Electronic Colloquium on Computational Complexity, Report No. 86 (2025)

An Introduction to Feasible Mathematics and Bounded Arithmetic for Computer Scientists

Jiatu Li[1]

June 30, 2025

[1] jiatuli@mit.edu. Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology. Cambridge, Massachusetts, USA.

ISSN 1433-8092

Abstract

This note gives an in-depth discussion on feasible mathematics and bounded arithmetic with a focus on Cook's theory PV (STOC'75). We present an informal characterization of PV based on three intuitive postulates and formulate the Feasible Mathematics Thesis, which asserts the equivalence between this informal framework and the formal system PV. To support this thesis, we provide a detailed exposition demonstrating how advanced programming and reasoning tools can be systematically constructed within the seemingly weak theory PV.

Copyright Notice

Preface

The main purpose of the note is to address two common and closely related questions that people frequently ask about bounded arithmetic.

“People” include both computer scientists and logicians

The first question is the *intuition* behind the theories people are studying. Successful candidates of mathematical foundation, such as the ZFC set theory and Peano Arithmetic, win their positions through an extensive line of literature and education practice that clearly establishes an intuition behind them. Similarly, the concept of *computation* as well as the *efficiency* of computation are clearly established by both the mathematical work of Turing (and many others) as well as the education and engineering practice in the past century. Compared to these well-established concepts, the intuition behind bounded arithmetic and more generally the concept of *feasible mathematics* is still relatively unclear.

The second question, which is sometimes criticism, is about the writing standard for formalizations in bounded theories. The common practice for writing proofs in a bounded theory T (see, e.g., [Oja04, Jer07, Pic15, CLO24]) is to say that “the following proof is done in T” and simply use natural language. However, this could be dangerous as “the provability of a statement φ in a bounded theory T” is a precise mathematical statement, but the common practice of writing proofs omits tons of details, including technical details in both logic and computer science. Such a writing standard is arguably reasonable for more popular concepts such as ZFC set theory or theory of computation, as clear intuition has been established and technical details have been tested over time by a long line of research, education, and engineering work. For bounded arithmetic and the concept of feasible mathematics, however, many people (including I) tend to feel much less confident to only write highly informal proofs.

I’ve heard this question from both logicians and computer scientists :(

One may argue that writing informal proofs in formalizations is not a big deal as even if there is indeed something that cannot be formalized, we could revise the theories according to the informal concepts we aim to capture. After all, we are mostly interested in the informal concepts that the bounded theories aim to capture rather than the particular formalizations of the theories. However, the excuse may easily lead to a catastrophe when we need to combine *formalizations* and *logical analysis results* of the same theory, as after revising the theory we will also need to perform a logical analysis accordingly, which may not necessarily be possible after the revision. This becomes an urgent matter as there have been examples where such combination is necessary: Recent works in cryptography (see, e.g., [JJ22, JKLV24, JKLM24]) rely on the combination of the propositional translation of PV as well as formalizations of cryptographic primitives in PV, and consequences in bounded reverse mathematics (see, e.g., [CLO24, LLR24, AT24]) rely on the combination of witnessing theorems and certain formalizations.

This note attempts to address both questions for Cook’s theory PV [Coo75] that corresponds to polynomial-time computation. On the one hand, we introduce the concept of feasible mathematics (that PV tries to capture) by three informal postulates and propose the Feasible Mathematics Thesis, which claims that the informal concept is exactly captured by the theory PV (see Chapter 1). On the other hand, we try to justify the Thesis by writing in (a disgusting level of) detail how we could formalize basic and advanced programming functionalities in PV, as well as how we could translate informal feasible proofs (captured by the three postulates) into formal proofs in PV. As the byproduct, the detailed technical investigation yields convenient meta-theorems that may be a good candidate for writing "pseudo-proofs" for PV (see, e.g., Chapter 4).

Most theorems proved in this note are either implicit in literature or widely believed to be true (without written proofs). Here is an incomplete list of references: A large portion of Chapter 2 is an exposition of the formal proofs in the appendix of [CU93]; the main result of Chapter 3 is claimed in [Coo75] without a written proof; and many examples in Chapter 4 can be seen as extensions of, e.g., [Bus85, Coo90]. Detailed historical remarks can be found at the end of each chapter. It should be mentioned that a large portion of the technical work is inspired by the engineering practice in the community of formal verification; for instance, the recursion and induction meta-theorems for lists (see Theorem 4.1.3 and Corollary 4.1.6) are inspired by the recursion and induction rules in proof assistants such as Coq [PCG$^+$10] or Lean [DMKA$^+$15].

In addition to the three-postulate interpretation of informal feasible mathematics, the note contains a few "new" theorems and perspectives that may be worth noting for readers who have been familiar with bounded arithmetic:

- In Section 1.3.2, we show (informally) that the correctness of Dijkstra's SSSP algorithm can be proved feasibly. Interestingly, the proof is different from the standard textbook proof (see, e.g., [CLRS09]) that directly performs an induction on number of rounds to prove the correctness of the algorithm.

- In Chapter 3, we define a theory $\mathsf{PV}\text{-}\mathsf{PL}$ according to Cook's original formulation of PV_1 [Coo75], which uses the universal fragment of first-order logic. The name PV_1 is now used to denote the theory axiomatized by PV equations in first-order logic rather than the universal fragment (see, e.g., [Kra19]). It is mentioned in [Coo75] (without a formal proof) that $\mathsf{PV}\text{-}\mathsf{PL}$ is conservative over PV. Indeed, we prove a slightly stronger "translation theorem" that provides an explicit embedding of $\mathsf{PV}\text{-}\mathsf{PL}$ into PV, which may be of independent interest.

- In Chapter 4, we prove two powerful meta-theorems that essentially allow us to manipulate *lists* just as a native data structure. This could be widely used as a standard functionality in later formalizations in PV. We also show how to simulate imperative programming languages and a Hoare logic over it in PV. This could be useful in later formalizations, as Hoare logic (in particular, the use of loop invariants) is arguably more natural to computer scientists and programmers compared to PV.

- In Chapter 5, we formulate the propositional translation theorem for PV (see Theorem 5.1.2) as an *equivalence* between PV proofs and PV-provably existence of propositional proofs, rather than only a translation from PV proofs to propositional proofs. This makes the propositional translation not only a technical result but a formal mean of *interpretation of* PV *provability.* This is closer to its counterpart $\mathsf{P} = \mathsf{P}\text{-}\mathsf{uniform}\ \mathsf{P}_{/\mathsf{poly}}$ in computational complexity theory, whereas the standard formalization is similar to the weaker version $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$.

Throughout the note, we stick to the original formulation of PV as an *equational theory* [Coo75] that does not allow quantifiers and logical connectives. We do not include arithmetic functions such as addition and multiplication as basic functions and do not include their definition equations as axioms. Moreover, the universe of the standard model of PV (see Chapter 2) is defined as the set of binary strings $\{0,1\}^*$ rather than natural numbers. The rationale is to emphasize that there is no need for special treatment for natural numbers and arithmetic operations: Natural numbers are encoded by binary strings just as other data structures, and with respect to the encoding we use, arithmetic operations are implemented (with PV provable correctness) by the fundamental programming functionalities provided by PV just as other programs. I want to make sure that we are not trying to add unnecessary functions into the theory — if we do that for addition and multiplication, how can we believe that we will not do that again and again for more complicated algorithms?

Finally, the note only covers the theory PV and does not consider many classical results in bounded arithmetic, including theories for even smaller complexity classes (e.g. NC^1 or TC^0), the first-order theories of bounded arithmetic, and model-theoretic approaches. Most of the missing results can be found in standard textbooks [Bus85, Kra95a, Kra19, CN10]. The survey [Oli24] also serves as a good reference for more recent results in bounded arithmetic.

Acknowledgement. The note is based on lecture notes of a reading group at MIT in 2024 Fall (see `https://github.com/ljt12138/Bounded-Arithmetic-Reading-Group`). I am grateful for the support from my advisor, Ryan Williams, and colleagues, including Marco Carmosino, Lijie Chen, Matthew Hong, Zhengzhong Jin, Oliver Korten, Surya Mathialagan, Hanlin Ren, Yiding Zhang, and many others. I would also like to thank all participants for insightful discussions and for catching and fixing a bug I made in Chapter 4; in particular, I am grateful to Professor Samuel Buss, who attended several meetings while he was visiting Boston. I would also like to thank Ján Pich and Igor C. Oliveira for their helpful comments on early drafts of the notes.

This note is a part of the author's master thesis archived in MIT library.

Some parts of the work are done when the Author is visiting NTT Research, Inc., as an intern. Some parts of the work are done when the author is supported by the National Science Foundation under Grant No. CCF-2127597. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

Chapter 1

Informal Feasible Mathematics

The concept of *feasibility* in computation has been extensively studied in the past century of the development of theoretical computer science. In theoretical computer science, it is generally accepted that a problem is feasible if and only if there is a polynomial-time algorithm that solves the problem — known as the *extended Church-Turing thesis* [Coo90].

Remarkably, this definition is widely believed to be independent of the underlying model of computation[1]. As far as we know, most natural computation models — single-tape Turing machines, multi-tape Turing machines, RAMs, or standard programming languages — are equivalent up to a fixed polynomial time overhead (see, e.g., [AB09, Chapter 1]).

The primary purpose of this chapter is to provide a conceptual description of *feasible mathematical proofs* that bounded arithmetic aims to capture. We will define an informal theory of feasible mathematics concerning the standard computation model, namely polynomial-time computable functions, which is formalized by Cook's theory PV [Coo75].

1.1 Background: Constructive Mathematics

Rather than viewing mathematics as a "World of Ideas" of Plato that is eternal, unchanging, and independent of human observation, constructive mathematicians assert that mathematical objects must be constructed by human intelligence. We quote from the book of Bishop [Bis67]:

"Platonism"

> "The positive integers and their arithmetic are presupposed by the very nature of our intelligence and, we are tempted to believe, by the very nature of intelligence in general. The development of the positive integers from the primitive concept of the unit, the concept of adjoining a unit, and the process of mathematical induction carries complete conviction. In the words of Kronecker, the positive integers were created by God."

[1] An exception is that quantum computers may be stronger.

The standard interpretation of constructive mathematics is due to Brouwer, Heyting, and Kolmogorov (or the BHK interpretation). For instance, universal and existential quantifiers are interpreted as follows:

1. The existence quantifier should be interpreted as a *procedure* that effectively constructs the quantified object, instead of merely its "existence" as an object in the mathematics "World of Ideas". In particular, if $\varphi(x, y)$ is a formula such that $\forall x\ \exists y\ \varphi(x, y)$ is provable, there must be a *procedure* that (given n that substitutes x) effectively construct an m that substitutes y as well as a *proof* of $\varphi(n, m)$.

2. A *proof* of a universally formula $\forall x\ \varphi(x)$ must provides a *procedure* that effectively generates a *proof* of $\varphi(n)$ given any n that substitutes the variable x.

Interpretation of "effective procedures". The interpretation of "effective procedures" varies. For instance, intuitionists suggest that the law of excluded middle $\varphi \vee \neg\varphi$ is troublesome and should be excluded (see [BPI22] for a comprehensive survey on variants of constructivism).

From a computer science perspective, it is natural to interpret effective procedures as feasible algorithms, or polynomial-time algorithms, under the extended Church-Turing thesis. This yields the basic notion of *feasible mathematics*, or *feasibly constructive mathematics*, as an interpretation of constructive mathematics by treating the "effective procedures" as polynomial-time algorithms. In other words, we assume a minimal mean of intelligence, or the intelligence of a humble computer scientist that is skeptical of anything that cannot be constructed in polynomial time.

As the adversary model in modern cryptography.

Example 1.1.1. Under this interpretation, the exponentiation function cannot be defined in feasibly constructive mathematics; that is, the sentence $\forall x\ \exists y\ \log_2(y) = x$ is not provable. This is because the function outputting y given x is not a polynomial-time function (in its input length).

Example 1.1.2. Similarly, the following version of the pigeonhole principle is unlikely to be feasibly provable: For every definable function $f : \{0,1\}^n \to \{0,1\}^{n-1}$, there are distinct strings $x, y \in \{0,1\}^n$ such that $f(x) = f(y)$. Suppose, towards a contradiction, that such principle is feasibly provable, then a polynomial-time algorithm must exist that, given f, outputs $x \neq y$ such that $f(x) = f(y)$. This breaks a widely believed cryptographic primitive: collision-resistant hash function.

The two examples above seem problematic: As a cost of being humble, computer scientists, or, *feasible mathematicians*, is unable to even define exponentially long strings, or understand the pigeonhole principle. It turns out that feasible mathematics is "strong and weak": Though it cannot prove the pigeonhole principle, it is capable of proving results like the PCP theorem [Pic15] and many known complexity lower bounds (see, e.g., the table in [PS21]). One exception is that the $\Omega(n^2)$ lower bound for deciding Palindrome on single-tape Turing machines [CLO24] is unlikely to be feasibly provable, as it requires a form of pigeonhole principle.

This seems like a slippery slope for those who are skeptical about working with (say) ZFC set theory and the existence of large cardinals.

Interpretation of "proofs". The "effective procedures" in the BHK interpretation manipulate "proofs" of a certain form, which is yet to be specified. The standard definition of "proofs" in theoretical computer science is the complexity class NP, which requires the following two properties:

1. The length of the proof is polynomial in the length of the statement.
2. The proof is *verifiable* by a polynomial-time algorithm.

Arguably, these two properties are necessary. After all, feasible mathematicians cannot construct exponentially long strings (see Example 1.1.1) and do not trust verification procedures that are not polynomial-time algorithms. Nevertheless, it is unclear what is the interpretation of *verification* in the second property, which we will explore from scratch.

Indeed, we will spend three chapters on exploring it :)

1.2 Informal Postulates of Feasible Mathematics

Our goal is to propose a set of informal postulates for feasible mathematics by exploring questions of the form "Is X considered feasible" — X consists of functions, axioms, and inference rules. In other words, we consider informally what functions, axioms and inference rules can be used by feasible mathematicians.

These questions formulate the underlying philosophy in the construction of Cook's theory [Coo75]. We also note that the constructions of other bounded theories, e.g., Parikh's theory [Par71] and Buss's theories [Bus86], follow similar intuition but use different models of computation.

1.2.1 Postulate 0: Finitism

Before stating the three postulates, we clarify that our definition of feasible mathematics is *finitistic.* Namely, one can neither define nor quantify over infinite sets of any kind. Subsequently, all mathematical objects in feasible mathematics should be able to be encoded by either natural numbers of Boolean strings of finite length.

Indeed, I identify myself as a finitist.

1.2.2 Postulate of Feasible Functions

We start with the following question:

Question 1.2.1. Is the addition of two numbers definable in feasible mathematical proofs?

Arguably, the answer to this question should be *yes.* On the one hand, the addition of two numbers is clearly a feasible operation. On the other hand, there is not much mathematics we can do if we are not even allowed to define and reason about addition. Moreover, comparably simple functions, such as the multiplication of two numbers, the bit-length function $|\cdot|$, etc., should be allowed.

Please consult a primary school student if you have any questions :)

Extending Question 1.2.1, we further ask:

Question 1.2.2. Should feasible mathematicians be able to define every feasible function?

Note that predicates (languages) are just Boolean-valued functions.

Example 1.2.1. It has been open for decades whether a *deterministic* polynomial-time algorithm could output an n-bit prime number given 1^n. The Cramér's conjecture [Cra36], which is true under the generalized Riemann Hypothesis, implies that the following simple algorithm suffices:

- Starting from $N \leftarrow 2^n$, we call the AKS primality testing algorithm to check whether N is a prime number. If so, the algorithm halts and outputs N; otherwise, it sets $N \leftarrow N + 1$ and continues.

The existence of an n-bit prime follows from the Bertrand-Chebyshev theorem.

Although the algorithm in Example 1.2.1 (say, represented by a Turing machine) is likely to be feasible, it is consistent with our knowledge that:

1. The Cramér's conjecture could be *wrong*, in which case we introduce an infeasible function in feasible mathematics.
2. The Cramér's conjecture could be *true* but does not have a *feasible proof*.

To avoid putting the foundation of feasible mathematics in danger, we should refrain from using the function in feasible mathematics.

Nevertheless, functions that are not only feasible but also *clearly demonstrates* their feasibility through its construction should be definable in feasible mathematics. For instance, one would generally agree that functions constructed in the following ways clearly demonstrate their feasibility:

It is confusing here, but should be more clear in subsequent examples.

- The composition of feasible functions is still feasible. That is, for any integer $k \geq 1$, if $f(x_1, \ldots, x_k)$ and $g_1(y_1), \ldots, g_k(y_k)$ are feasible functions, $f(g_1(y_1), \ldots, g_k(y_k))$ is also a feasible function.
- For any integer $c \geq 1$, any function $f_M(x)$ defined by a Turing machine $M(x)$ with a clock that makes the machine halt after $|x|^c$ steps is feasible.
- Suppose that f is feasible, the function $\sigma_f(n)$ that either outputs the smallest number $x < |n|$ such that $f(x) \neq 0$, or outputs $|n|$ if $\forall x < |n|\ f(x) = 0$, is a feasible function.

Note that $x \mapsto |x|^c$ is feasible, as it is a composition of the bit-length function and multiplication.

These observations motivate our first informal postulate:

Postulate 1 (Postulate of Feasible Functions)**.** Functions that are feasible and clearly demonstrate their feasibility through their constructions are definable in feasible mathematics.

