieved by using sets as objects of the language. By carefully choosing its axioms, one avoids Russell's antinomy, making it an alternative solution to type theory. Together with the advantages of first-order theories and the expressiveness that can be achieved with set theory, this idea provides a promising framework for much of modern mathematics. Among the most famous such theories are **ZF** and **ZFC**, the Zermelo-Fraenkel set theory (without or with the axiom of choice). In this section, we define a system slightly weaker than **ZF**, but sufficient to study $N_{\mathbf{ZFC}}$.

Our definitions are designed to be implemented in a Turing machine. Therefore, they may differ from the versions found in textbooks. Readers who want to explore the background of our definitions further may refer to [ORe16, zf2.nql].

To define a theory and deductive system, we have to specify the language as well as the set of axioms, the inference rules and the set of logical axioms. We set $\mathcal{L}_{ZF-R}$ as the first-order language with $\mathcal{R} = \{\in\}$, $\mathcal{V} = \{v_i \mid i \in \mathbb{N}\}$, $\mathcal{C} = \mathcal{F} = \emptyset$. Next, we define the set of logical axioms:

Definition 3.3.1. The logical axioms $\mathcal{A}_{\mathrm{Lo}}$ are:

L1. $(\varphi \to \psi) \to ((\psi \to \chi) \to (\varphi \to \chi))$

L2. $((\neg\varphi \to \varphi) \to \varphi)$

L3. $(\varphi \to (\neg\varphi \to \psi))$

L4. $\forall\xi\, (\varphi \to \psi) \to (\forall\xi\, \varphi \to \forall\xi\, \psi)$

L5. $\forall\xi\, \forall\zeta\, \varphi \to \forall\zeta\, \forall\xi\, \varphi$

L6. $(\exists\xi\, \forall\xi\, \varphi) \to \varphi$

L7. $\exists\xi\, \xi = \zeta$

L8. $\xi = \zeta \to (\xi = \vartheta \to \zeta = \vartheta)$

L9. $\xi = \zeta \to (\xi \in \vartheta \to \zeta \in \vartheta)$

L10. $\xi = \zeta \to (\vartheta \in \xi \to \vartheta \in \zeta)$

L11. $[\xi = \zeta \to \forall\vartheta\, (\xi = \zeta)] \wedge [\xi \in \zeta \to \forall\vartheta\, (\xi \in \zeta)]$ where $\xi \neq \vartheta, \zeta \neq \vartheta$

Note that ξ, ζ, ϑ are meta variables and not part of the language. The above list should be interpreted as a schema. Like before, our only inference rules are **MP** and **G**. Finally, we define the axioms $\mathcal{A}_{\mathrm{ZF}-R}$ of our theory.

Definition 3.3.2. The set $\mathcal{A}_{\mathrm{ZF}-R}$ is given by:

ZF1. $\forall v_2\, (v_2 \in v_0 \leftrightarrow v_2 \in v_1) \to v_0 = v_1$

ZF2. $\left[\forall v_3\, \exists v_1\, \forall v_2\, (\forall v_1\, \varphi \to v_2 = v_1)\right] \to \left[\exists v_1\, \forall v_2\, \left(v_2 \in v_1 \leftrightarrow \exists v_3\, (v_3 \in v_0 \wedge \forall v_1\, \varphi)\right)\right]$

ZF3. $\exists v_1\, \forall v_2\, \left(\forall v_3\, (v_3 \in v_2 \to v_3 \in v_0) \to v_2 \in v_1\right)$

ZF4. $\exists v_1\, \forall v_2\, \left(\exists v_3\, (v_2 \in v_3 \wedge v_3 \in v_0) \to v_2 \in v_1\right)$

ZF5. $\exists v_1\, \left[v_0 \in v_1 \wedge \forall v_0\, \left(v_0 \in v_1 \to \exists v_2\, (v_2 \in v_1 \wedge \forall v_1\, (v_1 \in v_2 \leftrightarrow v_1 = v_0))\right)\right]$

These axioms are known as axioms of *extensionality*, *replacement*, *power set*, *union* and *infinity*, respectively. We have made use of variable shadowing to keep the formulation simpler for our implementation later on.

The resulting theory will be denoted by $\mathcal{T}_{\mathrm{ZF}-R}$. If we extend $\mathcal{A}_{\mathrm{ZF}-R}$ by the *axiom of regularity*, we get the theory of **ZF**. If we further extend this system by the *axiom of choice*, we get the theory of **ZFC**. However, it can be shown that $\mathcal{T}_{\mathrm{ZF}-R}$ is consistent if and only if **ZFC** is consistent (and this proof can be reproduced in **ZFC**), see e.g. [Göd38] and [Vau01]. Because of this result, we can work with the simpler system $\mathcal{T}_{\mathrm{ZF}-R}$ for our purposes.

3.4 The Original Paper: BB(7910) and BB(748)

A first upper bound on $N_{\mathbf{ZFC}}$ was given by Adam Yedidia and Scott Aaronson in [YA16], where they showed that $N_{\mathbf{ZFC}} \leq 7910$. Unlike in Theorem 3.2.7, they did not construct a Turing machine that tries to find a contradiction in **ZFC**. Instead, they used a statement about order-invariant graphs, which Harvey Friedman proved to imply the consistency of **ZFC**. To construct a Turing machine that halts if and only if the statement is false, the authors developed a high-level programming language called *Laconic* that compiles to Turing machines.

This result was improved by Stefan O'Rear with the introduction of a new high-level language for Turing machines, `NQL` (Not-Quite-Laconic), see [ORe16]. Not-Quite-Laconic uses a different compilation method based on so-called *register machines*. With his compiler, O'Rear was able to improve the upper bound on $N_{\mathbf{ZFC}}$ to 1919 by constructing a proof enumerator. Further improvements to the `NQL` code resulted in a Turing machine with only 748 states. In the following, we will examine the corresponding code.

3.4.1 Basic Features of the Code

Every `NQL` project is built around a main loop that runs forever until a certain condition is met and the program stops. In our case, the corresponding condition is that a proof of $v_0 \in v_0$ has been found. Applying the generalization rule to this statement yields a formula that asserts that every set contains itself; this is easily disproved by $\mathcal{T}_{\mathrm{ZF}-R}$. Thus, $v_0 \in v_0$ can only be proved by $\mathcal{T}_{\mathrm{ZF}-R}$ (i.e., the resulting program only stops) if $\mathcal{T}_{\mathrm{ZF}-R}$ is inconsistent. On the other hand, if $\mathcal{T}_{\mathrm{ZF}-R}$ is inconsistent, it will prove anything, including $v_0 \in v_0$, and the machine will therefore eventually stop. `NQL` works like one would expect from a simple high-level language. Instead of functions, one can define procedures, short `proc`, that act similar to functions. Global variables are supported and can be accessed from anywhere in the code. Furthermore, the only available data type in `NQL` is natural numbers. Lists or arrays, for example, are not supported and need to be implemented by hand. The operation "`-`" for natural numbers is supported, however clamps at 0.

3.4.2 Encoding of Formulas and the WFF-Stack

Proofs are managed in Hilbert style with a large stack of proven formulas, the wff-stack (wff = well-formed-formulas). This stack is updated in each step of a proof and is not cleared between proofs. The algorithm modifies the stack several times during a proof step, but undoes the modifications if they were not part of the proof step. After each step, the algorithm checks whether the formula $v_0 \in v_0$ has been added to the top of the stack. If so, the program terminates, since this implies the existence of a proof of $v_0 \in v_0$. Otherwise, it continues with the next proof step. When the proof is finished, it starts checking the next proof. The wff-stack is represented as a list of natural numbers, which we can think of as formulas. For technical reasons, the wff-stack is separated into the top layer, `topwff`, and the rest of the stack, `wffstack`. To each well formed formula, we (recursively) assign a natural number in the following way:

$$\begin{aligned} \left|v_i = v_j\right| &\rightsquigarrow \Big((\,i\,.\,j\,)\,.\,0 \Big) \\ \left|v_i \in v_j\right| &\rightsquigarrow \Big((\,i\,.\,j\,)\,.\,1 \Big) \\ |\varphi \to \psi| &\rightsquigarrow \Big((\,|\varphi|\,.\,|\psi|\,)\,.\,2 \Big) \\ |\neg\varphi| &\rightsquigarrow (\,|\varphi|\,.\,3\,) \\ |\forall v_i\, \varphi| &\rightsquigarrow \Big((\,i\,.\,|\varphi|\,)\,.\,4 \Big) \end{aligned} \tag{39}$$

The notation $(\,k\,.\,n\,)$ signifies Cantor's pairing function from Lemma 2.2.7. For example, 0 corresponds to the formula $v_0 = v_0$ since $\Big((\,0\,.\,0\,)\,.\,0 \Big) = 0$ and 1 corresponds to the formula $v_0 \in v_0$ since $\Big((\,0\,.\,0\,)\,.\,1 \Big) = 1$. The wff-stack is also constructed using this pairing function. We can think of it as right-to-left oriented, i.e. the nesting takes place on the right argument. For example, a wff-stack consisting of the formulas $a_1, \ldots, a_n$ will be represented by `topwff`, `wffstack`, where:

$$\texttt{topwff} = a_n\,, \qquad \texttt{wffstack} = \Big(a_{n-1}\,.\,(\,a_{n-2}\,.\,(\,\ldots(\,a_2\,.\,a_1\,)\,)\,\Big).$$

Items can be added to the wff stack using the `pushwff()` procedure, or removed using `popwff()`. Applying `pushwff()` to the stack above would result in

$$\texttt{topwff} = 0\,, \qquad \texttt{wffstack} = \Big(a_n\,.\,(\,a_{n-1}\,.\,(\,\ldots(\,a_2\,.\,a_1\,)\,)\,\Big).$$

Conversely, applying `popwff()` would yield:

$$\texttt{topwff} = a_{n-1}\,, \qquad \texttt{wffstack} = \Big(a_{n-2}\,.\,(\,a_{n-3}\,.\,(\,\ldots(\,a_2\,.\,a_1\,)\,)\,\Big)$$

Hence, this stack follows the LIFO (last in first out) principle. The stack is well behaved for stack underflows. The empty stack is represented by

$$\texttt{topwff} = 0\,, \qquad \texttt{wffstack} = 0 = (\,0\,.\,0\,)\,,$$

and popping an element from this stack will result in the same stack. This can also be interpreted as the stack consisting of three equal formulas, each $v_0 = v_0$.

Formulas can be constructed in code using the definitions in Equation (39). Cantor's pairing function and its inverse are evaluated using highly efficient compiler implementations:

```
proc pair(out, in1, in2) { builtin_pair(out, in1, in2); }
proc unpair(out1, out2, in) { builtin_unpair(out1, out2, in); }
```

Code Snippet 1: Procs for pairing and unpairing

By checking its implementation, see Section 3.6, one can verify that `builtin_pair()` will destroy `in1` and `in2` (i.e. reset those variables). Similarly, `builtin_unpair()` will destroy `in`. Both pairing and unpairing overwrite the output variable(s). The following procs are implemented to recursively define formulas:

```
proc pushwff() { pair(wffstack, topwff, wffstack); }
proc popwff() { unpair(topwff, wffstack, wffstack); }
proc v_0_() { pushwff(); topwff = topwff + 0; }
proc v_1_() { pushwff(); topwff = topwff + 1; }
proc v_2_() { pushwff(); topwff = topwff + 2; }
proc v_3_() { pushwff(); topwff = topwff + 3; }
proc v_4_() { pushwff(); topwff = topwff + 4; }

proc cons() { unpair(t2, wffstack, wffstack); pair(topwff, t2, topwff); }
proc weq() { cons(); v_0_(); cons(); }
proc wel() { cons(); v_1_(); cons(); }
proc wim() { cons(); v_2_(); cons(); }
proc wn()  { v_3_(); cons(); }
proc wal() { cons(); v_4_(); cons(); }

proc wex() { wn(); wal(); wn(); }
proc wa() { wn(); wim(); wn(); }
```

Code Snippet 2: Procs for formula definition

The procedures `pushwff()` and `popwff()` were discussed above. The procedures `v_i_()` are short for pushing the current `topwff` to the stack and setting the new `topwff` to i. The variable `topwff` is incremented by i to save some states. However, since it is executed right after `pushwff`, `topwff` has been set to 0 and the effect is as described. Similar to `v_i_()`, `cons()` is also a helper function. Given the stack

$$\texttt{t2} = c\,, \qquad \texttt{topwff} = a_n\,, \qquad \texttt{wffstack} = \Big(a_{n-1}\,.\,(\,a_{n-2}\,.\,(\,\ldots(\,a_2\,.\,a_1\,)\,)\,\Big),$$

executing `cons()` will yield:

$$\texttt{t2} = 0\,, \qquad \texttt{topwff} = (\,a_{n-1}\,.\,a_n\,)\,, \qquad \texttt{wffstack} = \Big(a_{n-2}\,.\,(\,a_{n-3}\,.\,(\,\ldots(\,a_2\,.\,a_1\,)\,)\,\Big).$$

If we now execute, for instance, `v_4_()`, we get:

$$\texttt{t2} = 0, \qquad \texttt{topwff} = 4, \qquad \texttt{wffstack} = \Big((a_{n-1} . a_n) . (a_{n-2} . (\ldots (a_2 . a_1)) \Big).$$

Executing `cons()` again, will yield:

$$\texttt{t2} = 0, \qquad \texttt{topwff} = \Big((a_{n-1} . a_n) . 4 \Big), \qquad \texttt{wffstack} = \Big(a_{n-2} . (a_{n-3} . (\ldots (a_2 . a_1)) \Big).$$

Note that the value of `topwff` is, by definition, now equal to $\left| \forall v_{a_{n-1}} \varphi_{a_n} \right|$, where φ_{a_n} is the formula with $\left| \varphi_{a_n} \right| = a_n$. Next, the following procedures are defined:

```
proc v_0() { v_0_(); noop_7(); }
proc v_1() { v_1_(); noop_7(); }
proc v_2() { v_2_(); noop_7(); }
proc v_3() { v_3_(); noop_7(); }

proc par1() { pushwff(); topwff = param1; noop_7(); }
proc par2() { pushwff(); topwff = param2; noop_7(); }
proc par3() { pushwff(); topwff = param3; noop_7(); }
```

Code Snippet 3: Procs for pushing variables/parameters

These procs push the natural number corresponding to $v_0, \ldots, v_3$ or the current value of `param1`, ..., `param3` onto the stack. The `noop_7()` call is short for "no operation" (with argument 7). It is usually only used by the compiler for alignment purposes, and effectively does nothing. O'Rear added `noop_7()` to the code because it helps to save some states. However, since we are not going to examine the compiler in much detail, we will not explore this topic further. We can now deduce the behavior of `wal()`. Suppose we are given a stack

$$\texttt{t2} = c, \qquad \texttt{topwff} = a_n, \qquad \texttt{wffstack} = \Big(a_{n-1} . (a_{n-2} . (\ldots (a_2 . a_1)) \Big)$$

and have `param1` $= i$, `param2` $= |\varphi|$. Then, executing `par1();par2();wal()` will yield

$$\texttt{t2} = 0, \qquad \texttt{topwff} = |\forall v_i \, \varphi|, \qquad \texttt{wffstack} = \Big(a_n . (a_{n-1} . (\ldots (a_2 . a_1)) \Big).$$

This is called "reverse Polish notation": We first push $|v_i|$ onto the stack, followed by $|\varphi|$, and then apply the `wal()` procedure, which acts on the last two elements added to the stack. Likewise, the procs `weq()` and `wel()` are used to define atomic formulas, and `wim()`, `wn()`, `wex()`, and `wa()` to define implications, negations, existential quantifications, and logical conjunctions of formulas, respectively.

3.4.3 Proofs in the Code

Proofs are finite lists of quadruples: $(\texttt{axiomcode}, \texttt{param1}, \texttt{param2}, \texttt{param3}) \in \mathbb{N}^4$. The natural number `axiomcode` specifies the axiom or inference rule used in the specific proof step. The numbers `param1`, `param2`, `param3` specify the corresponding parameters to be inserted for the given axiom or inference rule. This is necessary because our axioms are mainly axiom schemes. A proof is encoded as a natural number, again using the pairing function, as follows: A proof of length 1 is denoted as

$$\texttt{prooflist} = 1 + \Big(\texttt{axiomcode}_1 . (\texttt{param1}_1 . (\texttt{param2}_1 . (\texttt{param3}_1 . 0))) \Big).$$

A proof of length $n + 1$ is denoted as

$$\texttt{prooflist} = 1 + \Big(\texttt{axiomcode}_1 \,.\, (\texttt{param1}_1 \,.\, (\texttt{param2}_1 \,.\, (\texttt{param3}_1 \,.\, \tilde{p}\,)))\Big), \tag{40}$$

where $\tilde{p}$ is the representation of a proof of length n. Within the main loop of the program, there is a loop that iterates every proof step of the current proof. At each iteration, the following lines will be executed:

```
prooflist = prooflist - 1;
unpair(t2, prooflist, prooflist); builtin_move(axiomcode, t2);
unpair(t2, prooflist, prooflist); builtin_move(param1, t2);
unpair(t2, prooflist, prooflist); builtin_move(param2, t2);
unpair(t2, prooflist, prooflist); builtin_move(param3, t2);
v_0();
```

Code Snippet 4: Part 2 of the main loop

This unpacks the current proof step, assigns the correct values to the global variables `axiomcode`, `param1`, `param2` and `param3` and pushes the number 0 to the stack. To save some states, the values are not unpacked directly into the correct variables, but first into the (temporary) variable `t2` and then moved with `builtin_move()`. It also updates `prooflist` to $\tilde{p}$ (using the notation in Equation (40)). Note that $\tilde{p} = 0$ if and only if this was the last step of the proof. Thus, by checking that `prooflist` is not 0, we can verify that there is a next step in this proof. Conversely, if `prooflist` is equal to 0, we move on to the next proof. These lines of code are located right above the last lines (at the beginning of the main loop):

```
proc main() {
    if (prooflist == 0) {
        prooflist = nextproof;
        nextproof = nextproof + 1;
    }
```

Code Snippet 5: Part 1 of the main loop

Next, depending on the unpacked proof step, the algorithm pushes a new element to the wff-stack. If the quadruple corresponds to an axiom and the conditions of the axiom are satisfied by the parameters, the algorithm pushes this instance of the axiom onto the wff-stack. If it corresponds to an inference rule, the algorithm will pop one or two elements from the stack and – if the conditions of the inference rule are met – will push the result. If at any time the conditions are not met, the algorithm will instead push 0, i.e. $v_0 = v_0$.

The algorithm interprets each proof step as follows:

`axiomcode`	`param1`	`param2`	`param3`	Corresponding axiom
1	$\lvert\psi\rvert$	–	–	**MP**: pushes $\lvert\psi\rvert$ if $\lvert\varphi\rvert$ and $\lvert\varphi \to \psi\rvert$ proved, else $v_0 = v_0$
2	i	–	–	**G**: pushes $\lvert\forall v_i\ \varphi\rvert$, where φ is last proved formula
3	$\lvert\varphi\rvert$	$\lvert\psi\rvert$	$\lvert\chi\rvert$	pushes $\lvert$**L1**$\rvert$ with φ, ψ and χ
4	$\lvert\varphi\rvert$	–	–	pushes $\lvert$**L2**$\rvert$ with φ
5	$\lvert\varphi\rvert$	$\lvert\psi\rvert$	–	pushes $\lvert$**L3**$\rvert$ with φ and ψ
6	i	$\lvert\varphi\rvert$	$\lvert\psi\rvert$	pushes $\lvert$**L4**$\rvert$ with v_i, φ and ψ
7	i	j	k	pushes $\lvert$**L11**$\rvert$ with v_i, v_j and v_k if conditions met, else $v_0 = v_0$

axiomcode	param1	param2	param3	Corresponding axiom
8	i	j	$\lvert\varphi\rvert$	pushes $\lvert$**L5**$\rvert$ with v_i, v_j and φ
9	i	$\lvert\varphi\rvert$	–	pushes $\lvert$**L6**$\rvert$ with v_i and φ
10	i	j	–	pushes $\lvert$**L7**$\rvert$ with v_i and v_j
11	i	j	k	pushes $\lvert$**L8**$\rvert$ with v_i, v_j and v_k
12	i	j	k	pushes $\lvert$**L9**$\rvert$ with v_i, v_j and v_k
13	i	j	k	pushes $\lvert$**L10**$\rvert$ with v_i, v_j and v_k
14	–	–	–	pushes $\lvert$**ZF1**$\rvert$
15	$\lvert\varphi\rvert$	–	–	pushes $\lvert$**ZF2**$\rvert$ with φ
16	–	–	–	pushes $\lvert$**ZF3**$\rvert$
17	–	–	–	pushes $\lvert$**ZF4**$\rvert$
> 17	–	–	–	pushes $\lvert$**ZF5**$\rvert$

The empty fields in the table correspond to parameters that are ignored by the algorithm. For example, if the axiom code in the current proof step is 3, we want to execute the following lines of code:

```
par1(); par2(); wim();
par2(); par3(); wim();
par1(); par3(); wim();
wim(); wim();
```

If instead the axiom code is, for instance, 15, we want to execute:

```
v_3(); v_1(); v_2();
v_1(); par1(); wal(); v_2(); v_1(); weq(); wim();
wal(); wex(); wal();

v_1(); v_2();
v_2(); v_1(); wel();
v_3(); v_3(); v_0(); wel(); v_1(); par1(); wal(); wa(); wex();  wim();
v_3(); v_3(); v_0(); wel(); v_1(); par1(); wal(); wa(); wex();
v_2(); v_1(); wel(); wim();
wa();
wal(); wex();

wim();
```

One way to act depending on the axiom code would be to use a long if-else or switch statement. However, to save some states, this is done differently in the algorithm. We define the following procedure:

```
proc select() {
    builtin_move(t2, topwff); popwff();
    if (axiomcode > 0) {
        axiomcode = axiomcode - 1; builtin_move(topwff, t2);
    }
}
```

Code Snippet 6: Proc for axiom code selection

Suppose we are given a stack

$$\texttt{t2} = c\,, \qquad \texttt{topwff} = a_n\,, \qquad \texttt{wffstack} = \big(a_{n-1}\,.\,(a_{n-2}\,.\,(\ldots(a_2\,.\,a_1\,))\,\big).$$

If `axiomcode` is equal to 0 and we execute `select()`, we get

$$\texttt{t2} = a_n\,, \qquad \texttt{topwff} = a_{n-1}\,, \qquad \texttt{wffstack} = \big(a_{n-2}\,.\,(a_{n-3}\,.\,(\ldots(a_2\,.\,a_1\,))\,\big).$$

On the other hand, if `axiomcode` is not 0 and we execute `select()`, we get

$$\texttt{t2} = 0\,, \qquad \texttt{topwff} = a_n\,, \qquad \texttt{wffstack} = \big(a_{n-2}\,.\,(a_{n-3}\,.\,(\ldots(a_2\,.\,a_1\,))\,\big)$$

and `axiomcode` is reduced by 1. Let `Axcode_1()`, ..., `Axcode_17()`, `Axcode_G17()` denote the `NQL` code we want to execute when `axiomcode` equals $1, \ldots, 17$ and > 17 respectively. After Code Snippet 4 we find:

```
Axcode_1(); select(); Axcode_2(); select(); Axcode_3(); select();
Axcode_4(); select(); Axcode_5(); select(); Axcode_6(); select();
Axcode_7(); select(); Axcode_8(); select(); Axcode_9(); select();
Axcode_10(); select(); Axcode_11(); select(); Axcode_12(); select();
Axcode_13(); select(); Axcode_14(); select(); Axcode_15(); select();
Axcode_16(); select(); Axcode_17(); select(); Axcode_G17(); select();
```

Code Snippet 7: Part 3 of the main loop

Note that we use these abbreviations for clarity. They are written out in the original code. If `axiomcode` is greater than 1, executing `Axcode_n();select();` will reduce `axiomcode` by 1, push a new formula onto the stack and delete the second-highest formula, effectively updating `topwff`. If `axiomcode` is 1, executing `Axcode_n();select();` will also update `topwff`. However, when the next part of the code is executed, the `select()` proc will undo the changes made by `Axcode_n+1()`, since `axiomcode` will be equal to 0. This behavior continues for all subsequent axioms. Thus, only the axiom with the appropriate axiom code will change the wff-stack.