We stress that the exact meaning of the phrase "clearly demonstrates their feasibility" can be interpreted in different ways, which will probably lead to multiple formal definitions of feasible mathematics.

We will go back to this later.

Remark 1.2.1. We say in Postulate 1 that functions with clear demonstrations of feasibility are allowed in feasible mathematics. To formalize meaningful mathematical statements, we also need to specify the *atomic predicates* that are allowed to be used. The equality predicate ("="), for instance, should certainly be allowed. Moreover, for any predicate $P(\vec{x})$ (e.g. $\leq$, $\wedge$) decidable by a feasible algorithm that can clearly demonstrate its feasibility, we can define a function $g_P(\vec{x})$ that outputs 1 (resp. 0) when the property $P(\vec{x})$ holds (resp. does not hold), and formalize the predicate as $P(\vec{x})$ as $g_P(\vec{x}) = 1$. Therefore, without loss of generality, we can work with only one atomic predicate "=".

1.2.3 Postulate of Definition Axioms

We then consider what axioms are allowed in feasible mathematics. To start with:

Question 1.2.3. Should a basic rule such as $n + (m + 1) = (n + m) + 1$, a defining equation for addition, be accepted as an axiom in feasible mathematics?

As by Postulate 1 we are allowed to work with functions that are far more complicated than addition and multiplication, it is natural to further include definition axioms of those functions as the axioms so that we could *reason* about the functions.

Recall that a feasible function f is definable only if it clearly demonstrates its feasibility through its constructions. This specific means of "construction" should be a set of mathematical facts that uniquely specify the function f; for instance, it could be a set of equations that inductively defines f (see Example 1.2.2), or the code of a Turing machine computing f with an explicit time bound. Since we accept that these mathematical facts as "clear demonstration" of the feasibility of f, these facts should be able to be used (i.e. be axioms) to reason about f. These mathematical facts are considered the "definition axioms" of the function f.

"we" := feasible mathematicians.

The following illustrates how an inductive definition yields a set of concrete axioms for a feasible function.

Example 1.2.2. Consider a function f that is inductively defined as

$$f(0) := 0 \tag{1.1}$$

$$f(2n) := f(n) + 2n \tag{1.2}$$

$$f(2n+1) := f(n) + 2n + 1 \tag{1.3}$$

Arguably, this is a feasible function that clearly demonstrates its feasibility through this particular inductive definition. In this case, Equations (1.1) to (1.3) are the definition axioms of f.

Note that in each recursive call to evaluate f the length of the input is reduced by 1.

This leads to the second postulate of feasible mathematics.

Postulate 2 (Postulate of Definition Axioms)**.** For any function admissible under Postulate 1, the mathematical facts constituting the construction that serves as the "clear demonstration of its feasibility" are accepted as axioms in feasible mathematics.

1.2.4 Postulate of Structural Induction

The definition axioms characterize the *local* behavior of definable functions, which are insufficient for establishing global properties. Consider the simple task of proving the inequality $\forall x \in \mathbb{N}\ f(x) \leq 2x$ for the function f in Example 1.2.2. Note that as $f(x)$, $x \mapsto 2x$, and the comparison predicate "$\leq$" are all feasible and can arguably demonstrate their feasibility under suitable definitions, this sentence can be stated in feasible mathematics.

It is unclear how we should prove it only from their definition axioms. If we consider the last bit of n in its binary encoding, we need to prove that $f(0) \leq 0$ (which is easy), and that

$$\forall x\ f(2x) \leq 2 \cdot 2x, \quad \forall x\ f(2x+1) \leq 2 \cdot (2x+1).$$

We can apply the definition axiom of f to unfold them as:

$$\forall x\ f(x) + 2x \leq 2 \cdot 2x, \quad \forall x\ f(x) + 2x + 1 \leq 2 \cdot (2x+1).$$

Assume that basic arithmetic that feasible mathematicians know basic arithmetic, which is a minor assumption, it suffices to prove that

$$\forall x\ f(x) \leq 2x, \quad \forall x\ f(x) \leq 2x + 1.$$

Attempting to prove the inequality $f(x) \leq 2x$ by unfolding the definition and analyzing the last bit of x leads to a circular situation: to prove $f(2x) \leq 4x$, one must already know $f(x) \leq 2x$. Thus, the definition alone cannot advance the argument.

This hints that we may need to include another tool in mathematics: the principle of mathematical induction. For instance, the inequality above can be proved if we allow an induction principle on the length of x in its binary encoding:

- (Base): $f(0) \leq 2 \cdot 0$, which is true as $f(0) = 0$, $2 \cdot 0 = 0$, and $0 \leq 0$.

- (Induction Case 1): $f(x) \leq 2x \to f(2x) \leq 2 \cdot 2x$. Assume that $f(x) \leq 2x$, we will prove $f(2x) \leq 2 \cdot 2x$. By the definition axiom (1.2), we know that $f(2x) = f(x) + 2x$, then
$$f(2x) = f(x) + 2x \leq 2x + 2x = 2 \cdot 2x.$$

Exercise: feasibly prove $b \leq c$ implies $b + a \leq c + a$.

- (Induction Case 2): $f(x) \leq 2x \to f(2x+1) \leq 2 \cdot (2x+1)$. Assume that $f(x) \leq 2x$, we will prove $f(2x+1) \leq 2 \cdot (2x+1)$. By the definition axiom (1.3), we know that $f(2x+1) = f(x) + 2x + 1$, then
$$f(2x+1) = f(x) + 2x + 1 \leq 2x + (2x+1) \leq (2x+1) + (2x+1) = 2 \cdot (2x+1).$$

Should we accept a general form of induction principle in feasible mathematics? Take the proof above as an example. As we proved that $\forall x\ f(x) \leq 2x$, by the constructivism interpretation of the universal quantifier (see Section 1.1), there must be an efficient "procedure" that generates a verifiable "proof" of $f(x) \leq 2x$ given n. Moreover, by our interpretation of "feasible mathematics", the "procedure" must be a feasible (i.e. polynomial-time) algorithm, and the "proof" must be of polynomial length that is in some sense "verifiable" by a polynomial-time algorithm (i.e. an NP-style proof).

We claim that the induction principle is feasible as long as the predicate for induction is "verifiable": Given an input m that substitutes the induction variable, there is a feasible (i.e. polynomial-time) algorithm that verifies the predicate. In this example, the predicate is $f(x) \le 2x$ and we are performing induction on the variable x; given any m, we can certainly evaluate $f(x)$ and check whether it is smaller than $2x$ feasibly. In particular, if we work with the only predicate "$=$", this is equivalent to saying that the predicate for induction must be of form $g(x) = h(x)$ for functions g, h that clearly demonstrates their feasibility.

Unfortunately, we are unable to provide a convincing clarification for this claim until we formally define the "proof" and the mean of "verifiability". Since the induction is performed over the length of the binary representation of x, the "chain of reasoning" is of length $\mathsf{poly}(|x|)$. Therefore, for any concrete input m that substitutes x, there is a convincing proof of length $\mathsf{poly}(|m|)$ following the chain of reasoning that clearly demonstrates the fact "$f(m) \le 2m$".

We will then give the third (and the last) postulate:

Postulate 3 (Postulate of Structural Induction)**.** Let $g(x)$ and $h(x)$ be functions admissible under Postulate 2. Then induction over the binary representation of x is allowed in feasible mathematics to prove statements of form $g(x) = h(x)$.

Recall that we are working with only one predicate "$=$".

Remark 1.2.2. The postulate permits induction on x with additional parameters $\vec{w}$ fixed. Let $g(\vec{w}, x)$ and $h(\vec{w}, y)$ be functions admissible under Postulate 2. Then the induction principle states that for any *fixed* $\vec{w}$, if

- $g(\vec{w}, 0) = h(\vec{w}, 0)$, and
- $g(\vec{w}, x) = h(\vec{w}, x)$ implies $g(\vec{w}, 2x + b) = h(\vec{w}, 2x + b)$ for $b \in \{0, 1\}$,

then $g(\vec{w}, x) = h(\vec{w}, x)$ for any $\vec{w}, x$.

This induction rule is different from the induction principle with universally generalized parameters $\vec{w}$, i.e., structural induction on x to prove the statement "$\forall \vec{w}\ g(\vec{w}, x) = h(\vec{w}, x)$". The induction principle with universally generalized parameters is *infeasible* as the statement "$\forall \vec{w}\ g(\vec{w}, x) = h(\vec{w}, x)$" is not feasibly verifiable.

For the example above, one can think of $g(x)$ as the function checking whether "$f(x) \le 2x$", and $h(x) = 1$. Note that we also call the induction principle a "structural induction" as it is a structural induction over the definition of (the binary representation of) numbers.

To streamline discussion, we now refer to any function that clearly demonstrates its feasibility through a specific construction simply as a *feasibly constructible function*. The particular construction that justifies feasibility will be called the *feasible construction* of the function.

1.3 Examples of Feasible Proofs

To understand the power of feasible mathematics with the three postulates we have, we provide two examples: A "stronger" induction principle and the correctness of Dijkstra's single-source shortest path algorithm.

1.3.1 A Stronger Induction Principle

Question 1.3.1. Suppose that f, g are functions that we are allowed to talk about (in particular, it is a feasible function), and $\varphi(n)$ is the sentence "$f(n) = g(n)$". Is the following induction rule, i.e.,

$$\varphi(0) \land \Big(\forall n \ (\varphi(n) \to \varphi(n+1))\Big) \to \forall n \ \varphi(n)$$

considered feasible?

This induction principle is not formulated as the structural induction over a feasibly defined function. In particular, the "chain of reasoning" from $\varphi(0)$ and $\varphi(n) \to \varphi(n+1)$ to $\forall n \ \varphi(n)$ seems to be exponentially long. That is, for every n, we need to derive $\varphi(n)$ from:

$$\Big\{\varphi(0), \quad \varphi(0) \to \varphi(1), \quad \varphi(1) \to \varphi(2), \quad \dots, \quad \varphi(n-1) \to \varphi(n)\Big\},$$

which consists of $n + 1 = |n|^{\omega(1)}$ formulas.

Here, we will demonstrate that this induction rule is indeed feasible.

Theorem 1.3.1 (informal). *Let f, g be feasibly constructible functions, and $\varphi(n) :=$ "$f(n) = g(n)$". It is feasibly provable that*

$$\varphi(0) \land \Big(\forall n \ (\varphi(n) \to \varphi(n+1))\Big) \to \forall n \ \varphi(n). \tag{1.4}$$

Instead of considering Equation (1.4) directly, we will work with the following sentence that is logically equivalent to it:

Exercise: prove the logical equivalence.

$$\forall n \ \exists m \ (\varphi(0) \land \neg\varphi(n) \to \varphi(m) \land \neg\varphi(m+1)). \tag{1.5}$$

This is true as long as we stick to the standard first-order predicate logic to formulate feasible mathematics.

Recall that the constructivism interpretation requires that a proof of $\forall x \ \exists y \ \varphi(x, y)$ must provide an effective procedure that finds y such that $\varphi(x, y)$ given x as well as a proof of $\varphi(x, y)$. This means that we need to find a feasible function $h(n)$ and prove feasibly that

$$\forall n \ (\varphi(0) \land \neg\varphi(n) \to \varphi(h(n)) \land \neg\varphi(h(n) + 1)). \tag{1.6}$$

The function $n \mapsto h(n)$ as an algorithmic task means that given an n such that $\varphi(0)$ is true and $\varphi(n)$ is false, we need to find an m such that $\varphi(m)$ is true and $\varphi(m + 1)$ is false. The algorithm should be feasibly constructible; in particular, it should run in time $\mathsf{poly}(|n|)$.

Indeed, it is not hard to see that a simple binary search algorithm works (see Algorithm 1).

Equivalently, one may implement the binary search by a recursive (instead of iterative) algorithm. In either case, the algorithm *clearly* runs in polynomial time through its construction, as the quantity $r - \ell$ is halved in each recursive call (iteration). Therefore the function computed by this algorithm, denoted by $h(n)$, is feasibly constructible.

```
Input: $n$ such that Equation (1.5) holds
Output: $m$ such that $\varphi(m) \wedge \neg\varphi(m+1)$
$\ell \leftarrow 0, r \leftarrow n$;
while $\ell + 1 < r$ do
    $m \leftarrow \lfloor (\ell + r)/2 \rfloor$;
    if $\varphi(m)$ is true then
        $\ell \leftarrow m$
    else
        $r \leftarrow m$
    end
end
return $\ell$
```

Algorithm 1: Binary Search for $n \mapsto h(n)$

It remains to demonstrate that Equation (1.6) is feasibly provable. The only available tools are the postulate of definition axioms and the postulate of structural induction. A straightforward idea is to prove by induction over the "while" loop on some properties regarding ℓ and r.

If this is not clear to you, think of the most natural way to prove it without worrying about the feasibility.

The standard approach to perform induction over Algorithm 1 is to prove a *loop invariant*: $\varphi(\ell) \wedge \neg\varphi(r) \wedge r > \ell$. This is true before the program enters the loop, and (by the definition axioms of the operations inside the loop) if the property holds in the i-th iteration, it also holds in the $(i+1)$-th iteration. Finally, when we leave the loop, we will have:

$$\varphi(\ell) \wedge \neg\varphi(r) \wedge r > \ell \wedge \ell + 1 \geq r$$

which implies immediately that $r = \ell + 1$, and thus ℓ (the output $h(n)$ of Algorithm 1) satisfies $\varphi(\ell) \wedge \neg\varphi(\ell+1)$, i.e., Equation (1.6). This proof realizes the informal discussion above that the length of the "chain of reasoning" needs to be a polynomial of the input length.

Equivalently, if we formulate the algorithm recursively, we can introduce a variable z indicating that we will perform $|z|$ iterations of the binary research, and prove by induction on the binary representation of z that the invariant holds.

We stress that there is one additional property to be verified: the loop invariant (i.e. the property in structural induction) must be formalizable in the form $f(x) = g(x)$ for some feasibly constructible functions $f(x), g(x)$. Since both $\varphi(\ell)$, $\neg\varphi(r)$, and $r > \ell$ are pretty easy to be checked by feasibly constructible functions, we can easily construct some feasible $f(x)$ (where x encodes the pair (ℓ, r)) that checks the loop invariant, and formalize the loop invariant as $f(x) = 1$. All the rewinding (e.g. parse x as a pair and the evaluation of "$\varphi(\ell), \varphi(r), r > \ell$") should be feasibly provable from the definition axioms under suitable formalization. Therefore, we conclude (from the informal postulates) that Theorem 1.3.1 should be true.

1.3.2 Correctness of Dijkstra's Algorithm

Next, we consider a more complicated algorithmic example: The correctness of Dijkstra's algorithm for single-source shortest path.

Suppose that a graph $G = (V, E)$ is encoded by its adjacency matrix $E \in \{0, 1\}^{n\times n}$, where $V = [n]$. Dijkstra's algorithm is given n, E, as well as a node $s \in [n]$, and outputs a vector $d \in [n]^n$ such that either $d_j \le n - 1$ is the length of the shortest path from s to j, or $d_j = \mathsf{INF} := n$ indicating that s and j are disconnected.

Welcome back to Algorithm 101!

Recall that Dijkstra's algorithm works as follows. At the beginning, we initialize $d_s \leftarrow 0$, $d_j \leftarrow n^2$ for any $j \ne s$, and a set $S \leftarrow [n]$. Let p be a vector of paths where p_s is initialized as the path $s \mapsto s$ of length 0. For $i = 1, 2, \ldots, n$, we find the node $u \in S$ with smallest d_u (breaking tie by node id), update

$$d_v \leftarrow \min(d_v, d_u + 1)$$

for each neighbor v of u that is not in S, and update $S \leftarrow S \setminus u$. If $d_v > d_u + 1$ before the update, we further update

$$p_v \leftarrow \text{“}p_u \mapsto v'',$$

i.e., the candidate shortest path from s to v is obtained by extending the path p_u from s to u with the edge (u, v).

Let $\mathsf{Dijkstra}(V, E, s)$ be the feasibly constructive function that simulates Dijkstra's algorithm and outputs the vectors d and p. Arguably, this is a feasibly constructible function, as it terminates in n rounds and all internal variables are of length $\mathsf{poly}(n)$. Let $(d, p) \leftarrow \mathsf{Dijkstra}(V, E, s)$. We can formalize the correctness of Dijkstra's algorithm by two statements:

- For any $j \in V$, if $d_j \le n - 1$, p_j is a path from s to j of length d_j.
- For any $j \in V$ and any path p from s to j, the length of p is at least d_j.

We omit the detail of defining $\mathsf{Path}(\cdot)$, which should be straightforward in a reasonable model of computation that should be supported by feasible mathematics.

These two properties only consist of a universal quantification over nodes or paths, which can be formalized in feasible mathematics by an equation on feasibly constructive functions. For instance, to formalize the second property, we need to define a function $\mathsf{Path}(p, s, j, d, V, E)$ to verify that p is a path from s to j in $G = (V, E)$ of length at most d, which is arguably feasibly constructive and thus allowed by the postulate of feasible functions.

We will show that both statements can be proved in (informal) feasible mathematics. Perhaps interestingly, the proof for the latter statement differs slightly from the usual textbook proof of the correctness.