At the end of each proof step, we should check whether we have proved 1, i.e. $v_0 \in v_0$. If so, we stop, if not, we go to the next iteration in the main loop:

```
    if (topwff == 1) {
        return ;
    }
}
```

Code Snippet 8: Part 4 of the main loop

We have reached the end of the main loop and have examined most of the `NQL` code. The complete project can be found on GitHub, see [ORe16]. Compiling this code will produce a Turing machine with 748 states that will halt if and only if $\mathcal{T}_{ZF-R}$ is inconsistent.

3.5 Failed Attempts to Optimize the Result

As part of this bachelor thesis, there have been various attempts to optimize the value of 748. Unfortunately, not all of these approaches have yielded positive results, although they have provided valuable insights. In this section, we will discuss some of the strategies that proved unsuccessful.

3.5.1 Optimizing the Pairing Function

The code discussed in the last section relies heavily on Cantor's pairing function. Of course, one could try to optimize its implementation. Since NQL is a high-level language, the implementation of a procedure to compute this function can be easily achieved. The following snippets present a first implementation idea:

```
proc pair(out, in1, in2) {
    out = (in1 + in2) * (in1 + in2 + 1) / 2 + in1 ;
}
```

Code Snippet 9: Attempt 1 of a pairing proc in NQL

```
proc pair(out, in1, in2) {
    out = 0;
    in2 = in1 + in2;
    while(0 < in2) {
          out = out + in2;
          in2 = in2 - 1;
     }
     out = out + in1;
}
```

Code Snippet 10: Attempt 2 of a pairing proc in NQL

Without a doubt, the first attempt will give the correct result. However, using the operators for multiplication and division will not yield the most effective Turing machine. The compiler uses subroutines for these calculation steps, that are highly optimized for general multiplications or divisions, but do not take advantage of the underlying structure of our function. The second algorithm avoids these operators and takes advantage of triangular numbers (the "little Gauss formula") to compute Cantor's pairing function. However, this procedure requires `out` and `in1` as well as `out` and `in2` to be distinct. For example, if we were to execute `pair(wffstack, topwff, wffstack)` with the procedure in Snippet 10, and `topwff` was unequal to 0, the while loop would run forever. Moreover, both attempts do not destroy `in1` and `in2`. This is not a problem but would a priori require us to rewrite the NQL code for the proof enumerator. Alternatively, we could implement this behaviour into our procedure. As long as `in1` and `in2` are distinct variables, this is a proc with the desired property:

```
proc pair(out, in1, in2) {
    in2 = in1 + in2;
    builtin_move(temp, in1);
    while(0 < in2) {
        temp = temp + in2; in2 = in2 - 1;
    }
    out = temp;
}
```

Code Snippet 11: Attempt 3 of a pairing proc in NQL

A corresponding unpairing procedure could be implemented in NQL as follows:

```
proc unpair(out1, out2, in) {
     out1 = 0; out2 = 0;
     while(true){
          temp_out1 = out1; temp_out2 = out2;
          pair(temp,temp_out1,temp_out2);
          if (temp >= in) {
               if (temp == in) { return; }
               out1 = 0; out2 = out2 + 1;
          } else { out1 = out1 + 1; }
     }
}
```

Code Snippet 12: A possible unpairing proc in NQL

Note that we only need to use `temp_out1` and `temp_out2` if our pairing proc destroys its input. Otherwise, we can simplify the above code a bit. Also note that this unpairing proc does not destroy its input.

Let us now implement these procedures in the proof enumerator code to see how they affect the state count. We will not worry about other changes that may be necessary depending on the specific procedures added. If we replace the original pairing and unpairing implementations with the procedures provided in Snippets 11 and 12, the resulting Turing machine has 1115 states. If we replace only the pairing procedure, but keep `builtin_unpair()`, we get a Turing machine with 841 states. On the other hand, if we replace only the unpairing procedure but keep `builtin_pair()`, we get a machine with 1017 states. If we keep `builtin_unpair()` and replace `builtin_pair()` with the procedure provided in Snippet 9, we get 845 states. If we replace it with the proc in Snippet 10, we get 829 states. In all these cases, the number of states has increased by at least 81. Of course, we would have to make some changes to the code, and theoretically this could reduce the state count. However, reducing it by more than 80 states would require significant optimization. Also note that we are limited by the compiler's capabilities. For example, one might be tempted to replace the `while` loop in our procs with a recursive implementation. However, this is not supported by NQL.

We conclude that the builtin implementations are much more efficient than our procedures and further exploration in this area is unlikely to yield significant gains.

3.5.2 Using a Different Pairing Function

A next approach would be to consider a different pairing function, i.e. a bijective mapping

$$f\colon \mathbb{N}^2 \to \mathbb{N}\,, \quad (x,y) \mapsto f(x,y)$$

between $\mathbb{N}^2$ and $\mathbb{N}$. Apart from Cantor's pairing function, we will mainly consider the following functions:

$$f(x,y) := 2^x \cdot (2y+1) - 1\,, \qquad e(x,y) := \begin{cases} y^2 + x & \text{if } x \neq \max\{x,y\} \\ x^2 + x + y & \text{if } x = \max\{x,y\} \end{cases}$$

We can easily check that f is a bijection, see [Pro22]: Every positive natural number n has a unique prime factor decomposition. If we set x to be the exponent of 2 in the prime factorization of n, then the product of all other prime factors must be odd, and thus of the form $2y + 1$. The function e is called *Elegant Pair*, for reference see [Szu06]. These functions can be easily implemented as procs in NQL. For example, the following procedure implements Elegant Pair:

```
proc pair(out, in1, in2) {
     if (in1 < in2){ out = in2 * in2 + in1; }
     else { out = in1 * in1 + in1 + in2; }
}
```

Code Snippet 13: A possible implementation of Elegant Pair

Of course, it would also be necessary to introduce a corresponding unpairing procedure. However, simply replacing `builtin_pair()` with the above procedure already results in 898 states. This is not surprising since we are using an if-else statement as well as multiplication. Likewise, if we tried to implement $f(x, y)$, we would have to use a `while` loop to compute 2^x and perform a multiplication afterwards. This is worse compared to Cantor's pairing function, where we could implement the procedure either without using a while or if-else block or without using multiplication. Also, we would have to implement an unpairing procedure, which would likely add additional states.

Thus, in the absence of a more easily implementable pairing function, and given the highly efficient compiler implementations of Cantor's pairing function, we conclude that this approach is not feasible.

3.5.3 Choosing a Different Undecidable Formula

Another idea would be to choose a different formula φ that is independent of **ZFC** set theory, i.e. neither this formula nor its negation can be proven in **ZFC**, assuming **ZFC** is consistent. If we can construct an n-state Turing machine that halts if and only if φ holds, and loops otherwise (or vice versa), we can easily show that **ZFC** does not settle the value of $\mathbf{BB}(n)$. Because for if it did, we could simulate this Turing machine for $\mathbf{BB}(n)$ steps within **ZFC** and conclude that it either has halted already or will never halt, thus having obtained a proof for either φ or $\neg\varphi$. This, however, is impossible if **ZFC** is consistent.

There are various known undecidable statements for **ZFC**. One popular example is the continuum hypothesis: There is no set A such that

$$|\mathbb{N}| < |A| < \left|2^{\mathbb{N}}\right| = |\mathbb{R}|.$$

Like most other known undecidable statements, this asserts some property in infinite terms. Of course, one could create a proof enumerator that searches for a proof of such a statement. However, this would basically yield the algorithm from Section 3.4 but with a very large number (that corresponds to this statement) replacing 1 in Snippet 8. Therefore, the resulting machine would have at least 748, but probably many more states. The alternative would be to check every set A and verify that it does not satisfy $|A| < \left|2^{\mathbb{N}}\right| = |\mathbb{R}|$. Obviously, this cannot be done in a countable, finitistic matter.

There is one undecidable statement from number theory, that sounds promising on first sight:

Theorem 3.5.1. *There is an explicit Diophantine equation with* 31 *unknowns that has a solution in* $\mathbb{N}$ *if and only if* ***ZFC*** *is inconsistent. Thus, assuming* ***ZFC*** *is consistent,* ***ZFC*** *does not prove that this equation is root-free.*

Proof. See [Jon80] for the explicit formula. □

We could, in theory, construct a program that enumerates all possible combinations of these 31 unknowns, evaluates the Diophantine equation for these numbers, checks whether the result is 0 and halts in that case. This would yield a Turing machine with n states for some $n \in \mathbb{N}$ and we would have shown that $\mathbf{BB}(n)$ cannot be calculated by **ZFC**. The problem with this statement is that it depends on a constant that would first have to be calculated for **ZFC**. This constant encodes, in a sense, a proof enumerator for **ZFC** and is likely to be gigantic. Thus, we would encode a Turing machine similar to the 748 state machine we explored in 3.4, substitute it for a constant in an already large equation and build a Turing machine that tries to find a solution iterating over 14 unknowns. This is unlikely to yield a Turing machine with less than 748 states. Furthermore, we would be confronted with a technical challenge: Just copying and pasting the large constant into NQL code and building the equation solver around it, would in theory compile down to a Turing machine. However, NQL has compilation problems with very large constants and would likely run out of memory. Thus, one would have to implement a subprogram that first unpacks this number. This is also part of the strategy discussed in the next section.

An alternative goal might be to improve the upper bound on $N_{\mathbf{PA}}$. Of course, $N_{\mathbf{PA}} \leq N_{\mathbf{ZFC}} \leq 748$ because **ZFC** contains Peano arithmetic. However, since **ZFC** is considerably stronger and more expressive than **PA**, we can assume that $N_{\mathbf{PA}}$ is smaller than $N_{\mathbf{ZFC}}$. An interesting task would be to find an upper bound for **PA** by adjusting the proof enumerator of Section 3.4. However, this is not part of this paper. Instead, we will briefly discuss the possibility of finding a statement that is independent of **PA** and easily implementable using NQL. Since **PA** is the theory about natural numbers, its propositions are much more accessible to implement in NQL. There are several known undecidable theorems, such as the strengthened finite Ramsey theorem (Paris–Harrington theorem), Goodstein's theorem, and the Kanamori–McAloon theorem. These theorems could be implemented in NQL in a somewhat effective way. However, they all mix existential and universal quantifiers. To implement a meaningful halting condition, we need instead a formula consisting only of existential quantifiers (called Σ^0_1 formulas): If elements matching the existential quantifiers are found, we return. Likewise, if we are given a proposition consisting only of universal quantifiers (called Π^0_1 formulas), we can consider its negation, which is Σ^0_1. However, if the given proposition mixes existential and universal quantification, there can be no halting condition that asserts the formula. This is an implicit advantage of Gödel's second incompleteness theorem over other undecidable propositions: The sentence "There is no proof of $0 = 1$" is Π^0_1.

There are some other examples of undecidable Π^0_1 or Σ^0_1 sentences for **ZFC** and **PA**. For example, we mentioned Harvey Friedman's graph-theoretic theorem in Section 3.4. In addition, Saharon Shelah provided a true Π^0_1 sentence in [She84] that is not provable in **PA**. However, implementing these statements in NQL would require considerable effort due to the complexity of both propositions.

We conclude that at this stage there is no obviously better suited undecidable formula in either **PA** or **ZFC** than searching for a contradiction.

3.5.4 Decompressing and Simulating the Existing Turing Machine

Turing machines are Turing complete, i.e. there exists a Turing machine (so called universal Turing machine) that can simulate any Turing machine with given input. Such a machine takes a tape as input which encodes both the Turing machine to simulate, as well as its input. If we were able to...