Existence of path. First, we (feasibly) prove that if $d_j \le n - 1$, a path of length d_j is given by p_j. Fix any graph $G = (V := [n], E)$, $s \in [n]$, and $j \in [n]$. Recall that Dijkstra's algorithm consists of n iterations. Let $d^{(i)}$, $S^{(i)}$, and $p^{(i)}$ be the values of corresponding variables at the end of the i-th iteration. We prove the following statement by induction on i:

- For every node $u \in V$, if $d_u^{(i)} \le n - 1$, $p_u^{(i)}$ is a path from s to u of length $d_u^{(i)}$.

Note that this suffices, as for $i = n$, $d^{(i)}$ and $p^{(i)}$ are exactly the output of the Dijkstra's algorithm. Also, as the predicate in the induction is checkable by a feasibly constructive algorithm that enumerates every $u \in V$, this induction is available in feasible mathematics either by Section 1.3.1 or by the third postulate — the "total length of the chain of reasoning" of the induction is merely $\mathsf{poly}(n)$.

Clearly, for $i = 0$, this statement is true as the only node u with its distance label smaller than n is the source s, where $p_s^{(0)}$ has been updated to the path $s \mapsto s$ of length 0. It remains to show that if the statement implies the statement obtained by replacing i to be $i+1$. Let $u \in S^{(i)}$ be the node with smallest $d_u^{(i)}$ selected by the Dijkstra's algorithm in the $(i+1)$-th iteration. Notice that for every $v \in V$, $d_v^{(i+1)} \le d_v^{(i)}$, and:

- If $d_v^{(i+1)} = d_v^{(i)}$, $p_v^{(i+1)} = p_v^{(i)}$. Subsequently if $d_v^{(i+1)} \le n-1$, we know by the induction hypothesis and $d_v^{(i)} = d_v^{(i+1)} \le n-1$ that $p_v^{(i)} (= p_v^{(i+1)})$ is a path from s to v of length $d_v^{(i)} (= d_v^{(i+1)})$.
- Otherwise, we know that $d_v^{(i+1)} = d_u^{(i)} + 1 \le n-1$ and v is a neighbor of u. In such case, we know that the algorithm updates $p_v^{(i+1)}$ by extending $p_u^{(i)}$ with the edge (u,v). As $d_u^{(i)} \le n-2 \le n-1$ and the induction hypothesis, we know that $p_u^{(i)}$ is a path from s to u of length $d_u^{(i)}$; therefore, $p_v^{(i+1)} = p_u^{(i)} \mapsto v$ is a path from s to v of length $d_u^{(i)} + 1 = d_v^{(i+1)}$.

The proof above (i.e. the "chain of reasoning") follows closely the computation of Dijkstra's algorithm; if we formalize the algorithm carefully, it could be implemented using the induction postulate as well as the definition axioms of the algorithm.

Shortness: the standard infeasible proof. To prove that every path from s to j is of length at least d_j, the standard textbook proof (see, e.g., [CLRS09, Section 24.3]) essentially uses an induction principle on i to prove that:

- for every node $u \in V$, $d_u^{(i)}$ is the shortest path from s to u via $V \setminus S^{(i)}$ (or INF if unconnected via $V \setminus S^{(i)}$).

For simplicity, we call this the *shortest path condition*.

Recall that the induction postulate only allows induction on feasibly verifiable property that can be encoded by equations of feasibly constructive functions. The property that "$d_u^{(i)}$ is the shortest path from s to u via $V \setminus S^{(i)}$" implies that "for any path p from s to u via $V \setminus S^{(i)}$, the length of p is at least $d_u^{(i)}$". The latter property can be "checked" by running Dijkstra's algorithm; however, this leads to a cyclic argument as we have not obtained the correctness proof of Dijkstra's algorithm yet! Without Dijkstra's algorithm (or other single-source shortest path algorithms) in hands, the only obvious approach to formalize the shortest path condition is to use a universal search over all (exponentially many) paths, which is infeasible.

This should be considered as a warning to feasible mathematicians: Easy/standard proofs are not necessarily feasible!

A feasible proof of shortness. Next, we present a *feasible* proof of the shortness property. Instead of performing induction on a property related to shortest paths, notice that the shortness follows from the *slackness condition*, namely:

- For every $u \in V$ and neighbor v of u, either $d_u = \mathsf{INF}$ or $d_v \le d_u + 1$.

Suppose that this can be proved for Dijkstra's algorithm, given any path p from s to $u \in V$, we can prove by structural induction on p that the length of p is at most d_u. Specifically:

- Let $|p|$ be the length of p and $p_{\le i}$ be the prefix of the path p of length i.
- Let $\mathsf{Endpoint}(p)$ be the endpoint of the path p.
- We prove by induction on $i \le |p|$ that $d_{\mathsf{Endpoint}[p_{\le i}]} \le i$. Clearly this is true for $i = 0$ as $\mathsf{Endpoint}[p_{\le 0}] = s$ and $d_s = 0$. Moreover, suppose that this holds for $i < p$, then
$$p_{\le i+1} = p_{\le i} \mapsto \mathsf{Endpoint}[p_{\le i+1}],$$
and by applying the slackness condition on $u := \mathsf{Endpoint}[p_{\le i}]$ and $v := \mathsf{Endpoint}[p_{\le i+1}]$, we can conclude that $d_v \le d_u + 1 \le i + 1$, where the last inequality follows from the induction hypothesis.

Therefore, it suffices to prove that the distance label d obtained by running Dijkstra's algorithm satisfies the slackness condition.

The key observation is that in contrast to the shortest path condition, the slackness condition can be checked by a straightforward feasible algorithm — we can enumerate the nodes $u \in V$, neighbors v of u, and check whether $d_u = \mathsf{INF}$ or $d_v \le d_u + 1$. The "correctness" of the algorithm does not depend on any unproven results; indeed, the algorithm is exactly how we *formalize* the slackness condition in feasible mathematics. Therefore, by the postulate of structural induction (see Postulate 3), we can prove by induction on this condition.

More formally, let $d^{(i)}$ and $S^{(i)}$ be the values of corresponding variables at the end of the i-th iteration, and $\mathsf{CheckDijkstra}(d, S, V, E)$ be the straightforward algorithm that outputs 1 if

- (*slackness condition*): for every $u \in V \setminus S$ and neighbor v of u, either $d_u = \mathsf{INF}$ or $d_v \le d_u + 1$, and
- (*monotonicity condition*): for every $u \in V \setminus S$ and $v \in S$, $d_u \le d_v$.

hold and outputs 0 otherwise. We prove by induction on $i \le n$ that

$$\mathsf{CheckDijkstra}(d^{(i)}, S^{(i)}, V, E) = 1.$$

Note that the base case is trivial as $S^{(0)} = V$. To prove the induction case, assume that $\mathsf{CheckDijkstra}(d^{(i)}, S^{(i)}, V, E) = 1$. Let $w \in S^{(i)}$ be the node chosen in the $(i+1)$-th iteration. We will prove that $\mathsf{CheckDijkstra}(d^{(i+1)}, S^{(i+1)}, V, E) = 1$, for which we need to prove that the *slackness condition* and *monotonicity condition* hold.

Hint: Recall that u is chosen to be the node in S with the smallest distance label.

The monotonicity condition is relatively simple and is left as an easy exercise. We only consider the slackness condition. Let $u \in S^{(i+1)} = S^{(i)} \cup \{w\}$ and v be its neighbor. We will perform the following case study:

- Suppose that $u \in S^{(i)}$ and v is a neighbor of u, we know that $d_u^{(i+1)} = d_u^{(i)}$ (as the Dijkstra's algorithm will only update the distance label of nodes not in S) and $d_v^{(i+1)} \le d_v^{(i)}$ (as the distance label will not increase). Therefore, the slackness condition holds for (u, v) after the $(i+1)$-th iteration provided that it held after the i-th iteration, which is true by the induction hypothesis.

- Suppose that $u = w$ and v is a neighbor of u that is in $S^{(i)}$, we know by the induction hypothesis that $d_v^{(i)} \leq d_u^{(i)}$. By the description of the algorithm, it is clear that $d_v^{(i+1)} \leq d_v^{(i)}$ and $d_u^{(i+1)} = d_u^{(i)}$, which implies that
$$d_v^{(i+1)} \leq d_v^{(i)} \leq d_u^{(i)} = d_u^{(i+1)}.$$
- Suppose that $u = w$ and v is a neighbor of u that is not in $S^{(i)}$, it follows immediately that the update step of the Dijkstra's algorithm in the $(i+1)$-th iteration ensures that $d_v^{(i+1)} = \min\{d_v^{(i)}, d_u^{(i+1)} + 1\} \leq d_u^{(i+1)} + 1$.

This completes the inductive proof and concludes that the slackness condition holds for the distance label obtained from Dijkstra's algorithm. Subsequently, it leads to a feasible proof of the shortness property.

Remark 1.3.1. The proof explained in this section is informal. To accept that this is a feasible proof, one would need to accept that the proof only utilizes allowed inductions (which we explained informally) and definition axioms of feasibly constructive algorithms (e.g., “the update step of the Dijkstra's algorithm will ensure ...”). will try to justify that the informal claims regarding the feasibility of proofs can indeed be naturally *formalized* in a robust theory that realizes the three informal postulates.

1.4 A Philosophical Perspective

At the end of the chapter, I would like to mention an interpretation of feasible mathematics as postulated that has not been discussed in the literature.

Our definition of feasible mathematics (which is intended to capture the intuition behind Cook's theory PV [Coo75]) is more or less the *minimum* theory to reason about polynomial-time algorithms. It only includes the definition axioms of the algorithms as well as the induction principle on the binary representation of numbers for predicates that can be effectively verified. (Recall that in the postulate of structural induction, the property must be an equation $f(x) = g(x)$ for feasibly constructive functions $f(x)$ and $g(x)$, and thus the equation can be verified efficiently given x.) If a property about the algorithm, e.g., its correctness, can be proved in feasible mathematics, it indicates that such a proof merely requires minimum power of reasoning, and in this sense, it is an “easy” property about the algorithm.

There is another sub-area of computer science — the research on programming languages — that particularly cares about proving the correctness of programs. In developing computer programs, a *specification* (either formally written or informally described) defines what is the intended behavior of the programs, and the programmers aim to *ensure* that the program satisfies the specification, i.e., it is *correct*. One of the main purposes of the design of modern programming languages is to make the programmers' lives easier, and in particular, it would be great if the correctness of the programs follows closely from the construction of the programs. I quote from E. W. Dijkstra's Turing award lecture *The Humble Programmer* [Dij72]:

> “The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not

> first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand."

What Dijkstra did not mention but should arguably be true is that a good programming language should also let *feasibility proof* and program grow hand in hand: Programmers may not see immediately whether the algorithm runs in $O(n^2)$ or $O(n^3)$ time, but it should be easy in most cases to distinguish between feasible and infeasible algorithms.

In particular, Dijkstra's famous "go to statement considered harmful" [Dij68] is supported by the fact that the correctness of a program with go-to statements can not "grow hand in hand"; it does not follow immediately from the structure of the programs, and in fact, programs with go to statements barely have any structure that can be easily understood by humans. A program written in modern programming languages, instead, usually clearly demonstrates their feasibility and correctness.

I would like to imagine feasible mathematics as the logic of the humble programmers as described by Dijkstra: The correctness proof and program can grow hand in hand if and only if the correctness proof is feasible. Putting it more carefully, I make the following *humble programmer conjecture*:

> A statement is feasibly provable, if and only if it is possible to write a feasible (i.e. polynomial-time) program M in a suitable programming language such that
>
> - the statement encodes the correctness of M in a natural sense;
> - the feasibility and correctness proof and M can "grow hand in hand" by a common programmer.

If this conjecture is true, it indicates that the past decades of education practice in programming are implicitly teaching people to be *feasible mathematicians*, although it has never been explicitly mentioned by teachers and students. Certain aspects of feasible mathematics are usually considered a part of "computational thinking" (see, e.g., [Aho11]) in the theory and practice of computer science education. Following this argument, one may further think that the highly successful development of computer science in the past century may eventually be the consequence of the power of feasible mathematics.

There are two clarifications of the conjecture.

First, it is possible that the correctness of some classical algorithms is not feasibly provable. As we mentioned in Section 1.3.2, the standard proof of the correctness of Dijkstra's algorithm on the single-source shortest path is not feasible. Indeed, it is an interesting open problem to systematically investigate the feasibility of the analysis of algorithms.

If your assumption to "common programmers" is stronger than mine, maybe you could reformulate the conjecture with respect to a certain level of Buss's hierarchy, rather than Cook's PV.

Nevertheless, this does not falsify the conjecture unless one thinks that the correctness of those algorithms is something that naturally comes to the mind of a programmer the first time they wrote down the algorithm (otherwise it is not "growing hand in hand"). I would guess that one of the main reasons for people to think that some algorithms are tricky and intellectually satisfying is that the proof is beyond the mind of programmers, i.e., feasible mathematics.

Second, the conjecture does not mean that feasible mathematics is *easy*. For instance, we know that the PCP theorem (in a natural formalization) is provable in Cook's theory PV [Pic15] and thus is feasibly provable.[2] However, it makes little sense to think that the PCP theorem is easy. The only claim I made is that the feasible proof can "grow" while writing a feasible program, while the final program could be extremely hard and thus the proof is extremely hard even if the statement we want to prove (i.e. the specification of the program) is simple. Computer programs (such as modern web browsers and operating systems) can certainly be extremely hard and the programs can be much more complicated compared to their specifications.

Although this conjecture is hard to verify, there are concrete tasks that may help to support it. I believe that the correctness of most programs, or at least most parts of most programs, can be formalized in a natural sense and proved in feasible theories such as PV [Coo75]. In particular, the correctness of compilers of structured programming languages that allow "correctness proof and program grow hand in hand", if formalized properly, should be provable in PV. In subsequent , we will partially demonstrate why this could be true: Starting from Cook's definition of PV that supports few programming functionalities, we will show how to construct a much stronger "programming language" that supports (for instance) data structures much as *lists* and *maps*, and how the correctness proof in the strong programming language translates back to a correctness proof in PV.

1.5 Bibliographical and Other Remarks

An early result on the "complexity" of mathematical proofs is Gödel's speedup theorem [Göd36] (see also [Bus94, Bus95a]) that separates the k-th and $(k+1)$-th order arithmetic in terms of the length of proofs. As a complexity measure, the length of proofs is inherently different from the feasibility: a proof of length 10^{100} could still be feasible if it satisfies the three postulates, while a single-line proof could be infeasible if it violates the postulates, e.g., it defines an infeasible function. A more recent result on the length of arithmetic proofs is due to Friedman (unpublished) and later improved by Pudlák [Pud87].

As Ján Pich pointed out (in private communication), ultrafinitists do not think PV proofs are feasible for this reason. I am not an ultrafinitist :)

The first bounded theory and the concept of feasible mathematics is introduced by Parikh [Par71], with the example that the exponentiation function is infeasible (see Example 1.1.1). Parikh's theory $I\Delta_0$ also follows variants of the three postulates, while the interpretation of "feasibility" is the complexity class called *linear-time hierarchy* rather than polynomial-time functions (see [Bus99]). Another notable example is Buss's theories S_2^i, T_2^i corresponding to the polynomial-time hierarchy [Bus86].

Instead of interpreting "effective procedures" as functions in certain complexity classes, another standard approach is to consider the *computability* of functions. This is highlighted in the program of reverse mathematics (see, e.g., [Sim09, Sti20, DM22]).

[2] In more details, Pich [Pic15] formalizes Dinur's combinatorial proof of the PCP theorem [Din07] in PV.

Example 1.1.2 is folklore (see, e.g., [Kra01, CLO24]). Unprovability results in bounded arithmetic based on cryptographic assumptions can be found in [KP98, Bus08, ILW23].

The provability of the induction principle in Section 1.3.1 in bounded arithmetic is well-known and is implicit in, e.g., Buss's proof of the witnessing theorem for S_2^1 (see [Bus86, Section 5.2]) and [KPT91]. To our knowledge, the example of Dijkstra's algorithm in Section 1.3.2 was not in the literature.

Almost all formalizations of feasible mathematics are finitistic, and in fact, the infeasible first-order Peano Arithmetic is generally considered to be finitistic. Nevertheless, there has been attempt to generalize the idea of feasible mathematics to non-finitistic mathematics, see [BBF$^+$19] and the references therein.

Chapter 2

Formal Definition and Basic Programming

In this chapter, we will formally define the theory for feasible mathematics characterized by the three informal postulates in the previous chapter. The theory, which is called PV [Coo75], provides a way to formally define feasibly constructible functions and reason about them.

2.1 Cook's Theory PV

Cook's theory PV [Coo75] is defined as an equational theory, i.e., it only deals with equations "$u = v$". Intuitively, PV is defined using a recursive-theoretic characterization of polynomial-time functions due to Cobham [Cob65]. The main reason to choose this particular construction is that the definition axioms of functions are easy to define; indeed, Cook's theory PV can be viewed as a time-bounded counterpart of Skolem's primitive recursive arithmetic [Sko23]. In addition, Cobham's characterization is *machine-independent*, i.e., does not depend on any concrete machine model, which makes it easier to work on and (probably) more general.

Concretely, we will define:

- *Feasibly constructible functions* are constructed according to Cobham's rules. Moreover, Cobham's characterization allows introducing a function in a recursion manner using the rule of *limited recursion on notation.* Explained below :)
- *Definition axioms* are Cobham's rules to define the functions.

Moreover, PV allows a *restricted version* of structural induction rule. Recall in the standard structural induction proofs, we need to prove that if an equation $f(x) = g(x)$, then $f(2x) = g(2x)$ and $f(2x+1) = g(2x+1)$. However, this requires a proper notion of "conditional proofs" that, in Cook's definition, is not natively supported. Instead, Cook [Coo75] introduced a weaker version of induction that is closely related to Cobham's characterization, and showed that the standard induction proofs can be translated back to the weaker version of the induction rule.

2.1.1 Cobham's Characterization of FP

Before proceeding with the formal definition, we fix the notation: ε denotes the empty string, $|a|$ denotes the bit-length of $a \in \{0,1\}^*$, FP is the set of functions $f(x_1, \dots, x_k) : (\{0,1\}^*)^k \to \{0,1\}^*$ (for a constant $k \in \mathbb{N}$) that are computable by polynomial-time algorithms (i.e., in $\mathsf{poly}(|x_1|, \dots, |x_n|)$ time). We use $\vec{x} = (x_1, \dots, x_k)$ to denote a vector of variables for some $k \in \mathbb{N}$.

We first define some simple functions:

- $c(x) := \varepsilon$;
- $\circ(x, y)$ means concatenation of (the binary representation of) x and y;
- $s_i(x) := x \circ i$, $i \in \{0,1\}$;
- $\#(x, y)$ means concatenation of the binary representation of x for $|y|$ times;
- $\mathsf{TR}(x)$ removes the rightmost bit of x;
- $\pi_i(x_1, \dots, x_k) := x_i$ for $i \in [k]$.

We may use $\circ$ and $\#$ as infix operators $x \circ y$ and $x \# y$, where $\#$ has higher precedence, and denote $\mathsf{TR}(x)$ by $\lfloor x/2 \rfloor$.

In particular, the identity function is $\pi_1(x) := x$.

We introduce two Cobham rules for defining new functions from existing functions: the composition rule and the rule of limited recursion on notation.

- (*Composition*): From $h(x_1, \dots, x_k)$ and $g_1(\vec{y}), g_2(\vec{y}), \dots, g_k(\vec{y})$, we can introduce $f(\vec{y}) := h(g_1(\vec{y}), g_2(\vec{y}), \dots, g_k(\vec{y}))$, where $k \in \mathbb{N}$.
- (*Limited Recursion on Notation*): From $g(\vec{x})$, $h_i(\vec{x}, y, z)$ ($i \in \{0,1\}$), and $k(\vec{x}, y)$, we can introduce $f(\vec{x}, y)$ as:

$$f(\vec{x}, 0) := g(\vec{x}) \qquad f(\vec{x}, s_i(y)) := h_i(\vec{x}, y, f(\vec{x}, y)),\ i \in \{0,1\}; \tag{2.1}$$

provided that $|f(\vec{x}, y)| \le |k(\vec{x}, y)|$.[1]

Let $\mathcal{F}$ be the smallest class of functions that contains the base functions and is closed under these two rules. It is clear that $\mathcal{F} \subseteq \mathsf{FP}$, and indeed, Cobham [Cob65] proved:

Theorem 2.1.1 (Cobham [Cob65]). *$\mathcal{F} = \mathsf{FP}$.*

We provide an illustrative example:

Example 2.1.1. The *if-then-else* function $\mathsf{ITE}(y, u, v)$ is defined as

$$\mathsf{ITE}(y, u, v) := \begin{cases} u & y = 1; \\ v & y = 0; \\ * & \text{otherwise;} \end{cases}$$

where $*$ is a placeholder meaning that we don't care about the value. To define this function, we use the rule of limited recursion on notation. Let $g(u, v) := \varepsilon$, $h_0(u, v, y, z) := v$, $h_1(u, v, y, z) := u$, and $k(u, v, y) := u \circ v$. Then let $f(y, u, v)$ as:

$$f(u, v, y) := g(u, v); \tag{2.2}$$

$$f(u, v, s_i(y)) := h_i(u, v, y, f(u, v, y)) \ \ i \in \{0,1\}. \tag{2.3}$$

[1] The inequality $|f(\vec{x}, y)| \le |k(\vec{x}, y)|$ ensures the recursion will not generate strings of super-polynomial bit-length.

It can be verified that $|f(u,v,y)| \leq |k(u,v,y)|$, as $|u| \leq |u \circ v|$ and $|v| \leq |u \circ v|$. Let

$$\mathsf{ITE}(y,u,v) := f(\pi_2(y,u,v), \pi_3(y,u,v), \pi_1(y,u,v)) = f(u,v,y)$$

is defined as the composition of f and π_2, π_3, π_1.

We can always permute variables wlog using π and composition :)

The intuition behind Theorem 2.1.1 is to show that the (clocked) universal Turing machine, i.e., the function $\mathcal{U}(M,x,t)$ that simulates the Turing machine M on the input x for $|t|$ steps, is definable in $\mathcal{F}$. To see that this is sufficient, we can check that if $M(x)$ runs in time $O(|x|^c)$ for a constant c, then

$$\mathcal{U}(M, x, \overbrace{x \# x \# \ldots \# x}^{c+1 \text{ times}}) = M(x)$$

when the input length is sufficiently large.

Note that one can always hard-code the output when the input length is small using the ITE function.

Remark 2.1.1. We note that our definition of Cobham's family $\mathcal{F}$ is different from the original definition [Cob65] and the standard expositions in literature, where $\mathcal{F}$ is defined as a class of functions over $\mathbb{N}$ rather than $\{0,1\}^*$. There is essentially no difference between these two formulations; we choose the formalization using Boolean strings rather than $\mathbb{N}$ because it is more intuitive from a computational perspective.

2.1.2 Definition of PV

The Cobham rules are simple and clean except for the presence of the "upper bound function" $k(\vec{x}, y)$ in the rule of limited recursion on notation.

This causes a technical issue in defining feasible mathematics Cobham rules. The rule of limited recursion on notation requires the inequality $|f(\vec{x},y)| \leq |k(\vec{x},y)|$ to be "true". To qualify as a feasible construction, however, it is not enough to have the inequality be true. According to the postulate of feasible functions (see Postulate 1), we require that the construction of functions needs to "clearly demonstrate" the feasibility of the function, and the feasibility of f obtained from limited recursion on notation using (g, h_0, h_1, k) is clearly demonstrated only if the inequality $|f(\vec{x},y)| \leq |k(\vec{x},y)|$ is *clearly demonstrated.*

Intuition of the definition. Cook [Coo75] resolves the issue with a natural idea — inductively define PV *functions* and PV *proofs* simultaneous. Concretely:

- Base functions in Cobham's class $\mathcal{F}$ are order-0 PV functions and their definition axioms are order-0 PV proofs.
- One can introduce an order-$(i+1)$ function f by limited recursion on notation if the inequality $|f(\vec{x},y)| \leq |k(\vec{x},y)|$ admits an order-i PV proof.
- An order-i PV proof can use the definition axioms of order-i PV functions as well as the structural induction over the definition of order-i PV functions.

and thus is clearly demonstrated

Finally, *feasible functions* are formally defined as PV functions of finite order, and *feasible proofs* are formally defined as PV proofs of finite order.

Before giving the formal definition, we make a slight technical change to the rule of limited recursion on notation. Instead of proving the inequality $|f(\vec{x},y)| \le |k(\vec{x},y)|$, we require to prove two inequalities $|h_i(\vec{x},y,z)| \le |z \circ k_i(\vec{x},y)|$ for $i \in \{0,1\}$. It is not hard to show that the updated rule is at most as strong as the original one:

Lemma 2.1.2. *Let $\mathcal{F}'$ be the class obtained by replacing the inequalities $h_i(\vec{x},y) \le k(\vec{x},y)$ in the rule of limited induction on notation in $\mathcal{F}$ to $|h_i(\vec{x},y,z)| \le |z \circ k_i(\vec{x},y)|$ for $i \in \{0,1\}$. Then $\mathcal{F}' \subseteq \mathcal{F}$.*

Proof Sketch. We first show that $\mathcal{F}' \subseteq \mathcal{F}$ by structural induction over the Cobham rules (with the updated limited recursion on notation rule). We need to prove that if f is introduced from $g, h_1, h_2 \in \mathcal{F}$ by Equation (2.1) such that $|h_i(\vec{x},y,z)| \le |z \circ k_i(\vec{x},y,z)|$ for some $k_i \in \mathcal{F}$, $i \in \{0,1\}$, then $f \in \mathcal{F}$. Indeed, we will prove $f \in \mathsf{FP}$ and apply Cobham's theorem (see Theorem 2.1.1). This follows from the fact that

We use $\mathsf{BIT}(x,j)$ to denote the j-th bit of x, and $y_{>j}$ be the suffix of y of length $|y|-j$.

$$\begin{aligned}
|f(\vec{x},s_i(y))| &\le |h_i(\vec{x},y,f(\vec{x},y))| \\
&\le |f(\vec{x},y)| + |k_i(\vec{x},y)| && \text{(unfolding } f \text{ recursively)} \\
&\le |g(\vec{x})| + \sum_{0 \le j < |s_i(y)|} |k_{\mathsf{BIT}(s_i(y),j+1)}(\vec{x},s_i(y)_{>j}))| \le \mathsf{poly}(|\vec{x}|,|y|)
\end{aligned}$$

and that $g, h_0, h_1 \in \mathcal{F} = \mathsf{FP}$. (Notice that the the computation of f only requires calling g, h_0, h_1 on inputs of $\mathsf{poly}(|\vec{x}|,|y|)$ length.) □

Indeed, a white-box inspection of the proof of Cobham's theorem shows that $\mathcal{F}' = \mathcal{F}$. We postpone this to Theorem 2.3.2, which will be proved in Chapter 4.

Moreover, since we will use the comparison of the bit-length of numbers, we will need to introduce the *iterative trimming function* $\mathsf{ITR}(x,y)$ that removes the leftmost $|y|$ bits of x as the base function, and formalize $|x| \le |y|$ by the equation $\mathsf{ITR}(x,y) = \varepsilon$.

Definition of PV. Now we present the formal definition of equational theory PV. We first define the base case:

- ε be a constant symbol.
- $s_0(x), s_1(x), \circ(x,y), \#(x,y), \mathsf{TR}(x), \mathsf{ITR}(x,y)$ are function symbols of order 0.
- A *term* of order i is defined by compositions of order-i functions, the constant symbol, and variables, e.g., $\mathsf{ITR}(s_0(s_1(x)), s_1(y))$ is a term of order 0.
- An *equation* of order i is of form $s = t$, where s, t are terms of order i.

where x, y are variables

The definition axioms of the function symbols are proofs of order 0:

$$x \circ \varepsilon = x, \quad x \circ s_i(y) = s_i(x \circ y) \qquad i \in \{0,1\} \tag{2.4}$$

$$x \# \varepsilon = \varepsilon, \quad x \# s_i(y) = x \circ (x \# y) \qquad i \in \{0,1\} \tag{2.5}$$

$$\mathsf{TR}(\varepsilon) = \varepsilon, \quad \mathsf{TR}(s_i(x)) = x \qquad i \in \{0,1\} \tag{2.6}$$

$$\mathsf{ITR}(x,\varepsilon) = x, \quad \mathsf{ITR}(x,s_i(y)) = \mathsf{TR}(\mathsf{ITR}(x,y)) \qquad i \in \{0,1\} \tag{2.7}$$

The intended meaning of the constant symbol ε is the *empty string.* The meaning of a variable is an arbitrary Boolean string. The intended meaning of a provable equation $s = t$ is that $s(\vec{x}) = t(\vec{x})$, where $\vec{x} = (x_1, \ldots, x_n)$ are variables in the terms s and t, s and t are interpreted as functions over $(\{0,1\}^*)^n$. We also note that s_0 and s_1 do not have definition axioms; the intended meaning of $s_i(x)$ is to append a bit $i \in \{0,1\}$ to the right of a string x. They are used as the encoding of Boolean strings, e.g., 0011 is encoded as $s_1(s_1(s_0(s_0(\varepsilon))))$.

We assume that one can choose variables from a countably infinite set of alphabets to form terms and introduce new functions (see the function introduction rules below).

Remark 2.1.2. We note that similar to our formalization of Cobham's class (see Remark 2.1.1), our treatment of the intended meaning of PV is slightly different from that of Cook's original definition [Coo75], in which he interpreted functions in PV as functions in $\mathbb{N}$. To deal with the encoding of natural numbers, Cook's definition uses the *dyadic notation* of natural numbers, and the functions s_0, s_1 are replaced by $s_1(x) = 2x + 1$ and $s_2(x) = 2x + 2$.

I feel slightly more comfortable working with $\{0,1\}$ instead of $\{1,2\}$. What about you?

Function introduction rules. For every $i \geq 1$, a function of order i can be introduced according to one of the following two rules.

The feasibility of $f_t^{(0)}$ is clearly demonstrated by Cobham's rule of composition.

- (*Composition*). If t is an order-$(i-1)$ term with variables $\vec{x}$, $f_t^{(0)}$ can be introduced as an order-i function with the definition axiom $f_t^{(0)}(\vec{x}) = t$.
- (*Recursion*). If $g(\vec{x})$, $h_0(\vec{x}, y, z)$, $h_1(\vec{x}, y, z)$, $k_0(\vec{x}, y)$, $k_1(\vec{x}, y)$ are order-$(i-1)$ functions, and there are order-$(i-1)$ proofs π_i of the equation:

$$\mathsf{ITR}(h_i(\vec{x}, y, z), z \circ k_i(\vec{x}, y)) = \varepsilon \tag{2.8}$$

Informally, eq. (2.8) means that: $|h_i(\vec{x}, y, z)| \leq |z \circ k_i(\vec{x}, y)|$

for $i \in \{0,1\}$, then $f_\Pi^{(1)}$ (where $\Pi := (g, h_0, h_1, k_0, k_1, \pi_0, \pi_1)$) may be introduced as an order-i function with the definition axioms: [2]

$$f_\Pi^{(1)}(\vec{x}, 0) = g(\vec{x}) \tag{2.9}$$

$$f_\Pi^{(1)}(\vec{x}, s_i(y)) = h_i(\vec{x}, y, f_\Pi^{(1)}(\vec{x}, y)) \quad i \in \{0,1\} \tag{2.10}$$

Moreover, any function of order $i - 1$ is also an order-i function.

Deduction rules. For every $i \geq 1$, an order-i proof of an order-i equation $s = t$ is a sequence of order-i equations $(e_1, e_2, \ldots, e_\ell)$ for some $\ell \in \mathbb{N}$, where $e_\ell =$ "$s = t$" and the equations are introduced one by one following the rules below:

- (*Logical Rules*). The logical rules for equations follow:
  - (L0): One may introduce $s = s$ for any term s.
  - (L1): If $s = t$ has been introduced, one may introduce $t = s$.

Exercise: (L0) follows from other rules.

[2] The feasibility of $f_\Pi^{(1)}$ is clearly demonstrated by Cobham's rule of limited recursion on notation (as modified in Lemma 2.1.2).

  - (L2): If $s = t$, $t = u$ have been introduced, one may introduce $s = u$.
  - (L3): If $s_1 = t_1, \ldots, s_n = t_n$ has been introduced and $f(x_1, \ldots, x_n)$ is an order-i function symbol with n variables, one may introduce the equation $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$.
  - (L4): If $s = t$ has been introduced, v is an order-i term, and x is an variable, one may introduce $s[x/v] = t[x/v]$, where $s[x/v]$ denote the term obtained from s by substituting all occurrences of x by v.

- (*Definition Axioms*). A definition axiom of an order-i function may be introduced without premise.

- (*Structural Induction*). If $g(\vec{x})$, $h_0(\vec{x}, y, z)$, $h_1(\vec{x}, y, z)$ are order i functions, and $f_1(\vec{x}, y)$, $f_2(\vec{x}, y)$ are two functions satisfying that equations

$$f_1(\vec{x}, \varepsilon) = g(\vec{x}), \quad f_1(\vec{x}, s_i(y)) = h(\vec{x}, y, f_1(\vec{x}, y)),\ i \in \{0, 1\} \tag{2.11}$$

$$f_2(\vec{x}, \varepsilon) = g(\vec{x}), \quad f_2(\vec{x}, s_i(y)) = h(\vec{x}, y, f_2(\vec{x}, y)),\ i \in \{0, 1\} \tag{2.12}$$

  have all been introduced, then one may introduce $f_1(\vec{x}, y) = f_2(\vec{x}, y)$.

It is tedious to the extent that authors usually omit it in research papers.

This completes the formal definition of PV.

We will use the standard notation $\mathsf{PV} \vdash s = t$ to denote that there is a (finite-order) PV proof of the equation $s = t$. The equational theory PV is defined as the set of equations with (finite-order) PV proofs.

Of course, you can force variables in different equations to have different names to avoid any confusion...

Remark 2.1.3. Here we explain the intended meaning of variables in PV equations. Variables in an equation $s = t$ should be considered to be *universally quantified.* If $s(\vec{x}) = t(\vec{x})$ is a PV-provable equation, the intended meaning is that for every $\vec{x}$, $s(\vec{x}) = t(\vec{x})$; or equivalently, the *functions* $\vec{x} \mapsto s(\vec{x})$ and $\vec{x} \mapsto t(\vec{x})$ are functionally equivalent. Variables with the same name in an equation are considered to be identical, while variables in different equations should be considered independent even if they share the same name.

2.1.3 Warm-up: Basic Properties of Concatenation

As a warm-up, we first prove a couple of basic properties about the concatenation function $\circ$. We will prove that ε is the left identity of concatenation (i.e. $\varepsilon \circ x = x$) and that concatenation is associative (i.e. $(x \circ y) \circ z = x \circ (y \circ z)$).[3]

Proposition 2.1.3. $\mathsf{PV} \vdash \varepsilon \circ x = x$.