- provide a universal Turing machine together with its coding for Turing machines and tapes,
- translate the existing 748-state Turing machine and the 0-tape to a tape according to this coding,
- compress this tape,
- build a Turing machine that first generates the decompressed tape,
- secondly decompresses this tape to the actual tape,
- and finally starts the universal Turing machine on this decompressed tape,

we would obtain a Turing machine that halts if and only if the original machine halts. However, if we are able to compress the tape efficiently enough, the resulting machine may have less than 748 states.

Constructing a universal Turing machine can easily be achieved in NQL. Let $(\mathtt{TP}(i))_{i\in\mathbb{Z}} \subseteq \{0,1\}$ be a tape with $\mathtt{TP}(i)$ representing the i-th cell. We can represent TP as a unique natural number $g(\mathtt{TP})$:

$$g(\mathtt{TP}) := \sum_{i\in\mathbb{Z}} \mathtt{TP}(i)\cdot 2^{q(i)}\,, \qquad q(i) := \begin{cases} -2i-1 & \text{if } i < 0\,, \\ 2i & \text{else} \end{cases}$$

In other words, we flip the left side of the tape to the right to get a one-sided tape which can be interpreted as the binary representation of a natural number. One can easily implement a variable for the read/write head that keeps track of its position on the single sided tape and moves accordingly. Furthermore, one can implement the decision process per step of the Turing machine, using if-else statements in `NQL`. Note that `NQL` does not currently support the option of reading tapes as input. Rather, it always assumes that it will start on an empty tape. However, we can simulate an input using natural numbers as follows: First, encode the 748-state Turing machine as a natural number and compress it accordingly. Second, construct two procedures in `NQL`, one to compute the compressed number and the other to decompress this number to the original encoding. Finally, we can construct a Turing machine in `NQL` that first executes the two procedures and passes the resulting value as input to our Turing machine simulator. This would compile to a Turing machine with the desired behavior.

A first attempt to encode our 748-state Turing machine as a natural number would be to nest the pairing function on the description of the states of the Turing machine. Although this would result in a simple to interpret number for `NQL`, the nesting of the pairing function grows much too fast to be feasible. A more parsimonious approach is the following: Our machine is defined by its transition function δ as a list of ordered tuples:

$$\left(\sigma_0^i, d_0^i, s_0^i, \sigma_1^i, d_1^i, s_1^i\right) \quad \in \quad \{0,1\} \times \{L,R\} \times \{0,1,\ldots,748\} \times \{0,1\} \times \{L,R\} \times \{0,1,\ldots,748\}$$

We can combine σ_0^i, d_0^i and σ_1^i, d_1^i to ω_0^i and ω_1^i, where

$$\omega_r^i = \begin{cases} 0 & \text{if } \sigma_r^i = 0, d_r^i = L \\ 1 & \text{if } \sigma_r^i = 0, d_r^i = R \\ 2 & \text{if } \sigma_r^i = 1, d_r^i = L \\ 3 & \text{if } \sigma_r^i = 1, d_r^i = R \end{cases}$$

The natural number we assign to our Turing machine is:

$$\sum_{i=1}^{748} \left(s_1^i + \omega_1^i \cdot 10^3 + s_0^i \cdot 10^4 + \omega_0^i \cdot 10^7\right) \cdot 10^{(i-1)8} \tag{41}$$

This is simply the integer interpretation of the string obtained by listing all states one after the other:

$$\omega_0^{748} \left[s_0^{748}\right] \omega_1^{748} \left[s_1^{748}\right] \omega_0^{747} \left[s_0^{747}\right] \omega_1^{747} \left[s_1^{747}\right] \ldots \, \omega_0^{1} \left[s_0^{1}\right] \omega_1^{1} \left[s_1^{1}\right] .$$

We use $\left[s_r^i\right]$ to denote s_r^i with leading zeros (if necessary).

The number in Equation (41) can be interpreted by a somewhat simple `NQL` procedure, and is much smaller than when nesting the pairing function. Nevertheless, it still has 5984 digits. Even if we could compress it by 90% (or even 99%), the resulting number would still have about 600 (or 60) digits, making it too large to use as a constant in `NQL`. Instead, we would have to calculate it somehow. In addition, we would need to implement the Turing machine simulator itself (including its procedures to interpret the number according to Equation (41)) and a decompression procedure. This is unlikely to compile down to a Turing machine with less than 748 states.

Another approach would be to use an existing universal Turing machine and construct a machine that outputs the encoded tape corresponding to our 748 state machine. However, `NQL` does not provide a way to control the output tape. Furthermore, while there are promising universal Turing machines with as few as 15 states (see [NW09]), their corresponding codings are much more complicated.

In light of these observations, we must conclude that this approach is unlikely to yield much improvement.

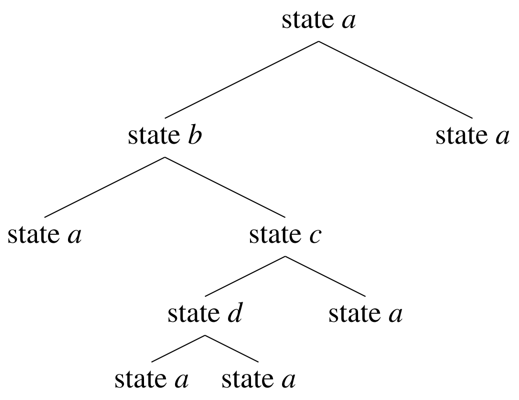

Figure 2: Example of a loop in a Turing machine

3.5.5 Peephole Optimization

A completely different strategy is optimizing the Turing machine itself, rather than the compiler or the `NQL` code. While it might be too complicated to construct a Turing machine by hand to search for a contradiction in **ZFC**, we might be able to make improvements to small parts of the existing 748 state machine. This technique is called *peephole optimization*.

A natural initial strategy is to identify ”useless” states that will never be reached because no other state leads to them. Another approach is to try to find indistinguishable states that can be merged into a single state. In the case of our Turing machine, however, no such states can be identified.

Another way is to look for loops in the state transition graph. Suppose there is a state a such that the machine will return to state a at some point in the future, regardless of the contents of the tape. Since we are only concerned with whether the machine will eventually halt or not, and the machine will never halt once it enters state a, we can replace all states reachable from a with a single looped state. The diagram in Figure 2 illustrates this idea. Here, the left child of a node denotes the state the machine will enter if the current symbol is 0, while the right child corresponds to the state entered if the current symbol is 1. In the example, if the machine enters state a, it will return to state a after at most four steps, regardless of the tape contents. Thus, we can replace all states in the tree (i.e., a, b, c, and d) with a new state e defined as

$$e : \ (0, L, e, 0, L, e)\,,$$

and replace all references to a, b, c, and d in the original Turing machine with e. The resulting Turing machine will have three less states, but will halt if and only if the original machine halts.

Unfortunately, our machine does not contain such a loop. We can use a Python function to check this, as shown in Snippet 14. Note that the time complexity of this function is not exponential; there are at most 748 different states to check in each layer. We can stop the search when the halting state is found or when the tree contains at least $748 - 3 = 745$ different states (since **ZFC** can compute the value of **BB**(4)). Running this function on every state of the 748-state Turing machine shows that there are no such loops. Except for the start states 309 and 384, the function identifies the halting state as a node in every tree after a maximum of 53 steps. If the function starts with states 309 or 384, it identifies more than 744 different states in the tree.

We could also investigate whether there is a state where the machine will always halt in the future, regardless of the contents of the tape. In this case we would look for a tree with only “halt” leaves. We can modify the function in Snippet 14 accordingly and confirm that our machine does not contain such a state.

3.6 Optimizing the Result: BB(745)

In this last section, we will show how the state count of the Turing machine can be improved.

```python
def loop_search(start_state, transition_function, max_depth=1000):
    # Initialize sets to keep track of visited and current states
    visited_states = set([start_state])
    current_states = set([start_state])

    for depth in range(max_depth):
        # Create a new set of states by applying the transition function
        next_states = set()
        for state in current_states:
            next_states.add(transition_function[state][0])
            next_states.add(transition_function[state][1])

        # Update the current set of states and remove the start state
        current_states = next_states - set([start_state])

        # Check whether all current states were the start state, if so return
        if not current_states: return "Loop detected", depth

        # Add the new states to the visited set
        visited_states.update(current_states)

        # Check whether too many states have been visited or the halting state
        # has been found, if so return
        if len(visited_states) >= len(transition_function) - 3:
            return "No loop, too many visited states", depth
        if 0 in visited_states:
            return "No loop, halt detected", depth

    # If we haven't come to a conclusion within max_depth iterations, return
    return f"No definite loop detected after {max_depth} steps", max_depth
```
Code Snippet 14: Python algorithm to search for a loop

3.6.1 Changing the Arguments in the Pairing Function

In Section 3.5.2, we examined various alternative pairing functions but did not find any function that performed better than Cantor pairing. Note that

$$(k \,_*\, n) := \tilde{C}(k, n) := C(n, k) = (n \,.\, k) = n + \frac{(n+k)\cdot(n+k+1)}{2} = n + \sum_{i=1}^{n+k} i$$

is obviously also a pairing function and can easily be implemented using `NQL`. We replace the `pair()` and `unpair()` procedures with the ones in Snippet 15.

```
proc pair(out, in1, in2) {
    builtin_pair(out, in2, in1);
}

proc unpair(out1, out2, in) {
     builtin_unpair(out2, out1, in);
}
```

Code Snippet 15: New pairing and unpairing procedures

Since we still rely on the compiler implementations, which destroy `in1`, `in2` for `pair()` and `in` for `unpair()`, we do not have to change much of the code. Note, however, that replacing the pairing function may change the formulas corresponding to 0 and 1. The number 0 is often pushed to the wff-stack, representing the true statement $v_0 = v_0$. It remains the same as $0 = (0 \,.\, 0) = (0 \,_*\, 0)$, so 0 can still be safely added to the stack at any point. Furthermore, the number 1 previously corresponded to the unprovable formula $v_0 \in v_0$. Now it corresponds to $v_1 = v_0$, which has the universal closure:

$$\forall v_0 \forall v_1 (v_1 = v_0)\,.$$

This statement can be easily refuted in $\mathcal{T}_{\mathrm{ZF}-R}$ by constructing two distinct sets. Consequently, the algorithm will only find a proof of $v_1 = v_0$ if $\mathcal{T}_{\mathrm{ZF}-R}$ is inconsistent.

Compiling the project with these new procedures yields a Turing machine with 747 states.