Proof. We define two PV functions $f_1(x) := \varepsilon \circ x$ and $f_2(x) := x$ by the composition rule using terms $\varepsilon \circ x$ and x, respectively. We will prove in PV that $f_1(x) = f_2(x)$, which immediately implies $\varepsilon \circ x = x$ by (L2) transitivity and the definition axioms of f_1 and f_2.

[3] Note that ε is the right identity of concatenation by the definition axiom.

Intuitively, the proof follows from a structural induction on x. We first verify the base case, i.e., $x = \varepsilon$:

$$\begin{aligned}
&\mathsf{PV} \vdash f_1(\varepsilon) = \varepsilon \circ \varepsilon && \text{((L4) substitution } x/\varepsilon \text{ to the definition axiom)}\\
&\mathsf{PV} \vdash x \circ \varepsilon = x && \text{(Axiom eq. (2.4))}\\
&\mathsf{PV} \vdash \varepsilon \circ \varepsilon = \varepsilon && \text{((L4) substitution } x/\varepsilon \text{ to the equation above)}\\
&\mathsf{PV} \vdash f_1(\varepsilon) = \varepsilon && \text{((L2) transitivity)}
\end{aligned}$$

Similarly, $f_2(\varepsilon) = \varepsilon$. Therefore, let $g = \varepsilon$ be a function with no variable, then $f_j(\varepsilon) = g$ ($j \in \{1, 2\}$) is provable in PV.

Then, we consider the induction step, i.e., x is of the form $s_i(x)$ for $i \in \{0, 1\}$. Notice that for every $i \in \{0, 1\}$:

$$\begin{aligned}
&\mathsf{PV} \vdash f_1(s_i(x)) = \varepsilon \circ s_i(x) && \text{((L4) substitution } x/s_i(x) \text{ to the definition axiom)}\\
&\mathsf{PV} \vdash \varepsilon \circ s_i(x) = s_i(\varepsilon \circ x) && \text{((L4) substitution } x/\varepsilon, y/x \text{ to Axiom eq. (2.4))}\\
(*)\ &\mathsf{PV} \vdash \varepsilon \circ x = f_1(x) && \text{((L1) symmetricity to the definition axiom)}\\
(\star)\ &\mathsf{PV} \vdash s_i(x) = s_i(x) && \text{((L0) reflexivity)}\\
&\mathsf{PV} \vdash s_i(\varepsilon \circ x) = s_i(f_1(x)) && \text{((L3) substitution with } (\star) \text{ and } (*))\\
&\mathsf{PV} \vdash f_1(s_i(x)) = s_i(f_1(x)) && \text{((L2) transitivity)}
\end{aligned}$$

Similarly, one can prove that $f_2(s_i(x)) = s_i(f_2(x))$. Therefore, let $h_i(x, z) = s_i(z)$, we can prove that for $j \in \{1, 2\}$ and $i \in \{0, 1\}$:

$$\mathsf{PV} \vdash f_j(s_i(x)) = h_i(x, f_j(x)).$$

Then, by the rule of structural induction, we can prove that $f_1(x) = f_2(x)$ as they are both inductively defined from (g, h_0, h_1). □

Proposition 2.1.4. $\mathsf{PV} \vdash (x \circ y) \circ z = x \circ (y \circ z)$.

Proof. We define two functions $f_1(x, y, z) := (x \circ y) \circ z$ and $f_2(x, y, z) := x \circ (y \circ z)$, and it suffices to prove that $f_1(x, y, z) = f_2(x, y, z)$. The strategy is to use the induction rule. Notice that

$$\begin{aligned}
&\mathsf{PV} \vdash f_1(x, y, \varepsilon) = (x \circ y) \circ \varepsilon && \text{((L4) substitution } z/\varepsilon \text{ to the definition axiom)}\\
&\mathsf{PV} \vdash x \circ \varepsilon = x && \text{(Definition axiom of } \circ)\\
&\mathsf{PV} \vdash (x \circ y) \circ \varepsilon = x \circ y && \text{((L4) substitution } x/(x \circ y) \text{ to equation above)}\\
&\mathsf{PV} \vdash f_1(x, y, \varepsilon) = x \circ y && \text{((L2) transitivity)}
\end{aligned}$$

Similarly, we can (with slightly more steps) prove that $\mathsf{PV} \vdash f_2(x, y, \varepsilon) = x \circ y$.

Also, notice that for every $i \in \{0, 1\}$,

$$\begin{aligned}
&\mathsf{PV} \vdash f_1(x, y, s_i(z)) = (x \circ y) \circ s_i(z) && \text{((L4) substitution to the definition axiom)}\\
&\mathsf{PV} \vdash x \circ s_i(y') = s_i(x \circ y') && \text{(Definition axiom of } \circ)\\
&\mathsf{PV} \vdash (x \circ y) \circ s_i(z) = s_i((x \circ y) \circ z) && \text{((L4) substitution } x/(x \circ y) \text{ and } y'/z)\\
&\mathsf{PV} \vdash (x \circ y) \circ z = f_1(x, y, z) && \text{((L1) symmetricity to the definition axiom)}\\
&\mathsf{PV} \vdash s_i((x \circ y) \circ z) = s_i(f_1(x, y, z)) && \text{((L3) using above and function } s_i)\\
&\mathsf{PV} \vdash f_1(x, y, s_i(z)) = s_i(f_1(x, y, z)) && \text{((L2) transitivity, twice)}
\end{aligned}$$

Similarly, we can (again with slightly more steps) prove that $\mathsf{PV} \vdash f_2(x, y, s_i(z)) = s_i(f_2(x, y, z))$. Therefore, by the induction rule using $g(x, y) := x \circ y$ and $h_i(x, y, z) = s_i(z)$, we can prove that $f_1(x, y, z) = f_2(x, y, z)$. □

The proof strategies of these two propositions are similar. Take the latter one as an example. To prove the equation $(x \circ y) \circ z = x \circ (y \circ z)$, we indeed prove that both functions (on the LHS and RHS of the equation) are identical to the following recursively defined function

$$f(x, y, \varepsilon) = x \circ y; \quad f(x, y, s_i(z)) = s_i(f(x, y, z)).$$

Note that this particular recursive construction of f is feasible, since it is constructed inductively on the variable z that takes at most $|z|$ rounds, and $|f(x, y, z)| \leq |f_1(x, y, z)|$ for a function $f_1(x, y, z) := (x \circ y) \circ z$ that is known to be feasible.

The function f is not formally introduced, but it is instructive to think of the proof as induction over f.

f_1 is feasible as $\circ$ is feasible, and the composition of feasible functions is feasible.

Moreover, the proofs that both sides of the original equation are identical to $f(x, y, z)$ are feasible, as they only require the definition axioms of $\circ$. Therefore, we can prove by structural induction over z that both sides of the equation are identical to $f(x, y, z)$, concluding that the equation is true. Throughout the proof, we are only utilizing definition axioms of feasibly constructible functions and the structural induction rule in PV.

2.2 Thesis of Feasible Mathematics

Cook [Coo75] proposed a *Verifiability Thesis*: the theory PV characterizes the informal concept of *polynomially verifiability*, which we have not explored carefully. Here, we propose an alternate *Thesis of Feasible Mathematics*: PV *is* the theory that characterizes feasible mathematics explained by the three postulates, or more concretely:

> A mathematical statement is provable in feasible mathematics (informally defined by the three postulates) if and only if there is a straightforward formalization of the statement in PV that is provable in PV.

Subsequently, the "proof" below is not a mathematical proof but rather an illustration.

Similar to the Church-Turing Thesis and Cook's Verifiability Thesis, the Thesis of Feasible Mathematics is not a precise mathematical statement but a philosophical claim, and thus cannot be proved. To some extent the Thesis is falsifiable — it will be broken if one can construct an equation in the language of PV that is clearly feasibly provable but does not have a straightforward proof in PV. However, people may also disagree with whether a statement is provable in informal feasible mathematics and the whether a formalization is "straightforward".

Nevertheless, we will try to explain why it is reasonable to believe that PV suffices for feasible mathematicians.

PV proofs $\Rightarrow$ feasible proofs. We will need to show that the function symbols in PV are feasibly constructible functions and PV proofs are proofs in the informal notion of feasible mathematics. Since PV functions and proofs are inductively defined simultaneously, we will prove by induction on k that PV functions of order k are feasibly constructible and PV proofs of order k are feasible proofs.[4]

We first consider the base case $k = 0$. Order-0 functions of PV are base functions, which are simple function that clearly demonstrates their feasibility, and thus should be considered as feasibly constructible functions by Postulate 1. Similarly, order-0 PV proofs are simply the definition axioms of order-0 functions that support their feasibility and thus are considered feasible by Postulate 2.

We then consider the induction case. Suppose that order-k PV functions are feasibly constructible and order-k PV proofs are feasible. Recall that order-$(k+1)$ PV functions are constructed either by composition or (limited) recursion from order-k PV functions. The composition of feasible functions is certainly feasible and clearly demonstrates their feasibility through the construction. The mathematical facts supporting the clear demonstration, or its definition axiom, are accepted as axioms in feasible mathematics.

Now we explain why functions introduced by the limited recursion rule are feasibly constructible. Recall that from g, h_0, h_1, k_0, k_1 we can introduce a function through limited recursion if there are order-k PV proofs of

$$\mathsf{ITR}(h_i(\vec{x}, y, z), z \circ k_i(\vec{x}, y)) = \varepsilon \quad (i \in \{0, 1\}).$$

By the induction hypothesis, we know that this proof is feasible, given which by the reasoning in Lemma 2.1.2 it is quite clear that the function f defined by limited recursion from g, h_0, h_1 is feasible. Moreover, the facts that enable the clear demonstration are the equations:

$$\begin{aligned}
&f(\vec{x}, 0) = g(\vec{x}) \\
&f(\vec{x}, s_i(y)) = h_i(\vec{x}, y, f(\vec{x}, y)) \quad (i \in \{0, 1\}) \\
&\mathsf{ITR}(h_i(\vec{x}, y, z), z \circ k_i(\vec{x}, y)) = \varepsilon \quad (i \in \{0, 1\})
\end{aligned}$$

could be included as axioms. Indeed, the first two equations are included as axioms in order-$(k + 1)$ PV proofs, and the last equation admits an order-k PV proof (and thus there is no need to include it as an axiom).

Finally, we show that order-$(k+1)$ PV proofs are also provable in feasible mathematics. The definition axioms are valid by Postulate 2. We now show that the structural induction rule is feasibly provable.

Suppose that f_1, f_2, g, h_0, h_1 are order-$(k + 1)$ PV functions and thus are feasibly constructible, and the equations

$$\begin{aligned}
f_1(\vec{x}, y) = g(\vec{x}), &\quad f_1(\vec{x}, s_i(y)) = h(\vec{x}, y, f_1(\vec{x}, y)),\ i \in \{0, 1\} \\
f_2(\vec{x}, y) = g(\vec{x}), &\quad f_2(\vec{x}, s_i(y)) = h(\vec{x}, y, f_2(\vec{x}, y)),\ i \in \{0, 1\}
\end{aligned}$$

are already feasibly provable, where free variables are considered to be universally quantified. We will need to prove feasibly that $\forall \vec{x}\ \forall y\ f_1(\vec{x}, y) = f_2(\vec{x}, y)$. Fix any $\vec{x}$. Note that given any y, the property $f_1(\vec{x}, y) = f_2(\vec{x}, y)$ is feasibly provable, and we are allowed to prove the equation by structural induction on y:

[4] In some sense, we are arguing that Lemma 2.1.2 is feasibly provable.

- (*Base Case*). $f_1(\vec{x}, \varepsilon) = f_2(\vec{x}, \varepsilon)$ is feasibly provable since both of them are feasibly provably equal to $g(\vec{x})$;
- (*Induction Case*). Recall that by the assumption, it is feasibly provable that $f_j(\vec{x}, s_i(x)) = h(\vec{x}, y, f_j(\vec{x}, y))$ for $j \in \{1, 2\}$ and $i \in \{0, 1\}$. By the induction hypothesis, we know that $f_1(\vec{x}, y) = f_2(\vec{x}, y)$, and thus

$$f_1(\vec{x}, s_i(y)) = h(\vec{x}, y, f_1(\vec{x}, y)) = h(\vec{x}, y, f_2(\vec{x}, y)) = f_2(\vec{x}, s_i(y))$$

  can be feasibly proved.

This induction proof is feasible by Postulate 3. Therefore, $f_1(\vec{x}, y) = f_2(\vec{x}, y)$ is also feasibly provable.

Feasible proofs ⇒ PV proofs. The other direction is a philosophical claim as feasible proofs are not formally defined.

This is similar to the hard direction of the Extended Church-Turing Thesis, where "efficient computation" does not have a precise mathematical definition.

For the Church-Turing Thesis, a justification that Turing machines suffice to capture the informal notion of computation is that it simulates seemingly stronger models, including multi-tape Turing machines, Turing machines with two-dimension tapes, Church's λ-calculus, and recursive functions. Similarly, ZFC set theory is generally accepted as a suitable foundation of mathematics because people demonstrate natural formalizations of many branches of mathematics in ZFC set theory.

To justify the belief that feasibly provable statements can be formalized and proved in PV, we will need to show that:

1. A wide range of feasibly constructible functions are definable in PV in a natural sense. That is, there is a "compiler" from functions that demonstrates their feasibility (possibly written in a reasonable "high-level" programming language) to functions in PV.
2. A wide range of feasibly provable statements are provable in PV in a natural sense. That is, there is a "compiler" from feasible proofs (possibly written in a reasonable "high-level" formal system) to functions in PV.

Note that both properties above are highly non-trivial. For instance, the function introduction rules in PV only allow a restricted version of recursion, and the functions in PV must deal with "raw data" in the form of binary strings rather than in a structured form. Similarly, the deduction rules in PV only allow a special version of induction rather than the general induction rule as discussed in Postulate 3. Nevertheless, just as elaborate modern computer systems can be built from a rather small number of CPU instructions, we will show that the function introduction rules and deduction rules suffice to build a rich system for feasibly constructible functions and feasible mathematics.

2.3 Standard Model of PV

As we demonstrated in the last section, Cobham's class $\mathcal{F}, \mathcal{F}'$ and the fact that $\mathcal{F}' \subseteq \mathcal{F} = \mathsf{FP}$ provide an informal explanation that the definition of PV "satisfies" the three postulates of feasible mathematics. Of course, we will not be able to formally "prove" this, as the three postulates themselves are intended to describe an informal concept of feasibly constructive proofs.

Nevertheless, we can try to formally prove a couple of facts showing that PV does not contradict the postulates in an obvious sense. We will check that all PV functions are indeed feasible (so that it is not too strong), and all feasible functions are definable in PV (so that it is not too weak).

Standard model of PV. We first formally define the *standard model* (or the *standard interpretation*) of PV, i.e., the "ground-truth" that PV tries to depict. We will first define what does it mean by the model of an equational theory.

Skip this part if you are a *real* feasible mathematician as you may refuse to believe the existence of the ground-truth.

Formally, an (equational) theory $\mathcal{T}$ is a set of equations that are considered true. For instance, the theory PV is defined as the set of equations $s = t$ that admits an order-k proof for some $k \in \mathbb{N}$.

Let $\mathcal{L} = (\Sigma, \mathcal{F})$ be the language of an (equational) theory $\mathcal{T}$, where Σ is the set of constant symbols and $\mathcal{F}$ is the set of function symbols. For PV, we have $\Sigma = \{\varepsilon\}$ and $\mathcal{F}$ be the set of all (finite order) PV functions.

A *model* or *structure* is defined as a tuple $\mathcal{M} = (\mathcal{U}, \mathcal{I}^{\mathsf{c}}, \mathcal{I}^{\mathsf{f}})$ where:

- $\mathcal{U}$ is the *universe*, i.e., the set of possible values.
- $\mathcal{I}^{\mathsf{c}} : c \mapsto u$ maps every $c \in \Sigma$ to an element $u \in \mathcal{U}$.
- $\mathcal{I}^{\mathsf{f}} : f \mapsto F$ maps every function symbol $f \in \mathcal{F}$ with k variables to a function $F : \mathcal{U}^k \to \mathcal{U}$ over the universe, for every $k \in \mathbb{N}$.

There is only one predicate "=" that is more or less embedded in the logic, so we don't include it in the language.

We call $u \in \mathcal{U}$ and $F : \mathcal{U}^k \to \mathcal{U}$ above the *interpretation* of $c \in \Sigma$ and $f \in \mathcal{F}$, respectively. We may denote the interpretation of $c \in \Sigma$ (resp. $f \in \mathcal{F}$) by $c^{\mathcal{M}}$ or $c^{\mathcal{I}^{\mathsf{c}}}$ (resp. $f^{\mathcal{M}}$ or $f^{\mathcal{I}^{\mathsf{f}}}$) for simplicity. Moreover, the mappings $\mathcal{I}^{\mathsf{c}}$ and $\mathcal{I}^{\mathsf{f}}$ naturally induce an interpretation of *terms*, i.e., a mapping from any term t in the language of $\mathcal{L}$ with $k \in \mathbb{N}$ free variables to a function $F : \mathcal{U}^k \to \mathcal{U}$, as follows. Recall that A term is a composition of function symbols and constant symbols, we replace the constant and function symbols by their interpretation in the universe $\mathcal{U}$ according to $\mathcal{I}^{\mathsf{c}}$ and $\mathcal{I}^{\mathsf{f}}$, respectively, and compose the functions and elements in the same way that the term is constructed. We will denote the induced interpretation of a term t as $t^{\mathcal{M}}$ or $t^{\mathcal{I}^{\mathsf{c}}, \mathcal{I}^{\mathsf{f}}}$ for simplicity.

Formally, one can define the mapping inductively over the structure of a term.

We say that an equation $s = t$ *satisfies* a model $\mathcal{M}$, denoted by $\mathcal{M} \vDash s = t$, if $s^{\mathcal{M}}$ is the same function as $t^{\mathcal{M}}$; a theory $\mathcal{T}$ is said to satisfy a model $\mathcal{M}$ if every equation in $\mathcal{T}$ satisfies $\mathcal{M}$. In particular, the theory of the model $\mathcal{M}$, denoted by the set of all equations satisfying $\mathcal{M}$, is the maximum theory that satisfies $\mathcal{M}$.

The standard theory of PV, denoted by $\mathbb{M}$, is defined as the tuple $(\{0,1\}^*, \mathcal{I}^{\mathsf{c}}, \mathcal{I}^{\mathsf{f}})$, where $\mathcal{I}^{\mathsf{c}}$ maps the only constant symbol ε to $\varepsilon \in \{0,1\}^*$, and $\mathcal{I}^{\mathsf{f}}$ maps the PV function symbols according to their definitions. Formally, we define the mapping $\mathcal{I}^{\mathsf{f}}_k$ from order-k PV functions to functions over $\{0,1\}^*$ inductively over $k \in \mathbb{N}$, and define $\mathcal{I}^{\mathsf{f}} = \bigcup_{k \in \mathbb{N}} \mathcal{I}^{\mathsf{f}}_k$. The mapping $\mathcal{I}^{\mathsf{f}}_k$ is defined as:

$\mathbb{M}$ is used to distinguish it from $\mathbb{N}$, i.e., the standard interpretation as natural numbers rather than strings.

- For $k = 0$, the base functions as functions satisfying their definition axioms, see Equation (2.4) to (2.7). That is, $\circ$ is the function that concatenates two strings, $\#(x, y)$ concatenates the string x with itself for $|y|$ times, $\mathsf{TR}(x)$ removes the leftmost bit of x, and $\mathsf{ITR}(x, y)$ removes the $|y|$ leftmost bits of x.

- For $k \geq 1$ consider two cases:
  - If $f = f_t^{(0)}$ is introduced by the composition rule, where t is an order-$(i-1)$ term, f is interpreted as the function $t^{\mathcal{I}^{\mathsf{c}},\mathcal{I}^{\mathsf{f}}_{k-1}}$, i.e., the interpretation of the term t based on $\mathcal{I}^{\mathsf{f}}_{k-1}$ as given in the induction hypothesis.
  - If $f = f_\Pi^{(1)}$ is introduced by the recursion rule, where $\Pi = (g, h_0, h_1, k_0, k_1, \pi_0, \pi_1)$, we define the interpretation of f as the unique function satisfying the definition axioms Equation (2.9) and (2.10) with g, h_0, h_1 replaced by their interpretation in $\mathcal{I}^{\mathsf{f}}_{k-1}$.

PV functions in the standard model. Now we show that PV functions are polynomial-time computable on the standard model, as required by the postulate of feasible functions (see Postulate 1).

Lemma 2.3.1. *The following two properties hold.*

- *For every* PV *function f, its standard interpretation $f^{\mathbb{M}} \in \mathcal{F}'$.*
- PV *satisfies its standard model.*

Proof Sketch. We prove by simultaneous induction on k that (1) every k-order PV function is interpreted as a polynomial-time computable function by $\mathcal{I}^{\mathsf{f}}_k$ and (2) every order-k provable equation $e : s = t$ satisfies that $s^{\mathcal{I}^{\mathsf{c}}_k,\mathcal{I}^{\mathsf{f}}_k}$ and $t^{\mathcal{I}^{\mathsf{c}}_k,\mathcal{I}^{\mathsf{f}}_k}$ are the same function. The base case for $k = 0$ is obvious. For $k \geq 1$, we can see that:

- For any function $f = f_t^{(0)}$ introduced by the composition rule, we know by the induction hypothesis that all order-$(k-1)$ functions are interpreted as functions in $\mathcal{F}'$ (by $\mathcal{I}^{\mathsf{f}}_k$), and thus all order-$(k-1)$ terms are interpreted as functions in $\mathcal{F}'$. It follows by the definition of $\mathcal{I}^{\mathsf{f}}_k$ that $f^{\mathcal{I}^{\mathsf{f}}_k} \in \mathcal{F}'$.

- For any function $f = f_\Pi^{(1)}$ introduced by the recursion rule, where the tuple $\Pi = (g, h_0, h_1, k_0, k_1, \pi_0, \pi_1)$, π_0, π_1 are order-$(k-1)$ proofs of Equation (2.8), and $g, h_0, h_1, k_0, k_1 \in \mathcal{F}'$. By the definition of Equation (2.8), its standard interpretation, and the induction hypothesis (2), we know that over the standard model $\mathbb{M}$,
$$|h_i^{\mathbb{M}}(\vec{x}, y, z)| \leq |z \circ k_i^{\mathbb{M}}(\vec{x}, y)| \quad i \in \{0, 1\}$$
and thus $f^{\mathbb{M}} \in \mathcal{F}'$.

- For every order-k equation $e : s = t$ that admits an order-k proof π, by induction over the length of π and a case study on the deduction rules, we can show that $s^{\mathbb{M}} = t^{\mathbb{M}}$.

This completes the proof. □

By Lemma 2.1.2 and Theorem 2.1.1, we know that every (finite-order) PV function is polynomial-time computable. On the other hand, we will prove in Chapter 4 that:

Theorem 2.3.2 ([Coo75])**.** *For every function $F \in \mathsf{FP}$, there is a (finite-order) function symbol f such that $f^{\mathbb{M}} = F$.*

This indicates that the formalization of PV is not syntactically too weak in the sense that it defines every polynomial-time computable function.

We note, however, that there might be two PV functions f_1, f_2 that are functionally equivalent in the standard model, but $f_1 = f_2$ is not PV provable — because their functional equivalence is not provable by feasible mathematics. This is expected, as PV (and feasible mathematics in general) only allows the use of the definition axioms and the induction over feasibly checkable properties, while the functional equivalence of two feasibly constructible functions may only be provable using more advanced mathematics. Here is an example:

Also note that the combination of Lemma 2.3.1 and Theorem 2.3.2 proves that $\mathcal{F}' \subseteq \mathsf{FP} = \mathcal{F}$, and thus by Lemma 2.1.2 $\mathcal{F}' = \mathcal{F} = \mathsf{FP}$.

Example 2.3.1. Let Prime be the function that $\mathsf{Prime}(x) = [x \text{ is a prime number}]$. By the celebrated AKS algorithm [AKS04] we know that $\mathsf{Prime} \in \mathsf{FP}$, and thus by Theorem 2.3.2 there is a function symbol f such that $f^{\mathbb{M}} = \mathsf{Prime}$. Then:

- Let AKS be the Turing machine implementing the AKS algorithm running in time cn^c, $\mathcal{U}(M, x, T)$ be the PV function that simulates the Turing machine $M(x)$ for $|T|$ steps. The equation $\mathcal{U}(\mathsf{AKS}, x, 1^{c|x|^c}) = f(x)$ may not be provable in PV, although it is a true equation in $\mathbb{M}$.
- Let $\mathsf{Div}, \mathsf{Le}, \mathsf{Imp}, \mathsf{And}, \mathsf{Not}$ be the PV functions: $\mathsf{Div}(x, y) := [x \mid y]$, $\mathsf{Le}(x, y) := [x < y]$, $\mathsf{Imp}(x, y) := [x \leq y]$, $\mathsf{And}(x, y) := x \times y$, $\mathsf{Not}(x) = 1 - x$. The equation

$$\mathsf{Imp}(f(x), \mathsf{Not}(\mathsf{And}(\mathsf{And}(\mathsf{Div}(y, x), \mathsf{Le}(1, y)), \mathsf{Le}(y, x)))) = 1,$$

  which formalizes the statement "$f(x) = 1$ implies y is not a non-trivial divisor of x" is a true statement, but may not be provable in PV.

$[p]$ is the function that returns 0/1 if the proposition p is false/true.

2.4 Control and Logic

Although the formal definition of PV has been quite tedious, the "programming functionality" or "instruction set" provided by PV is still limited. For instance, it is not immediately clear how Theorem 2.3.2 can be proved. Moreover, it is unclear whether PV can formalize classical algorithms, such as quick sort, in a straightforward way and prove their correctness.

Our goal is to build a "computer" or "programming language" with *feasibly provable functionality*: This will be similar to designing a CPU or programming language, except that the "underlying platform" is not (say) LLVM [LA04], but Cook's theory PV (i.e. modified Cobham's class $\mathcal{F}'$). For an analogy in mathematics, the purpose of this and subsequent chapters is similar to the first few chapters of an axiomatic set theory textbook that shows how to encode standard mathematical objects such as natural numbers in ZF or ZFC.

Glad to see you again, *real* feasible mathematicians.

We start by constructing necessary tools and proving meta-theorems about control and (propositional) logic PV.

2.4.1 If-Then-Else Function

We first show how to implement the *if-then-else* function $\mathsf{ITE}(y,u,v)$ discussed in Example 2.1.1. [5]Concretely, we will show that it is possible to define a PV-function $\mathsf{ITE}(y,u,v)$ such that the followings are provable in PV:

$$\mathsf{ITE}(s_0(x),u,v)=v, \quad \mathsf{ITE}(s_1(x),u,v)=u \tag{2.13}$$

Indeed, we define ITE to be slightly more general: it will choose to output u or v based on the rightmost bit of y.

Similar to Example 2.1.1, we will use the limited recursion rule in PV. Let $g(u,v):=\varepsilon$, $h_0(u,v,y,z):=v$, $h_1(u,v,y,z):=u$, $k_0(u,v,y):=v$, and $k_1(u,v,y)=u$. These three functions are of order-1 introduced by the rule of composition. We need to verify that for $i\in\{0,1\}$, there is a PV proof π_i for

$$\mathsf{ITR}(h_i(u,v,y,z),z\circ k_i(u,v,y))=0, \tag{2.14}$$

define $f_\Pi^{(1)}(u,v,y)$ from $\Pi:=(g,h_0,h_1,k_0,k_1,\pi_0,\pi_1)$, and then it is easy to see that Equation (2.13) follows if we define $\mathsf{ITE}(y,u,v)=f_\Pi^{(1)}(u,v,y)$.

It remains to prove Equation (2.14). Indeed, both of the cases for $i\in\{0,1\}$ can be derived from the following equation using the definition axioms of h_i and k_i as well as the substitution rules:

Proposition 2.4.1. $\mathsf{PV}\vdash\mathsf{ITR}(x,z\circ x)=\varepsilon$.

Informally, the proof will be done by induction on x. Let $f_1(z,x):=\mathsf{ITR}(x,z\circ x)$ and $f_2(z,x)=\varepsilon$. We first consider the base case x/ε. It is easy to see that $\mathsf{PV}\vdash f_2(z,\varepsilon)=\varepsilon$ by (L4) substitution, and therefore we only need to prove:

Proposition 2.4.2. $\mathsf{PV}\vdash f_1(z,\varepsilon)=\varepsilon$.

Proof. We need to prove in PV that $\mathsf{ITR}(\varepsilon,z\circ\varepsilon)=\varepsilon$. Notice that $z\circ\varepsilon=z$, and thus by applying (L3) on the function $w\mapsto\mathsf{ITR}(\varepsilon,w\circ\varepsilon)$ we know that $\mathsf{ITR}(\varepsilon,z\circ\varepsilon)=\mathsf{ITR}(\varepsilon,z)$. By transitivity, we only need to prove in PV that

$$\mathsf{ITR}(\varepsilon,x)=\varepsilon. \tag{2.15}$$

It can be proved using the rule of structural induction. Concretely, we will define functions $f_3(x)=\mathsf{ITR}(\varepsilon,x)$ and $f_4(x)=\varepsilon$, and prove that both of them can be constructed by recursion from $g=\varepsilon$, $h_i(x,z)=\mathsf{TR}(z)$. The details are omitted and left as an exercise. □

Now we consider the case when x is of form $s_i(x)$. Note that $\mathsf{PV}\vdash f_1(z,s_i(x))=\mathsf{ITR}(s_i(x),z\circ s_i(x))$ by (L4) substitution, and to apply the induction rule, we need to see how $f_1(z,s_i(x))$ can be constructed as a function of z, x, and $f_1(z,x)$. Notice that by the definition axiom of ITR and (L4) substitution we know that

$$\mathsf{PV}\vdash\mathsf{ITR}(s_i(x),z\circ s_i(x))=\mathsf{TR}(\mathsf{ITR}(s_i(x),z\circ x)).$$

[5] We will encode `False` with 0 (i.e. $s_0(\varepsilon)$) and `True` with 1 (i.e. $s_1(\varepsilon)$). Alternatively, one may encode them using (say) ε and 0.

By thinking about its interpretation in the standard model $\mathbb{M}$, it is easy to see that

$$\mathsf{TR}(\mathsf{ITR}(s_i(x), z \circ x)) = \mathsf{ITR}(x, z \circ x) \tag{2.16}$$

and the RHS of the equation is indeed $f_1(z, x)$ — completing our goal of constructing $f_1(z, s_i(x))$ as a function of z, x and $f_1(z, x)$. Therefore it suffices to prove that:

Proposition 2.4.3. $\mathsf{PV} \vdash \mathsf{TR}(\mathsf{ITR}(s_i(x), z \circ x)) = \mathsf{ITR}(x, z \circ x)$.

To prove this, we need the fact that ITR and TR commute:

Proposition 2.4.4. $\mathsf{PV} \vdash \mathsf{TR}(\mathsf{ITR}(x, y)) = \mathsf{ITR}(\mathsf{TR}(x), y)$.

Proof Sketch. We will prove by induction on y that both sides of the equation are identical to the feasibly constructive function recursively defined from $g'(x) = \mathsf{TR}(x)$ and $h'_i(x, y, z) = \mathsf{TR}(z)$.

To see this for the LHS, notice that $\mathsf{PV} \vdash \mathsf{TR}(\mathsf{ITR}(x, \varepsilon)) = \mathsf{TR}(x)$ by unfolding ITR, and that for $i \in \{0, 1\}$,

$$\mathsf{PV} \vdash \mathsf{TR}(\mathsf{ITR}(x, s_i(y))) = \mathsf{TR}(\mathsf{TR}(\mathsf{ITR}(x, y))).$$

by unfolding ITR.

To see this for the RHS, notice that $\mathsf{ITR}(\mathsf{TR}(x), \varepsilon) = \mathsf{TR}(x)$ by unfolding ITR, and that for $i \in \{0, 1\}$,

$$\mathsf{PV} \vdash \mathsf{ITR}(\mathsf{TR}(x), s_i(y)) = \mathsf{TR}(\mathsf{ITR}(\mathsf{TR}(x), y))$$

by unfolding ITR. Therefore, we can prove the original equation by the structural induction rule in PV using $g'(x) = \mathsf{TR}(x)$ and $h'_i(x, y, z) = \mathsf{TR}(z)$. □

Proof of Proposition 2.4.3. By Proposition 2.4.4 with the substitutions $x/s_i(x)$ and $y/z \circ x$, we know that the LHS of the equation is equal to $\mathsf{ITR}(\mathsf{TR}(s_i(x)), z \circ x)$. Note that $\mathsf{PV} \vdash \mathsf{TR}(s_i(x)) = x$ by the definition equation of TR, therefore we can unfold TR and further prove that $\mathsf{ITR}(\mathsf{TR}(s_i(x)), z \circ x) = \mathsf{ITR}(x, z \circ x)$, which is exactly the RHS of the equation. □

Therefore, by applying Proposition 2.4.3, we notice that for $i \in \{0, 1\}$,

$$\mathsf{PV} \vdash f_1(z, s_i(x)) = h_i(z, x, f_1(z, x))$$

for $h_i(z, x, z') := z'$. Also, notice that for $i \in \{0, 1\}$

$$\mathsf{PV} \vdash f_2(z, s_i(x)) = h_i(z, x, f_2(z, x))$$

as both sides of the equation directly evaluate to ε. Therefore, by the induction rule, we can prove that $f_1(z, x) = f_2(z, x)$, which leads to the proof of Proposition 2.4.1.

Finally, I gave up the plan of always using z as the last variable while writing the function h_i; but z' is not that bad, I guess.

Finally, one can easily verify Equation (2.13) are provable in PV by the definition axioms

$$f_\Pi^{(1)}(u, v, s_i(y)) = h_i(u, v, y, f_\Pi^{(1)}(u, v, y)) \quad (i \in \{0, 1\})$$

of $f_\Pi^{(1)}$ and the definition axiom $\mathsf{ITE}(y,u,v) = f_\Pi^{(1)}(u,v,y)$ of ITE. Therefore, we will safely refer ITE as the function defined here and use Equation (2.13) while writing PV proofs.

We note that an additional benefit of our definition is that PV proves

$$\mathsf{ITE}(\varepsilon,u,v) = \varepsilon. \tag{2.17}$$

We can treat ε as a symbol for "error" or "undefined", which may be useful at some point.