3.6.2 Optimizing the Pairing Function in the Compiler

As we have discussed in previous sections, the compiler implementation of Cantor's pairing function is highly efficient. It can be found in the file `nqlast.py` where it is defined as follows:

```
def emit_builtin_pair(self, out, in1, in2):
    t0 = self.get_temp()
    extract = self.gensym()
    nextdiag = self.gensym()
    done = self.gensym()
    self.emit_label(extract)
    self.emit_dec(in1)
    self.emit_goto(nextdiag)
    self.emit_inc(t0)
    self.emit_inc(in2)
    self.emit_goto(extract)
    self.emit_label(nextdiag)
    self.emit_dec(in2)
```

```
        self.emit_goto(done)
        self.emit_inc(t0)
        self.emit_transfer(in2, in1)
        self.emit_goto(extract)
        self.emit_label(done)
        self.emit_transfer(out)
        self.emit_transfer(t0, out)
        self.put_temp(t0)
```

Code Snippet 16: Pairing function in the Compiler

Since this function is part of the compiler, it will not provide a set of instructions on how to calculate the pairing of `in1` and `in2`. Rather, it describes how to create a subprogram of a Turing machine that will calculate this pairing for its interpretation of natural numbers. Since we will not study the compiler in more detail, we will instead try to understand the underlying idea of this implementation. Let t, r, k and n be given natural numbers (in what follows taken as global variables $\in \mathbb{N}$), that we can relate to the code as follows:

$$\texttt{t0} \rightsquigarrow t \in \mathbb{N} \qquad \texttt{out} \rightsquigarrow r \in \mathbb{N} \qquad \texttt{in1} \rightsquigarrow k \in \mathbb{N} \qquad \texttt{in2} \rightsquigarrow n \in \mathbb{N}$$

First, we define three methods, `extract()`, `nextdiag()`, and `done()`, which manipulate the variables t, r, k, and n, as described in Pseudocode 20, 21, and 22. We denote the initial values of n, k, and t as n_0, k_0, and t_0, respectively. Now consider what happens when we run the `extract()` method. Until k is 0, t and n are incremented

```
def extract():
    if $k == 0$:
        nextdiag()
    else:
        $k = k - 1$
        $t = t + 1$
        $n = n + 1$
        extract()
```

Algorithm 20: extract method

```
def nextdiag():
    if $n == 0$:
        done()
    else:
        $n = n - 1$
        $t = t + 1$
        $k = n + k$
        $n = 0$
        extract()
```

Algorithm 21: nextdiag method

by 1 and k is decremented by 1. After that the `nextdiag()` method is entered. At this point, the variables are:

$$n = n_0 + k_0\,, \qquad t = t_0 + k_0\,, \qquad k = 0\,.$$

```
def done(t, r, k, n):
    r = 0
    r = r + t
    t = 0
```
Algorithm 22: done method

The `nextdiag()` method increases t by 1, decreases n by 1, and transfers its value to k. By doing so, n is set to 0. Thus, the variables have the following values before re-entering `extract()`:

$$n = 0\,, \qquad t = t_0 + k_0 + 1\,, \qquad k = n_0 + k_0 - 1\,.$$

Right before entering `nextdiag()` for the second time, the variables have these values:

$$n = n_0 + k_0 - 1\,, \qquad t = t_0 + k_0 + (n_0 + k_0)\,, \qquad k = 0\,,$$

and just after exiting `nextdiag()` for the second time:

$$n = 0\,, \qquad t = t_0 + k_0 + (n_0 + k_0) + 1\,, \qquad k = n_0 + k_0 - 2\,.$$

This pattern continues. For example, before entering `nextdiag()` for the third time, the variables are:

$$n = n_0 + k_0 - 2\,, \qquad t = t_0 + k_0 + ((n_0 + k_0) + (n_0 + k_0 - 1))\,, \qquad k = 0\,,$$

and after exiting `nextdiag()` for the third time:

$$n = 0\,, \qquad t = t_0 + k_0 + ((n_0 + k_0) + (n_0 + k_0 - 1)) + 1\,, \qquad k = n_0 + k_0 - 3\,.$$

Let us skip ahead until the variables have the following values:

$$n = n_0 + k_0 - (n_0 + k_0 - 1)\,, \qquad t = t_0 + k_0 + ((n_0 + k_0) + \cdots + (n_0 + k_0 - (n_0 + k_0 - 2)))\,, \qquad k = 0\,.$$

As usual, we enter `nextdiag()` which updates n, k, t to:

$$n = 0\,, \qquad t = t_0 + k_0 + ((n_0 + k_0) + \cdots + 2 + 1)\,, \qquad k = 0\,.$$

Then we enter `extract()`, but since $n = k = 0$, we immediately skip to `done()`. This method simply sets r to t, t to 0 and does not call any other method. Hence, in conclusion, after executing `extract()` with variables of initial values $n = n_0, k = k_0$ and $t = t_0$, these variables will be updated to:

$$n = 0\,, \qquad k = 0\,, \qquad t = 0\,, \qquad r = t_0 + (\,k_0 \,.\, n_0\,)\ .$$

One can easily see how the function in Snippet 16 corresponds to this idea. Note that the compiler does not actually work with natural numbers. Instead, the variables mentioned can be understood as registers that the compiler uses for constructing the Turing machine.

If we assume the arguments of `pair()` to be distinct, we can optimize this function. We suggest the following changes: Move the lines

```
self.emit_transfer(out),    self.emit_transfer(t0, out)
```

directly after the first line (i.e. after `t0 = self.get_temp()`). Also, in the following lines, use `out` instead of `t0`.

```python
def emit_builtin_pair(self, out, in1, in2):
    t0 = self.get_temp()
    self.emit_transfer(out)
    self.emit_transfer(t0, out)
    extract = self.gensym()
    nextdiag = self.gensym()
    done = self.gensym()
    self.emit_label(extract)
    self.emit_dec(in1)
    self.emit_goto(nextdiag)
    self.emit_inc(out)
    self.emit_inc(in2)
    self.emit_goto(extract)
    self.emit_label(nextdiag)
    self.emit_dec(in2)
    self.emit_goto(done)
    self.emit_inc(out)
    self.emit_transfer(in2, in1)
    self.emit_goto(extract)
    self.emit_label(done)
    self.put_temp(t0)
```

Code Snippet 17: New pairing function in the Compiler

This corresponds to setting $r = t_0$ and $t = 0$ before executing `extract()`, replacing every occurrence of t in Algorithms 20 and 21 with r, and emptying Algorithm 22. The updated function can be found in Snippet 17.

In some projects – including our proof enumerator – this pairing implementation improves the state count (by 1 in our case). Note, however, that we cannot use this updated function with our project because our arguments for `pair()` are not always distinct. For instance, in the `cons()` procedure, the arguments `in2` and `out` are both set to `topwff`. Furthermore, although the changes made in this chapter should produce valid results, they still need to be checked for correctness with the compiler. Without compiler knowledge, however, this process will prove difficult.

Fortunately, the improvement in the next section works with the old pairing function as well, and produces a Turing machine with the same number of states as with the new function. Still, our adapted version may be useful for other projects in `NQL`. For example, in the `pairtest` project, which can be found as a sample project on GitHub, it improves the state count by one (from 239 to 238).

3.6.3 Changing the Formula Definitions

In Equation (39), we assigned a natural number to each formula. We update this definition to the following:

$$\begin{aligned}
\left|v_i = v_j\right| &\quad\rightsquigarrow\quad \Big(0 \,{}_*\, (j \,{}_*\, i)\Big) \\
\left|v_i \in v_j\right| &\quad\rightsquigarrow\quad \Big(1 \,{}_*\, (j \,{}_*\, i)\Big) \\
|\varphi \to \psi| &\quad\rightsquigarrow\quad \Big(2 \,{}_*\, (|\psi| \,{}_*\, |\varphi|)\Big) \\
|\neg\varphi| &\quad\rightsquigarrow\quad (3 \,{}_*\, |\varphi|) \\
|\forall v_i\, \varphi| &\quad\rightsquigarrow\quad \Big(4 \,{}_*\, (|\varphi| \,{}_*\, i)\Big)
\end{aligned} \tag{42}$$

Accordingly, we update the `cons()` procedure in `NQL` (switch the inputs `topwff` and `t2` in `pair`):

```
proc cons() { unpair(t2, wffstack, wffstack); pair(topwff, topwff, t2); }
```

Code Snippet 18: New definition of cons procedure

One can easily verify that `weq()`, `wel()`, `wim()`, `wn()`, `wal()`, `wex()`, and `wa()` behave as expected with the updated `cons()` procedure and formula codings. For instance, let the current state of the wff-stack be as follows:

$$\texttt{t2 = c}, \qquad \texttt{topwff} = a_n, \qquad \texttt{wffstack} = \Big(a_{n-1} * (a_{n-2} * (\ldots(a_2 * a_1))\Big)$$

If we execute `cons()`, we get

$$\texttt{t2} = 0, \qquad \texttt{topwff} = (a_n * a_{n-1}), \qquad \texttt{wffstack} = \Big(a_{n-2} * (a_{n-3} * (\ldots(a_2 * a_1))\Big).$$

Executing `v_4_()` yields

$$\texttt{t2} = 0, \qquad \texttt{topwff} = 4, \qquad \texttt{wffstack} = \Big((a_n * a_{n-1}) * (a_{n-2} * (\ldots(a_2 * a_1))\Big).$$

If we again execute `cons()`, we get

$$\texttt{t2} = 0, \qquad \texttt{topwff} = \Big(4 * (a_n * a_{n-1})\Big), \qquad \texttt{wffstack} = \Big(a_{n-2} * (a_{n-3} * (\ldots(a_2 * a_1))\Big).$$

Hence, when we execute `par1();par2();wal()` with `param1` $= i$ and `param2` $= |\varphi|$, we correctly push

$$|\forall v_i \, \varphi|$$

to the stack. The same holds true for the remaining procedures mentioned.

Note that the formula codings are, by construction, the same as the original. Thus, 0 corresponds to $v_0 = v_0$ which can safely be pushed to the stack at any time and 1 corresponds to $v_0 \in v_0$ which is provable if and only if $\mathcal{T}_{\mathrm{ZF}-R}$ is inconsistent.

Compiling the project with this change yields a Turing machine with 745 states.

To support the conclusions of this paper (and the original result), an interesting task would be to confirm the correctness of the `NQL` compiler using a proof assistant system. The same system could then be used to verify the `NQL` code that generates the 745 state machine. This would eliminate the possibility of missed bugs and result in a very rigorous validation of our results. For now we can state the following:

Theorem 3.6.1. *If the `NQL` compiler and code are both bug-free, then the equation*

$$\mathbf{BB}(745) = \underbrace{1 + \cdots + 1}_{\mathbf{BB}(745)\ \textit{times}}$$

is not provable within **ZFC** set theory.

The 745-State Turing Machine

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
001 :	0	*R*	448	1	*R*	448
002 :	0	*R*	692	1	*R*	469
003 :	0	*R*	146	1	*R*	457
004 :	0	*R*	022	1	*R*	003
005 :	0	*L*	006	1	*L*	006
006 :	0	*L*	007	0	*L*	007
007 :	0	*L*	008	0	*L*	008
008 :	1	*L*	009	1	*L*	009
009 :	1	*L*	010	1	*L*	010
010 :	0	*L*	011	0	*L*	011
011 :	0	*L*	059	0	*L*	059
012 :	0	*R*	515	1	*R*	005
013 :	0	*R*	515	1	*R*	475
014 :	0	*R*	012	1	*R*	013
015 :	0	*R*	645	1	*R*	014
016 :	0	*R*	563	1	*R*	015
017 :	0	*R*	005	1	*R*	450
018 :	0	*R*	017	1	*R*	449
019 :	0	*R*	627	1	*R*	018
020 :	0	*R*	019	1	*R*	455
021 :	0	*R*	016	1	*R*	020
022 :	0	*R*	021	1	*R*	692
023 :	0	*R*	153	1	*R*	457
024 :	0	*R*	022	1	*R*	023
025 :	0	*L*	027	1	*R*	029
026 :	0	*R*	025	0	*R*	025
027 :	1	*L*	539	1	*L*	539
028 :	0	*L*	030	1	*R*	029
029 :	0	*R*	028	1	*R*	029
030 :	0	*L*	031	1	*L*	031
031 :	0	*L*	540	0	*L*	032
032 :	1	*L*	031	1	*L*	032
033 :	1	*L*	034	0	*L*	035
034 :	0	*L*	050	1	*L*	050
035 :	1	*L*	050	0	*L*	051
036 :	0	*L*	038	1	*L*	038
037 :	1	*L*	038	0	*L*	039
038 :	0	*L*	040	1	*L*	040
039 :	1	*L*	040	0	*L*	041
040 :	0	*L*	042	1	*L*	042
041 :	1	*L*	042	0	*L*	043
042 :	0	*L*	044	1	*L*	044
043 :	1	*L*	044	0	*L*	045
044 :	0	*L*	046	1	*L*	046
045 :	1	*L*	046	0	*L*	047
046 :	0	*L*	048	1	*L*	048
047 :	1	*L*	048	0	*L*	049
048 :	0	*L*	001	1	*L*	001
049 :	1	*L*	001	0	*L*	001
050 :	0	*L*	052	1	*L*	052
051 :	1	*L*	052	0	*L*	053
052 :	0	*L*	054	1	*L*	054
053 :	1	*L*	054	0	*L*	055
054 :	0	*L*	056	1	*L*	056
055 :	1	*L*	056	0	*L*	057
056 :	0	*L*	058	1	*L*	058
057 :	1	*L*	058	0	*L*	059
058 :	0	*L*	060	1	*L*	060
059 :	1	*L*	060	0	*L*	061
060 :	0	*L*	062	1	*L*	062
061 :	1	*L*	062	0	*L*	063
062 :	0	*L*	064	1	*L*	064
063 :	1	*L*	064	0	*L*	065
064 :	0	*L*	036	1	*L*	036
065 :	1	*L*	036	0	*L*	037
066 :	0	*L*	538	0	*R*	067
067 :	1	*R*	066	1	*R*	067
068 :	0	*R*	069	1	*R*	069
069 :	0	*R*	070	1	*R*	070
070 :	1	*L*	539	1	*R*	071
071 :	0	*R*	070	1	*R*	071
072 :	0	*L*	073	1	*L*	073
073 :	0	*L*	074	0	*L*	074
074 :	1	*L*	478	1	*L*	478
075 :	0	*L*	076	1	*L*	076
076 :	1	*L*	077	1	*L*	077
077 :	0	*L*	078	0	*L*	078
078 :	0	*L*	083	0	*L*	083
079 :	0	*L*	080	1	*L*	080
080 :	1	*L*	081	1	*L*	081
081 :	0	*L*	082	0	*L*	082
082 :	1	*L*	083	1	*L*	083
083 :	0	*L*	055	0	*L*	055
084 :	0	*L*	085	1	*L*	085
085 :	0	*L*	669	0	*L*	669
086 :	0	*L*	087	1	*L*	087
087 :	0	*L*	097	0	*L*	097