Remark 2.4.1. Another application of Proposition 2.4.1 is to show that that the function $x \# y$ in PV and its definition equations are indeed redundant. This is because we can define the function via limited recursion by $g(x) = \varepsilon$, $h_i(x,y,z) = z \circ x$, $k_i(x) = x$ where $i \in \{0,1\}$. The requirement for limited recursion is that $\mathsf{PV} \vdash \mathsf{ITE}(z \circ x, z \circ x) = \varepsilon$, which can be obtained from Proposition 2.4.1 by (L4) substitutions z/ε, $x/z \circ x$, and applying Proposition 2.4.2.

The function $\mathsf{LastBit}$. As an immediate application, we can define the function $\mathsf{LastBit}(x)$ as $\mathsf{LastBit}(x) = \mathsf{ITE}(x,1,0)$. It can be easily proved from the PV proofs of Equation (2.13) that

$$\mathsf{PV} \vdash \mathsf{LastBit}(\varepsilon) = \varepsilon; \quad \mathsf{PV} \vdash \mathsf{LastBit}(s_i(x)) = i \tag{2.18}$$

for $i \in \{0,1\}$.

IsEps and $\mathsf{IsNotEps}$. Similar to ITE, we can define functions that determine whether a string is ε or not. Let $\mathsf{IsEps}(x)$ and $\mathsf{IsNotEps}(x)$ be defined as follows:

$$\mathsf{IsEps}(\varepsilon) = 1; \quad \mathsf{IsEps}(s_i(x)) = 0 \quad (i \in \{0,1\}) \tag{2.19}$$

$$\mathsf{IsNotEps}(\varepsilon) = 0; \quad \mathsf{IsNotEps}(s_i(x)) = 1 \quad (i \in \{0,1\}) \tag{2.20}$$

Notice that: $\mathsf{IsEps}(x)$ can be defined using limited recursion from $g = 1$, $h_i(x,z) = 0$, and $k_i(x) = 0$; and $\mathsf{IsNotEps}(x)$ can be defined from $g = 0$, $h_i(x,z) = 1$, and $k_i(x) = 1$. The equations $\mathsf{ITR}(h_i(x,z), z \circ k_i(x)) = 0$ can be proved in PV using Proposition 2.4.1 by unfolding k_i, h_i and applying (L4) substitution to the variable x in Proposition 2.4.1.

2.4.2 Proof by Case Study

We now introduce a meta-theorem that implements the idea of proof by case study. Let $s(\vec{x},y) = t(\vec{x},y)$ be an equation. A standard way to prove the equation is to consider whether the last bit of y is 0, 1, or $y = \varepsilon$. Formally:

Theorem 2.4.5. *Let $s(\vec{x},y) = t(\vec{x},y)$ be an equation. If*

- $\mathsf{PV} \vdash s(\vec{x},\varepsilon) = t(\vec{x},\varepsilon)$*;*
- $\mathsf{PV} \vdash s(\vec{x},s_i(y)) = t(\vec{x},s_i(y))$ *for $i \in \{0,1\}$;*

then $\mathsf{PV} \vdash s(\vec{x},y) = t(\vec{x},y)$*.*

Proof. We define two functions $f_s(\vec{x}, y) = s(\vec{x}, y)$ and $f_t(\vec{x}, y) = t(\vec{x}, y)$ by the rule of composition. We will prove that both f_s and f_t are identical to the function inductively defined by some $g(\vec{x}), h_0(\vec{x}, y, z), h_1(\vec{x}, y, z)$. Indeed, we will simply choose

$$\begin{aligned} g(\vec{x}) &:= f_s(\vec{x}, \varepsilon) \\ h_i(\vec{x}, y, z) &:= f_s(\vec{x}, s_i(y)) \quad (i \in \{0, 1\}); \end{aligned}$$

that is, we define g, h_0, h_1 based on the LHS of the equation to prove. We need to prove that:

$$\begin{aligned} &\mathsf{PV} \vdash f_t(\vec{x}, \varepsilon) = g(\vec{x}) \\ &\mathsf{PV} \vdash f_t(\vec{x}, s_i(y)) = h_i(\vec{x}, y, z) \quad (i \in \{0, 1\}). \end{aligned}$$

Indeed, by unfolding the definition of g and h_i and applying (L2) transitivity, these three equations are exactly the equations in the assumption. This completes the proof. □

Case on ITE. A typical application of the theorem is to perform a case study on the condition in the if-then-else function, summarized as the following meta-theorem:

Theorem 2.4.6. *Let $f_c(\vec{x}), f_0(\vec{x}), f_1(\vec{x}), g(\vec{x}, y), f'_0(\vec{x}), f'_1(\vec{x}), g'(\vec{x}, y)$ are* PV *functions. Suppose that*

- $\mathsf{PV} \vdash g(\vec{x}, \varepsilon) = g'(\vec{x}, \varepsilon)$
- $\mathsf{PV} \vdash g(\vec{x}, f_i(\vec{x})) = g'(\vec{x}, f'_i(\vec{x}))$ *for* $i \in \{0, 1\}$*;*

then $\mathsf{PV} \vdash g(\vec{x}, \mathsf{ITE}(f_c(\vec{x}), f_1(\vec{x}), f_0(\vec{x}))) = g'(\vec{x}, \mathsf{ITE}(f_c(\vec{x}), f'_1(\vec{x}), f'_2(\vec{x})))$.

Proof Sketch. We can prove a stronger result that

$$\mathsf{PV} \vdash g(\vec{x}, \mathsf{ITE}(z, f_1(\vec{x}), f_0(\vec{x}))) = g'(\vec{x}, \mathsf{ITE}(z, f'_1(\vec{x}), f'_0(\vec{x})))$$

and the theorem follows by performing (L4) substitution $z/f_c(\vec{x})$ to the equation. To prove this stronger result, we perform a case study on the variable z using Theorem 2.4.5, and the three cases (after unfolding ITE) will be the three cases in the assumption. For instance, consider the case that $z = \varepsilon$ in Theorem 2.4.5, we need to prove that

$$\mathsf{PV} \vdash g(\vec{x}, \mathsf{ITE}(\varepsilon, f_1(\vec{x}), f_0(\vec{x}))) = g'(\vec{x}, \mathsf{ITE}(\varepsilon, f'_1(\vec{x}), f'_0(\vec{x}))). \tag{2.21}$$

Notice that $\mathsf{PV} \vdash \mathsf{ITE}(\varepsilon, x, y) = \varepsilon$ by the definition axiom of ITE Equation (2.17). Then, by applying (L4) substitution $x/f_1(\vec{x})$ and subsequently $y/f_0(\vec{x})$, we have

This "unfolding" trick will be used multiple times; we will simply say "unfolding" later on.

$$\begin{aligned} &\mathsf{PV} \vdash \mathsf{ITE}(\varepsilon, f_1(\vec{x}), f_0(\vec{x})) = \varepsilon \\ &\mathsf{PV} \vdash \mathsf{ITE}(\varepsilon, f'_1(\vec{x}), f'_0(\vec{x})) = \varepsilon. \end{aligned}$$

We now apply (L3) to the equations above with the PV function $y \mapsto g(\vec{x}, y)$ to obtain

$$\begin{aligned} &\mathsf{PV} \vdash g(\vec{x}, \mathsf{ITE}(\varepsilon, f_1(\vec{x}), f_0(\vec{x}))) = g(\vec{x}, \varepsilon) \\ &\mathsf{PV} \vdash g'(\vec{x}, \mathsf{ITE}(\varepsilon, f'_1(\vec{x}), f'_0(\vec{x}))) = g(\vec{x}, \varepsilon) \end{aligned}$$

Recall that $\mathsf{PV} \vdash g(\vec{x}, \varepsilon) = g'(\vec{x}, \varepsilon)$ holds by the assumption. We can therefore obtain Equation (2.21) by applying (L2) transitivity. The other cases can be proved accordingly using a similar approach. □

Remark 2.4.2. We further note that the same proof technique leads to a more general result: We can perform case analysis even if there are multiple ITE's in g and g' with the same condition $f_c(\vec{x})$.

Suppose we have sequences of PV functions $\vec{f_1}(\vec{x}), \vec{f_0}(\vec{x}), \vec{f_1'}(\vec{x}), \vec{f_0'}(\vec{x})$ each of which has say $k \in \mathbb{N}$ functions, e.g., $\vec{f_1}(\vec{x}) = (f_1^{(1)}(\vec{x}), \dots, f_1^{(k)}(\vec{x}))$, and both g and g' takes $\vec{x}$ and k additional variables. Suppose that we define $f_{-1}^{(j)}(\vec{x}), f_{-1}^{(j)}(\vec{x})$ as ε for simplicity of the notation. If PV proves for each $\vec{j} \in \{0, 1, -1\}^k$ that

$$g(\vec{x}, f_{j_1}^{(1)}(\vec{x}), \dots, f_{j_k}^{(k)}(\vec{x})) = g'(\vec{x}, {f'}_{j_1}^{(1)}(\vec{x}), \dots, {f'}_{j_k}^{(k)}(\vec{x}))),$$

Then PV also proves that

$$\begin{aligned} &g(\vec{x}, \mathsf{ITE}(f_c(\vec{x}), f_1^{(1)}(\vec{x}), f_0^{(1)}(\vec{x})), \dots, \mathsf{ITE}(f_c(\vec{x}), f_1^{(k)}(\vec{x}), f_0^{(k)}(\vec{x}))) \\ = {}&g'(\vec{x}, \mathsf{ITE}(f_c(\vec{x}), {f'}_1^{(1)}(\vec{x}), {f'}_0^{(1)}(\vec{x})), \dots, \mathsf{ITE}(f_c(\vec{x}), {f'}_1^{(k)}(\vec{x}), {f'}_0^{(k)}(\vec{x}))), \end{aligned}$$

This also holds for the case study on "dirty" ITE that will be discussed below.

Why do I know this? There is no magic: I stuck at Section 2.5.1 and found this necessary.

Case on "dirty" ITE. We will occasionally need an alternative version of Theorem 2.4.6: When it can be feasibly proved that $f_c(\vec{x})$ is not ε, or not of form $s_i(y)$ for some $i \in \{0, 1\}$, we will be able to remove one of the three assumptions of Theorem 2.4.6. To formalize this method, we will define a "dirty" ITE, denoted by $\mathsf{ITE}_j^{(\varepsilon)}(x, u_1, u_0)$ as follows:

$$\mathsf{ITE}_j^{(\varepsilon)}(\varepsilon, u_1, u_0) = u_j \quad \mathsf{ITE}^{(\varepsilon)}(s_i(x), u_1, u_0) = u_i \quad (i \in \{0, 1\}). \tag{2.22}$$

The formal definition of the function is similar to the definition of ITE and is left as an exercise. Informally, this function is called "dirty" ITE as it does not "correctly" deal with the case for $x = \varepsilon$. Then we have

Theorem 2.4.7. *Let $f_c(\vec{x}), f_0(\vec{x}), f_1(\vec{x}), g(\vec{x}, y), f_0'(\vec{x}), f_1'(\vec{x}), g'(\vec{x}, y)$ are PV functions. Suppose that*

- $\mathsf{PV} \vdash g(\vec{x}, f_i(\vec{x})) = g'(\vec{x}, f_i'(\vec{x}))$ *for* $i \in \{0, 1\}$*;*

then $\mathsf{PV} \vdash g(\vec{x}, \mathsf{ITE}_j^{(\varepsilon)}(f_c(\vec{x}), f_1(\vec{x}), f_0(\vec{x}))) = \mathsf{PV} \vdash g'(\vec{x}, \mathsf{ITE}_j^{(\varepsilon)}(f_c(\vec{x}), f_1'(\vec{x}), f_0'(\vec{x})))$ *for* $j \in \{0, 1\}$*.*

The proof is similar to Theorem 2.4.6 and is left as an exercise. As a corollary:

Corollary 2.4.8. *Let $f_c(\vec{x}), f_0(\vec{x}), f_1(\vec{x}), g(\vec{x}, y), f_0'(\vec{x}), f_1'(\vec{x}), g'(\vec{x}, y)$ are PV functions, $j \in \{0, 1\}$. Suppose that*

- $\mathsf{PV} \vdash \mathsf{ITE}_j^{(\varepsilon)}(f_c(\vec{x}), u_1, u_0) = \mathsf{ITE}(f_c(\vec{x}), u_1, u_0)$*;*
- $\mathsf{PV} \vdash g(\vec{x}, f_i(\vec{x})) = g'(\vec{x}, f_i'(\vec{x}))$ *for* $i \in \{0, 1\}$*;*

then $\mathsf{PV} \vdash g(\vec{x}, \mathsf{ITE}_j^{(\varepsilon)}(f_c(\vec{x}), f_1(\vec{x}), f_0(\vec{x}))) = g'(\vec{x}, \mathsf{ITE}_j^{(\varepsilon)}(f_c(\vec{x}, f_1'(\vec{x}), f_0'(\vec{x}))))$*.*

Remark 2.4.3. Similarly, we can define "dirty" ITE that does not correctly deal with the case where x is of form $s_i(y)$ for some $i \in \{0, 1\}$, and prove a meta-theorem similar to Theorem 2.4.7.

2.4.3 Propositional Logic: And, Or, Not

An important application of the ITE function is to simulate propositional logic, i.e., implementing the connectives $\mathsf{And}, \mathsf{Or}, \mathsf{Not}$. They are naturally defined as

$$\mathsf{And}(x, y) := \mathsf{ITE}(x, \mathsf{ITE}(y, 1, 0), \mathsf{ITE}(y, 0, 0)) \tag{2.23}$$

$$\mathsf{Or}(x, y) := \mathsf{ITE}(x, \mathsf{ITE}(y, 1, 1), \mathsf{ITE}(y, 1, 0)) \tag{2.24}$$

$$\mathsf{Not}(x) := \mathsf{ITE}(x, 0, 1) \tag{2.25}$$

by compositions of ITE.

One may wonder why we write $\mathsf{ITE}(y, 0, 0)$ instead of 0 in the definition equation of And, and $\mathsf{ITE}(y, 1, 1)$ instead of 1 in the definition equation of Or. It prevents *short-circuit evaluation*, i.e., it ensures that if one of the inputs is ε, the output of the function is always ε. Therefore, we can use ε to encode "undefined" or "error".

Therefore, we can embed an arbitrary propositional formula with constants and variables into a term in PV. We will show a powerful meta-theorem showing that PV is sound and complete as a propositional proof system (PPS for short) using this embedding.

Theorem 2.4.9 (PV as a PPS). *Let φ_1, φ_2 be propositional formulas consisting of variables and constants (i.e. $\texttt{True}$ or $\texttt{False}$), and $p_{\varphi_1}, p_{\varphi_2}$ be the PV-terms by replacing propositional connectives with $\{\mathsf{And}, \mathsf{Or}, \mathsf{Not}\}$ and replacing $\texttt{True}$ and $\texttt{False}$ with $s_1(t)$ and $s_0(t')$ for arbitrary terms t and t' (not necessarily the same for all replacements). Assume that there is no variable that appears in exactly one of φ_1 and φ_2. Then: $\varphi_1 \equiv \varphi_2$ if and only if $\mathsf{PV} \vdash \mathsf{LastBit}(p_{\varphi_1}) = \mathsf{LastBit}(p_{\varphi_2})$.*

Proof. One side is easy: If $p_{\varphi_1} = p_{\varphi_2}$ admits a PV proof, the equation holds in the standard model $\mathbb{M}$. By the interpretation of ITE in $\mathbb{M}$, we can see that it immediately implies that for all assignments to variables $\vec{x} \in \{\texttt{True}, \texttt{False}\}^*$, $\varphi_1(\vec{x}) = \varphi_2(\vec{x})$, which implies that $\varphi_1 \equiv \varphi_2$. (In the rest of the proof, we identify 0 and $\texttt{False}$, as well as 1 and $\texttt{True}$.)

The proof is not feasibly constructive as it involves the standard model $\mathbb{M}$.

Now we prove the other side. Let $k \in \mathbb{N}$ be the number of variables involved in φ_1, φ_2. We prove that there is a PV-proof of $p_{\varphi_1} = p_{\varphi_2}$ for every φ_1, φ_2 such that $\varphi_1 \equiv \varphi_2$ by induction on k. (Note that this induction is not made within PV but in our meta-theory.) The case for $k = 0$ is simple: By referring to the axioms of ITE (see Equation (2.13)) and using the logical rules for equation (as well as using the meta-theorem Theorem 2.4.6), we can prove in PV for some $b \in \{0, 1\}$ that $\mathsf{LastBit}(p_{\varphi_1}) = b$ and $\mathsf{LastBit}(p_{\varphi_2}) = b$, i.e., we can evaluate both sides of the equation. Thus by (L1) symmetricity and (L2) transitivity, we can prove that $p_{\varphi_1} = p_{\varphi_2}$.

Now, assuming that it is true for all φ_1, φ_2 consisting of at most k variables, we want to prove the case for $k + 1$. Let $\varphi_1'(\vec{x}, y)$ and $\varphi_2'(\vec{x}, y)$ be formulas consisting of at most $k + 1$ variables and every variable in $\{\vec{x}, y\}$ appears in both φ_1' and φ_2', where $|\vec{x}| = k$. Assume that $\varphi_1'(\vec{x}, y) \equiv \varphi_2'(\vec{x}, y)$. Let $p_{\varphi_1'}(\vec{x}, y)$ and $p_{\varphi_2'}(\vec{x}, y)$ be the terms embedding φ_1' and φ_2' into PV as mentioned above, respectively. Notice that for $i \in \{0, 1\}$, $j \in \{1, 2\}$, and a fresh variable w, we will have that $p_{\varphi_j'}(\vec{x}, s_i(w))$ is a valid translation of the formula $\varphi_j'(\vec{x}, y/i)$ obtained by replacing occurrences of y in φ_j' to i as mentioned above, i.e., by

To make everything super formal, we can prove by induction on the formation of the formulas.