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
088 :	0	L	089	1	L	089
089 :	0	L	090	0	L	090
090 :	0	L	091	0	L	091
091 :	1	L	092	1	L	092
092 :	1	L	093	1	L	093
093 :	1	L	094	1	L	094
094 :	0	L	058	0	L	058
095 :	0	L	096	1	L	096
096 :	1	L	097	1	L	097
097 :	0	L	098	0	L	098
098 :	1	L	099	1	L	099
099 :	1	L	055	1	L	055
100 :	0	L	101	1	L	101
101 :	0	L	102	0	L	102
102 :	0	L	051	0	L	051
103 :	0	L	104	1	L	104
104 :	0	L	105	0	L	105
105 :	0	L	106	0	L	106
106 :	0	L	053	0	L	053
107 :	0	L	108	1	L	108
108 :	1	L	109	1	L	109
109 :	0	L	110	0	L	110
110 :	0	L	111	0	L	111
111 :	0	L	112	0	L	112
112 :	0	L	113	0	L	113
113 :	1	L	059	1	L	059
114 :	0	R	068	1	R	068
115 :	0	R	114	1	R	114
116 :	0	R	115	1	R	115
117 :	0	R	116	1	R	116
118 :	0	R	514	1	R	100
119 :	0	R	527	1	R	095
120 :	0	R	118	1	R	119
121 :	0	R	120	1	R	454
122 :	0	R	121	1	R	575
123 :	0	R	117	1	R	122
124 :	0	R	651	1	R	601
125 :	0	R	531	1	R	514
126 :	0	R	450	1	R	450
127 :	0	R	125	1	R	126
128 :	0	R	638	1	R	127
129 :	0	R	124	1	R	128
130 :	0	R	129	1	R	456
131 :	0	R	123	1	R	130

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
132 :	0	R	609	1	R	560
133 :	0	R	629	1	R	454
134 :	0	R	132	1	R	133
135 :	0	R	134	1	R	456
136 :	0	R	684	1	R	135
137 :	0	R	131	1	R	136
138 :	0	R	609	1	R	577
139 :	0	R	631	1	R	454
140 :	0	R	138	1	R	139
141 :	0	R	140	1	R	456
142 :	0	R	684	1	R	141
143 :	0	R	609	1	R	582
144 :	0	R	633	1	R	454
145 :	0	R	143	1	R	144
146 :	0	R	145	1	R	456
147 :	0	R	684	1	R	146
148 :	0	R	142	1	R	147
149 :	0	R	137	1	R	148
150 :	0	R	609	1	R	590
151 :	0	R	635	1	R	454
152 :	0	R	150	1	R	151
153 :	0	R	152	1	R	456
154 :	0	R	684	1	R	153
155 :	0	R	154	1	R	458
156 :	0	R	155	1	R	459
157 :	0	R	149	1	R	156
158 :	0	R	024	1	R	004
159 :	0	R	702	1	R	158
160 :	0	R	157	1	R	159
161 :	0	R	512	1	R	498
162 :	0	R	655	1	R	454
163 :	0	R	619	1	R	162
164 :	0	R	518	1	R	568
165 :	0	R	164	1	R	672
166 :	0	R	657	1	R	165
167 :	0	R	588	1	R	166
168 :	0	R	163	1	R	167
169 :	0	R	088	1	R	515
170 :	0	R	079	1	R	450
171 :	0	R	169	1	R	170
172 :	0	R	171	1	R	627
173 :	0	R	103	1	R	450
174 :	0	R	173	1	R	449
175 :	0	R	174	1	R	644

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
176 :	0	R	172	1	R	175
177 :	0	R	086	1	R	450
178 :	0	R	177	1	R	449
179 :	0	R	611	1	R	178
180 :	0	R	179	1	R	580
181 :	0	R	176	1	R	180
182 :	0	R	168	1	R	181
183 :	0	R	631	1	R	611
184 :	0	R	183	1	R	162
185 :	0	R	184	1	R	456
186 :	0	R	185	1	R	457
187 :	0	R	182	1	R	186
188 :	0	R	187	1	R	558
189 :	0	R	744	1	R	188
190 :	0	R	161	1	R	189
191 :	0	R	160	1	R	190
192 :	0	R	024	1	R	459
193 :	0	R	192	1	R	498
194 :	0	R	512	1	R	735
195 :	0	R	193	1	R	194
196 :	0	R	558	1	R	459
197 :	0	R	196	1	R	498
198 :	0	R	504	1	R	744
199 :	0	R	197	1	R	198
200 :	0	R	195	1	R	199
201 :	0	R	191	1	R	200
202 :	0	R	504	1	R	512
203 :	0	R	744	1	R	498
204 :	0	R	202	1	R	203
205 :	0	R	512	1	R	744
206 :	0	R	744	1	R	744
207 :	0	R	205	1	R	206
208 :	0	R	204	1	R	207
209 :	0	R	745	1	R	459
210 :	0	R	209	1	R	498
211 :	0	R	197	1	R	210
212 :	0	R	744	1	R	196
213 :	0	R	203	1	R	212
214 :	0	R	211	1	R	213
215 :	0	R	208	1	R	214
216 :	0	R	201	1	R	215
217 :	0	R	498	1	R	498
218 :	0	R	209	1	R	504
219 :	0	R	217	1	R	218

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
220 :	0	R	206	1	R	197
221 :	0	R	219	1	R	220
222 :	0	R	744	1	R	735
223 :	0	R	202	1	R	222
224 :	0	R	498	1	R	504
225 :	0	R	735	1	R	498
226 :	0	R	224	1	R	225
227 :	0	R	223	1	R	226
228 :	0	R	221	1	R	227
229 :	0	R	194	1	R	206
230 :	0	R	504	1	R	739
231 :	0	R	197	1	R	230
232 :	0	R	229	1	R	231
233 :	0	R	161	1	R	230
234 :	0	R	735	1	R	744
235 :	0	R	234	1	R	224
236 :	0	R	233	1	R	235
237 :	0	R	232	1	R	236
238 :	0	R	228	1	R	237
239 :	0	R	216	1	R	238
240 :	0	R	737	1	R	512
241 :	0	R	240	1	R	224
242 :	0	R	737	1	R	735
243 :	0	R	744	1	R	460
244 :	0	R	242	1	R	243
245 :	0	R	241	1	R	244
246 :	0	R	580	1	R	139
247 :	0	R	515	1	R	568
248 :	0	R	518	1	R	667
249 :	0	R	247	1	R	248
250 :	0	R	658	1	R	249
251 :	0	R	596	1	R	250
252 :	0	R	246	1	R	251
253 :	0	R	088	1	R	518
254 :	0	R	107	1	R	450
255 :	0	R	253	1	R	254
256 :	0	R	255	1	R	644
257 :	0	R	174	1	R	627
258 :	0	R	256	1	R	257
259 :	0	R	654	1	R	454
260 :	0	R	593	1	R	259
261 :	0	R	258	1	R	260
262 :	0	R	252	1	R	261
263 :	0	R	659	1	R	515

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
264 :	0	R	075	1	R	450
265 :	0	R	263	1	R	264
266 :	0	R	265	1	R	627
267 :	0	R	084	1	R	450
268 :	0	R	267	1	R	449
269 :	0	R	268	1	R	644
270 :	0	R	266	1	R	269
271 :	0	R	167	1	R	270
272 :	0	R	174	1	R	611
273 :	0	R	272	1	R	455
274 :	0	R	273	1	R	456
275 :	0	R	271	1	R	274
276 :	0	R	262	1	R	275
277 :	0	R	276	1	R	558
278 :	0	R	277	1	R	498
279 :	0	R	278	1	R	202
280 :	0	R	733	1	R	279
281 :	0	R	245	1	R	280
282 :	0	R	735	1	R	735
283 :	0	R	504	1	R	498
284 :	0	R	282	1	R	283
285 :	0	R	194	1	R	234
286 :	0	R	284	1	R	285
287 :	0	R	197	1	R	224
288 :	0	R	735	1	R	460
289 :	0	R	288	1	R	461
290 :	0	R	287	1	R	289
291 :	0	R	286	1	R	290
292 :	0	R	281	1	R	291
293 :	0	R	198	1	R	197
294 :	0	R	741	1	R	293
295 :	0	R	739	1	R	460
296 :	0	R	224	1	R	295
297 :	0	R	296	1	R	741
298 :	0	R	294	1	R	297
299 :	0	R	498	1	R	512
300 :	0	R	739	1	R	504
301 :	0	R	299	1	R	300
302 :	0	R	231	1	R	301
303 :	0	R	512	1	R	739
304 :	0	R	303	1	R	206
305 :	0	R	304	1	R	231
306 :	0	R	302	1	R	305
307 :	0	R	298	1	R	306

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
308 :	0	R	292	1	R	307
309 :	0	R	239	1	R	308
310 :	0	R	737	1	R	504
311 :	0	R	299	1	R	310
312 :	0	R	512	1	R	737
313 :	0	R	312	1	R	206
314 :	0	R	311	1	R	313
315 :	0	R	161	1	R	240
316 :	0	R	231	1	R	315
317 :	0	R	314	1	R	316
318 :	0	R	504	1	R	737
319 :	0	R	318	1	R	206
320 :	0	R	196	1	R	711
321 :	0	R	711	1	R	702
322 :	0	R	320	1	R	321
323 :	0	R	319	1	R	322
324 :	0	R	737	1	R	711
325 :	0	R	704	1	R	737
326 :	0	R	324	1	R	325
327 :	0	R	744	1	R	711
328 :	0	R	327	1	R	325
329 :	0	R	326	1	R	328
330 :	0	R	323	1	R	329
331 :	0	R	317	1	R	330
332 :	0	R	737	1	R	744
333 :	0	R	321	1	R	332
334 :	0	R	333	1	R	733
335 :	0	R	735	1	R	702
336 :	0	R	704	1	R	739
337 :	0	R	335	1	R	336
338 :	0	R	719	1	R	704
339 :	0	R	212	1	R	338
340 :	0	R	337	1	R	339
341 :	0	R	334	1	R	340
342 :	0	R	711	1	R	704
343 :	0	R	498	1	R	735
344 :	0	R	342	1	R	343
345 :	0	R	739	1	R	744
346 :	0	R	342	1	R	345
347 :	0	R	344	1	R	346
348 :	0	R	289	1	R	741
349 :	0	R	347	1	R	348
350 :	0	R	341	1	R	349
351 :	0	R	331	1	R	350

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
352 :	0	R	735	1	R	704
353 :	0	R	711	1	R	711
354 :	0	R	352	1	R	353
355 :	0	R	719	1	R	719
356 :	0	R	325	1	R	355
357 :	0	R	354	1	R	356
358 :	0	R	702	1	R	737
359 :	0	R	704	1	R	498
360 :	0	R	358	1	R	359
361 :	0	R	360	1	R	289
362 :	0	R	357	1	R	361
363 :	0	R	733	1	R	741
364 :	0	R	744	1	R	719
365 :	0	R	719	1	R	702
366 :	0	R	364	1	R	365
367 :	0	R	737	1	R	704
368 :	0	R	367	1	R	343
369 :	0	R	366	1	R	368
370 :	0	R	363	1	R	369
371 :	0	R	362	1	R	370
372 :	0	R	342	1	R	332
373 :	0	R	372	1	R	733
374 :	0	R	363	1	R	373
375 :	0	R	704	1	R	711
376 :	0	R	212	1	R	375
377 :	0	R	711	1	R	737
378 :	0	R	355	1	R	377
379 :	0	R	376	1	R	378
380 :	0	R	348	1	R	379
381 :	0	R	374	1	R	380
382 :	0	R	371	1	R	381
383 :	0	R	351	1	R	382
384 :	0	R	309	1	R	383
385 :	0	R	365	1	R	332
386 :	0	R	735	1	R	711
387 :	0	R	386	1	R	325
388 :	0	R	385	1	R	387
389 :	0	R	222	1	R	461
390 :	0	R	389	1	R	741
391 :	0	R	388	1	R	390
392 :	0	R	196	1	R	704
393 :	0	R	711	1	R	719
394 :	0	R	392	1	R	393
395 :	0	R	737	1	R	719

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
396 :	0	R	393	1	R	395
397 :	0	R	394	1	R	396
398 :	0	R	358	1	R	461
399 :	0	R	398	1	R	733
400 :	0	R	397	1	R	399
401 :	0	R	391	1	R	400
402 :	0	R	741	1	R	372
403 :	0	R	402	1	R	348
404 :	0	R	702	1	R	704
405 :	0	R	392	1	R	404
406 :	0	R	737	1	R	702
407 :	0	R	406	1	R	404
408 :	0	R	405	1	R	407
409 :	0	R	324	1	R	342
410 :	0	R	367	1	R	375
411 :	0	R	409	1	R	410
412 :	0	R	408	1	R	411
413 :	0	R	403	1	R	412
414 :	0	R	401	1	R	413
415 :	0	R	702	1	R	739
416 :	0	R	367	1	R	415
417 :	0	R	744	1	R	704
418 :	0	R	417	1	R	415
419 :	0	R	416	1	R	418
420 :	0	R	375	1	R	332
421 :	0	R	420	1	R	733
422 :	0	R	419	1	R	421
423 :	0	R	289	1	R	733
424 :	0	R	741	1	R	389
425 :	0	R	423	1	R	424
426 :	0	R	422	1	R	425
427 :	0	R	518	1	R	072
428 :	0	R	518	1	R	079
429 :	0	R	427	1	R	428
430 :	0	R	641	1	R	429
431 :	0	R	619	1	R	430
432 :	0	R	665	1	R	000
433 :	0	R	432	1	R	449
434 :	0	R	644	1	R	433
435 :	0	R	434	1	R	455
436 :	0	R	431	1	R	435
437 :	0	R	436	1	R	457
438 :	0	R	437	1	R	458
439 :	0	R	558	1	R	438

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
440 :	0	*R*	439	1	*R*	460
441 :	0	*R*	440	1	*R*	461
442 :	0	*R*	441	1	*R*	451
443 :	0	*R*	363	1	*R*	442
444 :	0	*R*	443	1	*R*	452
445 :	0	*R*	426	1	*R*	444
446 :	0	*R*	414	1	*R*	445
447 :	0	*R*	446	1	*R*	453
448 :	0	*R*	384	1	*R*	447
449 :	0	*L*	035	1	*L*	035
450 :	0	*L*	033	1	*L*	033
451 :	0	*L*	037	1	*L*	037
452 :	0	*L*	041	1	*L*	041
453 :	0	*L*	047	1	*L*	047
454 :	0	*L*	051	1	*L*	051
455 :	0	*L*	053	1	*L*	053
456 :	0	*L*	055	1	*L*	055
457 :	0	*L*	057	1	*L*	057
458 :	0	*L*	059	1	*L*	059
459 :	0	*L*	061	1	*L*	061
460 :	0	*L*	063	1	*L*	063
461 :	0	*L*	065	1	*L*	065
462 :	0	*R*	520	1	*R*	568
463 :	0	*R*	462	1	*R*	480
464 :	0	*R*	463	1	*R*	484
465 :	0	*R*	612	1	*R*	487
466 :	0	*R*	464	1	*R*	465
467 :	0	*R*	611	1	*R*	641
468 :	0	*R*	467	1	*R*	455
469 :	0	*R*	466	1	*R*	468
470 :	0	*L*	471	1	*L*	471
471 :	0	*L*	472	0	*L*	472
472 :	0	*L*	473	0	*L*	473
473 :	0	*L*	474	0	*L*	474
474 :	0	*L*	054	0	*L*	054
475 :	0	*L*	476	1	*L*	476
476 :	1	*L*	477	1	*L*	477
477 :	0	*L*	478	0	*L*	478
478 :	1	*L*	053	1	*L*	053
479 :	0	*R*	525	1	*R*	568
480 :	0	*R*	528	1	*R*	529
481 :	0	*R*	479	1	*R*	480
482 :	0	*R*	470	1	*R*	516
483 :	0	*R*	475	1	*R*	528

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
484 :	0	*R*	482	1	*R*	483
485 :	0	*R*	481	1	*R*	484
486 :	0	*R*	470	1	*R*	450
487 :	0	*R*	486	1	*R*	449
488 :	0	*R*	615	1	*R*	487
489 :	0	*R*	485	1	*R*	488
490 :	0	*R*	621	1	*R*	642
491 :	0	*R*	490	1	*R*	455
492 :	0	*R*	489	1	*R*	491
493 :	0	*R*	652	1	*R*	611
494 :	0	*R*	580	1	*R*	493
495 :	0	*R*	494	1	*R*	508
496 :	0	*R*	492	1	*R*	495
497 :	0	*R*	496	1	*R*	458
498 :	0	*R*	497	1	*R*	459
499 :	0	*R*	653	1	*R*	611
500 :	0	*R*	584	1	*R*	499
501 :	0	*R*	500	1	*R*	508
502 :	0	*R*	492	1	*R*	501
503 :	0	*R*	502	1	*R*	458
504 :	0	*R*	503	1	*R*	459
505 :	0	*R*	654	1	*R*	611
506 :	0	*R*	593	1	*R*	505
507 :	0	*R*	641	1	*R*	454
508 :	0	*R*	507	1	*R*	455
509 :	0	*R*	506	1	*R*	508
510 :	0	*R*	492	1	*R*	509
511 :	0	*R*	510	1	*R*	458
512 :	0	*R*	511	1	*R*	459
513 :	0	*R*	026	1	*R*	513
514 :	0	*R*	513	1	*R*	514
515 :	0	*R*	514	1	*R*	515
516 :	0	*R*	525	1	*R*	516
517 :	0	*R*	516	1	*R*	517
518 :	0	*R*	515	1	*R*	518
519 :	0	*R*	518	1	*R*	519
520 :	0	*R*	519	1	*R*	520
521 :	0	*R*	520	1	*R*	521
522 :	0	*R*	521	1	*R*	522
523 :	0	*R*	522	1	*R*	523
524 :	0	*R*	523	1	*R*	524
525 :	0	*R*	524	1	*R*	525
526 :	0	*R*	067	1	*R*	526
527 :	0	*R*	526	1	*R*	527

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
528 :	0	*R*	527	1	*R*	528
529 :	0	*R*	537	1	*R*	529
530 :	0	*R*	529	1	*R*	530
531 :	0	*R*	571	1	*R*	531
532 :	0	*R*	531	1	*R*	532
533 :	0	*R*	532	1	*R*	533
534 :	0	*R*	533	1	*R*	534
535 :	0	*R*	534	1	*R*	535
536 :	0	*R*	535	1	*R*	536
537 :	0	*R*	536	1	*R*	537
538 :	0	*L*	033	1	*L*	539
539 :	0	*L*	538	1	*L*	539
540 :	0	*L*	449	1	*L*	541
541 :	0	*L*	540	1	*L*	541
542 :	0	*L*	543	1	*L*	543
543 :	0	*L*	081	0	*L*	081
544 :	0	*R*	616	1	*R*	607
545 :	0	*R*	639	1	*R*	454
546 :	0	*R*	544	1	*R*	545
547 :	0	*R*	546	1	*R*	456
548 :	0	*R*	547	1	*R*	699
549 :	0	*R*	521	1	*R*	542
550 :	0	*R*	533	1	*R*	521
551 :	0	*R*	549	1	*R*	550
552 :	0	*R*	126	1	*R*	449
553 :	0	*R*	551	1	*R*	552
554 :	0	*R*	609	1	*R*	611
555 :	0	*R*	553	1	*R*	554
556 :	0	*R*	555	1	*R*	508
557 :	0	*R*	556	1	*R*	457
558 :	0	*R*	548	1	*R*	557
559 :	0	*R*	521	1	*R*	597
560 :	0	*R*	559	1	*R*	600
561 :	0	*R*	521	1	*R*	568
562 :	0	*R*	561	1	*R*	571
563 :	0	*R*	562	1	*R*	574
564 :	0	*L*	565	1	*L*	565
565 :	0	*L*	566	0	*L*	566
566 :	0	*L*	567	0	*L*	567
567 :	0	*L*	052	0	*L*	052
568 :	0	*L*	569	1	*L*	569
569 :	1	*L*	102	1	*L*	102
570 :	0	*R*	519	1	*R*	568
571 :	0	*R*	528	1	*R*	571

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
572 :	0	*R*	570	1	*R*	571
573 :	0	*R*	564	1	*R*	450
574 :	0	*R*	573	1	*R*	449
575 :	0	*R*	572	1	*R*	574
576 :	0	*R*	522	1	*R*	597
577 :	0	*R*	576	1	*R*	600
578 :	0	*R*	522	1	*R*	568
579 :	0	*R*	578	1	*R*	571
580 :	0	*R*	579	1	*R*	574
581 :	0	*R*	523	1	*R*	597
582 :	0	*R*	581	1	*R*	600
583 :	0	*R*	585	1	*R*	571
584 :	0	*R*	583	1	*R*	574
585 :	0	*R*	523	1	*R*	568
586 :	0	*R*	571	1	*R*	530
587 :	0	*R*	585	1	*R*	586
588 :	0	*R*	587	1	*R*	574
589 :	0	*R*	524	1	*R*	597
590 :	0	*R*	589	1	*R*	600
591 :	0	*R*	524	1	*R*	568
592 :	0	*R*	591	1	*R*	571
593 :	0	*R*	592	1	*R*	574
594 :	0	*R*	528	1	*R*	530
595 :	0	*R*	591	1	*R*	594
596 :	0	*R*	595	1	*R*	574
597 :	0	*L*	598	1	*L*	598
598 :	1	*L*	035	1	*L*	035
599 :	0	*R*	514	1	*R*	597
600 :	0	*R*	646	1	*R*	450
601 :	0	*R*	599	1	*R*	600
602 :	0	*R*	532	1	*R*	646
603 :	0	*R*	118	1	*R*	602
604 :	0	*R*	528	1	*R*	646
605 :	0	*R*	118	1	*R*	604
606 :	0	*R*	520	1	*R*	597
607 :	0	*R*	606	1	*R*	600
608 :	0	*R*	520	1	*R*	100
609 :	0	*R*	608	1	*R*	604
610 :	0	*R*	516	1	*R*	597
611 :	0	*R*	610	1	*R*	600
612 :	0	*R*	613	1	*R*	602
613 :	0	*R*	516	1	*R*	100
614 :	0	*R*	537	1	*R*	646
615 :	0	*R*	613	1	*R*	614

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
616 :	0	*R*	613	1	*R*	604
617 :	0	*R*	516	1	*R*	568
618 :	0	*R*	617	1	*R*	571
619 :	0	*R*	618	1	*R*	574
620 :	0	*R*	525	1	*R*	597
621 :	0	*R*	620	1	*R*	600
622 :	0	*R*	624	1	*R*	602
623 :	0	*R*	624	1	*R*	640
624 :	0	*R*	525	1	*R*	100
625 :	0	*R*	624	1	*R*	604
626 :	0	*R*	515	1	*R*	597
627 :	0	*R*	626	1	*R*	600
628 :	0	*R*	533	1	*R*	646
629 :	0	*R*	636	1	*R*	628
630 :	0	*R*	534	1	*R*	646
631 :	0	*R*	636	1	*R*	630
632 :	0	*R*	535	1	*R*	646
633 :	0	*R*	636	1	*R*	632
634 :	0	*R*	536	1	*R*	646
635 :	0	*R*	636	1	*R*	634
636 :	0	*R*	515	1	*R*	100
637 :	0	*R*	527	1	*R*	646
638 :	0	*R*	636	1	*R*	637
639 :	0	*R*	636	1	*R*	602
640 :	0	*R*	529	1	*R*	646
641 :	0	*R*	636	1	*R*	640
642 :	0	*R*	636	1	*R*	614
643 :	0	*R*	518	1	*R*	597
644 :	0	*R*	643	1	*R*	600
645 :	0	*R*	649	1	*R*	628
646 :	0	*L*	647	1	*L*	647
647 :	0	*L*	648	0	*L*	648
648 :	0	*L*	050	0	*L*	050
649 :	0	*R*	518	1	*R*	100
650 :	0	*R*	531	1	*R*	646
651 :	0	*R*	649	1	*R*	650
652 :	0	*R*	649	1	*R*	630
653 :	0	*R*	649	1	*R*	632
654 :	0	*R*	649	1	*R*	634
655 :	0	*R*	649	1	*R*	640
656 :	0	*R*	517	1	*R*	100
657 :	0	*R*	656	1	*R*	632
658 :	0	*R*	656	1	*R*	634
659 :	0	*L*	660	1	*L*	660

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
660 :	0	*L*	661	0	*L*	661
661 :	0	*L*	662	0	*L*	662
662 :	1	*L*	663	1	*L*	663
663 :	1	*L*	664	1	*L*	664
664 :	0	*L*	056	0	*L*	056
665 :	0	*L*	666	1	*L*	666
666 :	0	*L*	035	0	*L*	035
667 :	0	*L*	668	1	*L*	668
668 :	1	*L*	669	1	*L*	669
669 :	0	*L*	670	0	*L*	670
670 :	0	*L*	099	0	*L*	099
671 :	0	*R*	605	1	*R*	601
672 :	0	*R*	515	1	*R*	667
673 :	0	*R*	527	1	*R*	520
674 :	0	*R*	672	1	*R*	673
675 :	0	*R*	607	1	*R*	674
676 :	0	*R*	671	1	*R*	675
677 :	0	*R*	665	1	*R*	659
678 :	0	*R*	677	1	*R*	449
679 :	0	*R*	678	1	*R*	603
680 :	0	*R*	659	1	*R*	450
681 :	0	*R*	680	1	*R*	449
682 :	0	*R*	681	1	*R*	454
683 :	0	*R*	679	1	*R*	682
684 :	0	*R*	676	1	*R*	683
685 :	0	*R*	625	1	*R*	621
686 :	0	*R*	537	1	*R*	520
687 :	0	*R*	672	1	*R*	686
688 :	0	*R*	607	1	*R*	687
689 :	0	*R*	685	1	*R*	688
690 :	0	*R*	678	1	*R*	622
691 :	0	*R*	690	1	*R*	682
692 :	0	*R*	689	1	*R*	691
693 :	0	*R*	537	1	*R*	516
694 :	0	*R*	672	1	*R*	693
695 :	0	*R*	611	1	*R*	694
696 :	0	*R*	685	1	*R*	695
697 :	0	*R*	678	1	*R*	623
698 :	0	*R*	697	1	*R*	682
699 :	0	*R*	696	1	*R*	698
700 :	0	*R*	492	1	*R*	457
701 :	0	*R*	700	1	*R*	458
702 :	0	*R*	701	1	*R*	459
703 :	0	*R*	709	1	*R*	458

State	σ_0^i	d_0^i	s_0^i	σ_1^i	d_1^i	s_1^i
704 :	0	*R*	703	1	*R*	459
705 :	0	*R*	720	1	*R*	449
706 :	0	*R*	705	1	*R*	454
707 :	0	*R*	706	1	*R*	455
708 :	0	*R*	707	1	*R*	456
709 :	0	*R*	492	1	*R*	708
710 :	0	*R*	717	1	*R*	458
711 :	0	*R*	710	1	*R*	459
712 :	0	*R*	529	1	*R*	529
713 :	0	*R*	712	1	*R*	449
714 :	0	*R*	713	1	*R*	454
715 :	0	*R*	714	1	*R*	455
716 :	0	*R*	715	1	*R*	456
717 :	0	*R*	492	1	*R*	716
718 :	0	*R*	725	1	*R*	458
719 :	0	*R*	718	1	*R*	459
720 :	0	*R*	529	1	*R*	450
721 :	0	*R*	712	1	*R*	720
722 :	0	*R*	721	1	*R*	454
723 :	0	*R*	722	1	*R*	455
724 :	0	*R*	723	1	*R*	456
725 :	0	*R*	492	1	*R*	724
726 :	0	*R*	712	1	*R*	712
727 :	0	*R*	726	1	*R*	454
728 :	0	*R*	727	1	*R*	455
729 :	0	*R*	728	1	*R*	456
730 :	0	*R*	492	1	*R*	729
731 :	0	*R*	209	1	*R*	744
732 :	0	*R*	209	1	*R*	460
733 :	0	*R*	731	1	*R*	732
734 :	0	*R*	002	1	*R*	730
735 :	0	*R*	734	1	*R*	743
736 :	0	*R*	002	1	*R*	709
737 :	0	*R*	736	1	*R*	743
738 :	0	*R*	002	1	*R*	700
739 :	0	*R*	738	1	*R*	743
740 :	0	*R*	209	1	*R*	735
741 :	0	*R*	740	1	*R*	732
742 :	0	*R*	002	1	*R*	717
743 :	0	*R*	002	1	*R*	458
744 :	0	*R*	742	1	*R*	743
745 :	0	*R*	725	1	*R*	002

References

[Aar20] Scott Aaronson. “The Busy Beaver Frontier”. In: *SIGACT News* 51.3 (2020), pp. 32–54.

[Bar77] Jon Barwise. *Handbook of Mathematical Logic*. North-Holland, 1977.

[BBJ02] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and logic*. 4th Ed. Cambridge University Press, 2002.

[Ber26] Paul Bernays. “Axiomatische Untersuchung des Aussagen-Kalküls der ’Principia Mathematica’”. In: *Mathematische Zeitschrift* 25 (1926), pp. 305–320.

[Coq22] Thierry Coquand. “Type Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University, 2022.

[Cro18] Laura Crosilla. “Exploring predicativity”. In: *Proof and Computation*. Ed. by K. Mainzer, P. Schuster, and H. Schwichtenberg. World Scientific, 2018, pp. 83–108.

[De 21] Liesbeth De Mol. “Turing Machines”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2021. Metaphysics Research Lab, Stanford University, 2021.

[Ges13] Stefan Geschke. *Model Theory*. Winter semester 2012/13. URL: `https://www.math.uni-hamburg.de/home/geschke/teaching/ModelTheory.pdf`.

[Göd29] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls*. University of Vienna, 1929.

[Göd31] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme”. In: *Monatshefte für Mathematik und Physik* 38.1 (1931), pp. 173–198.

[Göd38] Kurt Gödel. “The consistency of the axiom of choice and of the generalized continuum-hypothesis”. In: *Proceedings of the National Academy of Sciences of the United States of America* 24.12 (1938), pp. 556–557.

[HA38] David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. 2nd Ed. Springer, 1938.

[HB34] David Hilbert and Paul Bernays. *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen: Grundlagen der Mathematik – Band I*. Vol. 40. Springer-Verlag, 1934.

[HB39] David Hilbert and Paul Bernays. *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen: Grundlagen der Mathematik – Band II*. Vol. 50. Springer-Verlag, 1939.

[Hof17] Dirk W. Hoffmann. *Die Gödel’schen Unvollständigkeitssätze – Eine geführte Reise durch Kurt Gödels historischen Beweis*. Springer-Verlag, 2017.

[Jon80] James P. Jones. “Undecidable Diophantine equations”. In: *Bulletin of the American Mathematical Society* 3 (1980), pp. 859–862.

[Men97] Elliott Mendelson. *Introduction to mathematical logic*. 4th Ed. Chapman & Hall, 1997.

[NC22] Harold Noonan and Ben Curtis. “Identity”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University, 2022.

[NW09] Turlough Neary and Damien Woods. “Four Small Universal Turing Machines”. In: *Fundamenta Informaticae* 91.1 (2009), pp. 123–144.

[ORe16] Stefan O’Rear. *metamath-turing-machines*. `https://github.com/sorear/metamath-turing-machines`. 2016.

[Pro21] Open Logic Project. *Incompleteness and Computability: An Open Introduction to Gödel’s Theorems*. `https://ic.openlogicproject.org/`. Accessed March 21, 2023. 2021.

[Pro22] Open Logic Project. *Sets, Logic, Computation: An Open Introduction to Metalogic*. `https://slc.openlogicproject.org/`. Accessed March 21, 2023. 2022.

[Ram19] Srinivasa Ramanujan. “A proof of Bertrand’s postulate”. In: *Journal of the Indian Mathematical Society* 11 (1919), pp. 181–182.

[Rau09] Wolfgang Rautenberg. *A Concise Introduction to Mathematical Logic*. 3rd Ed. Springer, 2009.

[Ros36] J. Barkley Rosser. “Extensions of Some Theorems of Gödel and Church”. In: *The Journal of Symbolic Logic* 1.3 (1936), pp. 87–91.

[Sch61] Joseph R. Schoenfield. “The problem of predicativity”. In: *Essays on the Foundations of Mathematics*. Ed. by Y. Bar-Hillel et al. Magnes, 1961, pp. 132–139.

[She84] Saharon Shelah. “On Logical Sentences in PA”. In: *Logic Colloquium ’82*. Ed. by G. Lolli, G. Longo, and A. Marcja. Vol. 112. Studies in Logic and the Foundations of Mathematics. Elsevier, 1984, pp. 145–160.

[Szu06] Matthew Szudzik. *An Elegant Pairing Function*. 2006. URL: `http://szudzik.com/ElegantPairing.pdf`.

[Vää21] Jouko Väänänen. “Second-order and Higher-order Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2021. Metaphysics Research Lab, Stanford University, 2021.

[Vau01] Robert L. Vaught. *Set Theory: An Introduction*. 2nd Ed. Springer, 2001.

[Wei87] Klaus Weihrauch. *Computability*. EATCS Monographs on Theoretical Computer Science 9. Springer, 1987.

[WR10] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910.

[YA16] Adam Yedidia and Scott Aaronson. “A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory”. In: *Complex Systems* 25.4 (2016).

[Yud08] Eliezer Yudkowsky. *Cartoon Guide to Löb’s Theorem*. 2008. URL: `https://de.scribd.com/doc/4844564/Cartoon-Guide-to-Lob-s-Theorem`.

[Zer10] Ernst Zermelo. *Ernst Zermelo - Collected Works/Gesammelte Werke: Volume I/Band I - Set Theory, Miscellanea/Mengenlehre, Varia*. Schriften der mathematisch-naturwissenschaftlichen Klasse. Springer, 2010, pp. 114–119.