- replacing propositional connectives with $\{\mathsf{And}, \mathsf{Or}, \mathsf{Not}\}$,
- replacing $\texttt{True}$ and $\texttt{False}$ with $s_1(t)$ and $s_0(t')$ for arbitrary terms t, t'.

Notice that as $\varphi_1' \equiv \varphi_2'$, we know that $\varphi_1'(\vec{x}, y/i) \equiv \varphi_2'(\vec{x}, y/i)$ for $i \in \{0, 1\}$. Therefore, by the induction hypothesis, there are PV-proofs of

$$\mathsf{LastBit}(p_{\varphi_1'}(\vec{x}, s_i(w))) = \mathsf{LastBit}(p_{\varphi_2'}(\vec{x}, s_i(w))) \tag{2.26}$$

for $i \in \{0, 1\}$. Also, we can prove (by Equation (2.17) and Equation (2.18)) that

$$\mathsf{LastBit}(p_{\varphi_1'}(\vec{x}, \varepsilon)) = \mathsf{LastBit}(p_{\varphi_2'}(\vec{x}, \varepsilon)); \tag{2.27}$$

indeed, PV proves that both sides of the equation are equal to ε (recall that y appears in both φ_1' and φ_2'). We can then prove

$$\mathsf{LastBit}(p_{\varphi_1'}(\vec{x}, y)) = \mathsf{LastBit}(p_{\varphi_2'}(\vec{x}, y)) \tag{2.28}$$

by applying Theorem 2.4.5 (i.e. proof by case study) on y. □

Obviously, I'm being a bit more sloppy while writing proofs in meta-theory compared to writing proofs in PV. Hopefully, there is no confusion for readers.

Remark 2.4.4. We note that the reason to put $\mathsf{LastBit}(\cdot)$ outside of both p_{φ_1} and p_{φ_2} is to deal with the case that p_{φ_i} is not wrapped by $\mathsf{And}, \mathsf{Or}, \mathsf{Not}$. The assumption that no variable appears in exactly one of φ_1 and φ_2 is necessary because it can be the case that the variable is ε in PV so that one side is ε and another side is 0 or 1.

2.4.4 Conditional Equations

Another application of the if-then-else function is to implement *conditional equations*. Let t_c be a term and t_1, t_2 be terms. (As we will not deal with variables in this section, we hide all $\vec{x}$ in terms.) The conditional equation $t_c \Rightarrow t_1 = t_2$ is defined as the PV equation

$$\mathsf{ITE}(t_c, t_2, t_1) = t_1.$$

We call t_c the *antecedent* of the conditional equation, while $t_1 = t_2$ is the *consequence* of the conditional equation.

Intuitively, this equation is true if $\mathsf{LastBit}(t_c) = 0$ or $t_1 = t_2$ and is false if $\mathsf{LastBit}(t_c) = 1$ and $t_1 \neq t_2$. The case for $t_c = \varepsilon$ is considered a non-defined behavior.

It is not hard to see that:

Proposition 2.4.10 (Modus Ponens)**.** *If* PV *proves* $t_c \Rightarrow t_1 = t_2$ *and* $t_c = 1$*, then* $\mathsf{PV} \vdash t_1 = t_2$*.*

Proof. If $\mathsf{PV} \vdash t_c = 1$, we know that $\mathsf{ITE}(t_c, y, x) = x$ by unfolding ITE. Therefore, $\mathsf{PV} \vdash \mathsf{ITE}(t_c, t_2, t_1) = t_2$ by substitution, and thus $t_1 = t_2$ by transitivity (as $\mathsf{PV} \vdash \mathsf{ITE}(t_c, t_2, t_1) = t_1$ by the assumption). □

Proposition 2.4.11 (Explosion Rule)**.** PV *proves that* $s_0(z) \Rightarrow x = y$*.*

Proof. By unfolding the definition, we need to prove that $\mathsf{ITE}(s_0(z), y, x) = x$, where the LHS of the equation PV-provably evaluates to x. □

Conditioning is an important tool in PV, and we will see later on that with conditional equations we can make it possible for PV to encode *equations*, which is the key to developing useful meta-theorems in the next chapter. Here we provide a simple example of how conditioning could help:

Theorem 2.4.12. *Let t be a PV-term. Suppose that PV proves $\mathsf{IsEps}(t) = 1$, then PV also proves $t = \varepsilon$.*

Proof. We first prove that $\mathsf{IsEps}(x) \Rightarrow x = \varepsilon$. This can be proved by a simple case study using Theorem 2.4.5. Suppose that $x = \varepsilon$ the conditional equation PV-provably evaluates to $\varepsilon = \varepsilon$, which is true by (L0) reflexivity. Otherwise, $\mathsf{PV} \vdash \mathsf{IsEps}(s_i(x)) = 0$ and by the explosion rule, PV also proves the equation.

By (L3) substitution, we can then prove that $\mathsf{IsEps}(t) \Rightarrow t = \varepsilon$, and thus by Modus Ponens, we can conclude that $\mathsf{PV} \vdash t = \varepsilon$. □

2.5 Basic Data Structures in PV

In this section, we move on to design basic data structures for the programming language of feasible mathematicians. We will design *pairs* and *tuples* that are critical to implement algorithms in PV in a natural sense.

2.5.1 Pairs

We first show how to encode pairs in PV. To support making and unwinding pairs, we need to implement three PV functions: $\mathsf{MakePair}(x, y)$ intended to make a pair $\tau = (x, y)$, $\mathsf{Left}(\tau) = x$, and $\mathsf{Right}(\tau) = y$. We also need to ensure that the properties of pairs are PV-provable:

$$\mathsf{Left}(\mathsf{MakePair}(x, y)) = x, \quad \mathsf{Right}(\mathsf{MakePair}(x, y)) = y. \tag{2.29}$$

Description of encoding. A standard trick of encoding a pair (x, y) using a string is to encode y with the alphabet $\{00, 10\}$ and use 11 as a comma which separates x and y.

We define a couple of functions in turn. Let $\mathsf{PLen}(x)$ be the function that outputs 1 (i.e. $s_1(\varepsilon)$) if $|x|$ is odd and 0 (i.e. $s_0(\varepsilon)$) if $|x|$ is even. This can be easily defined by limited recursion as

$$\mathsf{PLen}(\varepsilon) := 0 \tag{2.30}$$

$$\mathsf{PLen}(s_i(x)) := \mathsf{Not}(\mathsf{PLen}(x)) \quad (i \in \{0, 1\}) \tag{2.31}$$

We define $\mathsf{PEnc}(x)$, $\mathsf{PDec}(y)$ be the encoding and decoding functions from the alphabet

$\{0,1\}$ to the alphabet $\{00,10\}$, as follows:[6]

$$\mathsf{PEnc}(\varepsilon) := \varepsilon \tag{2.32}$$

$$\mathsf{PEnc}(s_i(x)) := s_0(s_i(\mathsf{PEnc}(x))) \quad (i \in \{0,1\}) \tag{2.33}$$

$$\mathsf{PDec}(\varepsilon) := \varepsilon \tag{2.34}$$

$$\mathsf{PDec}(s_i(x)) := \mathsf{ITE}(\mathsf{LastBit}(\mathsf{PLen}(x)), s_i(\mathsf{PDec}(x)), \mathsf{PDec}(x)) \tag{2.35}$$

To formally define these functions in PV, we need to apply the limited recursion rule using some g, h_0, h_1, k_0, k_1. We take PDec as an example:

$$\begin{aligned} g &:= \varepsilon \\ h_i(x,z) &:= \mathsf{ITE}(\mathsf{LastBit}(\mathsf{PLen}(x)), s_i(z), z) \quad (i \in \{0,1\}) \\ k_i(x) &:= s_i(\varepsilon) \quad (i \in \{0,1\}) \end{aligned}$$

and we need to prove that:

Proposition 2.5.1. $\mathsf{PV} \vdash \mathsf{ITR}(h_i(x,z), z \circ k_i(x))$ *for* $i \in \{0,1\}$.

Proof Sketch. The idea is to perform case analysis on the condition $\mathsf{LastBit}(\mathsf{PLen}(x))$ using Theorem 2.4.6. It suffices to prove in PV that $\mathsf{ITR}(\varepsilon, z \circ \varepsilon) = 0$ and $\mathsf{ITR}(s_i(z), z \circ i) = 0$ for $i \in \{0,1\}$. The former equation follows from Proposition 2.4.1 by applying (L4) substitution z/ε, while the later equation (for $i \in \{0,1\}$) follows from $\mathsf{PV} \vdash \mathsf{ITR}(x,x) = 0$, which can be prove following the proof of Proposition 2.4.1. The details are omitted and left as an exercise. □

It can be verified that:

Proposition 2.5.2. $\mathsf{PV} \vdash \mathsf{PLen}(\mathsf{PEnc}(x)) = 0$.

Proof Sketch. We prove by induction on x. More formally, let $f_1(x) := \mathsf{PLen}(\mathsf{PEnc}(x))$ and $f_2(x) = 0$, it can be verified that both $f_1(x)$ and $f_2(x)$ are identical to the function recursively defined by $g = 0$ and $h_i(x,z) := z$ $(i \in \{0,1\})$. □

Proposition 2.5.3. $\mathsf{PV} \vdash \mathsf{PDec}(\mathsf{PEnc}(x)) = x$.

Proof Sketch. We prove by induction on x. More formally, let $f_1(x) := \mathsf{PDec}(\mathsf{PEnc}(x))$ and $f_2(x) = x$, it can be verified that both $f_1(x)$ and $f_2(x)$ are identical to the function recursively defined by $g = \varepsilon$ and $h_i(x,z) = s_i(z)$. To see how to prove

$$\mathsf{PV} \vdash f(s_i(x)) = h_i(x, f(x)),$$

notice that (by applying definition axioms) the LHS is PV provably equal to:

$$\begin{aligned} & \mathsf{ITE}(\mathsf{LastBit}(\mathsf{PLen}(s_0(s_i(\mathsf{PEnc}(x))))), s_0(s_i(\mathsf{PDec}(\mathsf{PEnc}(x)))), \mathsf{PDec}(s_0(s_i((\mathsf{PEnc}(x)))))) \\ = {} & \mathsf{ITE}(\mathsf{LastBit}(\mathsf{Not}(\mathsf{Not}(\mathsf{PLen}(\mathsf{PEnc}(x))))), s_0(s_i(\mathsf{PDec}(\mathsf{PEnc}(x)))), \mathsf{PDec}(s_0(s_i(\mathsf{PEnc}(x))))) \\ = {} & \mathsf{ITE}(\mathsf{LastBit}(\mathsf{PLen}(\mathsf{PEnc}(x))), s_0(s_i(f(x))), \mathsf{PDec}(s_0(s_i(\mathsf{PEnc}(x))))) \\ = {} & \mathsf{ITE}(0, s_0(s_i(f(x))), \mathsf{PDec}(s_0(s_i(\mathsf{PEnc}(x))))) && \text{(Proposition 2.5.2)} \\ = {} & \mathsf{PDec}(s_0(s_i(\mathsf{PEnc}(x)))) \end{aligned}$$

[6] Wrapping $\mathsf{LastBit}$ outside of the condition for ITE helps to apply Theorem 2.4.9.

Note that the second equation applied Theorem 2.4.9 with the propositional formula $\neg\neg x = x$. Again, after a sequence of tedious but straightforward unfolding, the last line is PV-provably equal to $s_i(f(x))$, which completes the proof. □

Pairing and unpacking functions. Then, we can simply define $\mathsf{MakePair}(x, y)$ as

Recall that $11 = s_1(s_1(\varepsilon))$

$$\mathsf{MakePair}(x, y) := x \circ 11 \circ \mathsf{PEnc}(y). \tag{2.36}$$

Then we define the functions $\mathsf{Left}(\tau)$ and $\mathsf{Right}(\tau)$. Notice that if we can define $\mathsf{Right}(\tau)$, $\mathsf{Left}(\tau)$ can be simply defined as

$$\mathsf{Left}(\tau) := \mathsf{TR}(\mathsf{TR}(\mathsf{ITR}(\tau, \mathsf{PEnc}(\mathsf{Right}(\tau))))). \tag{2.37}$$

Therefore, we now focus on defining $\mathsf{Right}(\tau)$. The idea is to read the encoding of y bit by bit until touching the comma "11", and then decode the string using PDec.

We first define a function $\mathsf{RightRaw}(\tau)$ that reads a suffix of τ before "11":

$$\mathsf{RightRaw}(\varepsilon) := \varepsilon \tag{2.38}$$
$$\mathsf{RightRaw}(s_0(\tau)) := s_0(\mathsf{RightRaw}(\tau)) \tag{2.39}$$
$$\mathsf{RightRaw}(s_1(\tau)) := \mathsf{ITE}(\tau, 1, s_1(\mathsf{RightRaw}(\tau))) \tag{2.40}$$

The formal definition of RightRaw in PV should be clear and left as an exercise. Note that there could be two cases: $\mathsf{RightRaw}(\mathsf{MakePair}(x, y))$ may be equal to $1 \circ \mathsf{PEnc}(y)$ if the leftmost bit of y is 0 or $y = \varepsilon$, and is equal to $\mathsf{PEnc}(y)$ otherwise. We need to define a function that removes the leftmost 1 when the string is of odd length.

Recall that $\mathsf{IsEps}(x)$ is the PV function that determines whether $x = \varepsilon$. Let RT and CleanLeft be defined as follows

$$\mathsf{RT}(\varepsilon) := \varepsilon \tag{2.41}$$
$$\mathsf{RT}(s_i(x)) := \mathsf{ITE}(\mathsf{IsEps}(x), \varepsilon, s_i(\mathsf{RT}(x))) \quad (i \in \{0, 1\}) \tag{2.42}$$
$$\mathsf{CleanLeft}(x) := \mathsf{ITE}(\mathsf{LastBit}(\mathsf{PLen}(x)), \mathsf{RT}(x), x) \tag{2.43}$$

then it can be verified that:

$$\mathsf{PV} \vdash \mathsf{CleanLeft}(\mathsf{PEnc}(y)) = \mathsf{PEnc}(y) \tag{2.44}$$
$$\mathsf{PV} \vdash \mathsf{CleanLeft}(1 \circ \mathsf{PEnc}(y)) = \mathsf{PEnc}(y) \tag{2.45}$$

(Recall that PV proves $\mathsf{PLen}(\mathsf{PEnc}(y)) = 0$ by Proposition 2.5.2, and $\mathsf{PLen}(1 \circ \mathsf{PEnc}(y)) = 1$ by essentially the same proof.)

Finally, we define:

$$\mathsf{Right}(\tau) := \mathsf{PDec}(\mathsf{CleanLeft}(\mathsf{RightRaw}(\tau))). \tag{2.46}$$

Here we hide tons of unfolding; I may try to write a Lean/Coq proof at some point as these unfolding can be done by calling “simp”, I think.

Proof of Correctness. We will sketch the proof that PV proves Equation (2.29):

Lemma 2.5.4. PV *proves the following equations:*

- $\mathsf{Left}(\mathsf{MakePair}(x, y)) = x$*;*
- $\mathsf{Right}(\mathsf{MakePair}(x, y)) = y$*.*

Proof Sketch. (*Correctness of* Right). We start by proving the second equation. We first need to prove that:

$$\mathsf{RightRaw}(\mathsf{MakePair}(x, y)) = \mathsf{ITE}_0^{(\varepsilon)}(\mathsf{FirstBit}(y), \mathsf{PEnc}(y), 1 \circ \mathsf{PEnc}(y)) \tag{2.47}$$

where $\mathsf{FirstBit}(y) := \mathsf{ITR}(y, \mathsf{TR}(y))$, $\mathsf{ITE}_0^{(\varepsilon)}(c, x, y)$ outputs y if $c = \varepsilon$ or $\mathsf{LastBit}(c) = 1$, and outputs x if $\mathsf{LastBit}(c) = 0$. This equation formally describes the fact that

$$\mathsf{RightRaw}(\mathsf{MakePair}(x, y))$$

is $1 \circ \mathsf{PEnc}(y)$ if $y = \varepsilon$ or the leftmost bit of y is 0, and is $\mathsf{PEnc}(y)$ otherwise. Equation (2.47) can be proved by induction on y that both sides of the equation are identical to the function recursively defined by $g(x) = 1$ and

$$h_0(x, y, z) = \mathsf{ITE}(\mathsf{IsEps}(y), 100, s_0(s_0(z))) \tag{2.48}$$
$$h_1(x, y, z) = \mathsf{ITE}(\mathsf{IsEps}(y), 10, s_0(s_1(z))) \tag{2.49}$$

for $i \in \{0, 1\}$. Note that

- To prove that the LHS of Equation (2.47) is equal to the function recursively defined by g, h_0, h_1, notice that PV proves

  $$\mathsf{RightRaw}(\mathsf{MakePair}(x, s_i(y))) = s_0(\mathsf{RightRaw}(s_i(\mathsf{MakePair}(x, y))))$$

  Here we hide tons of unfolding.

  by unfolding for $i \in \{0, 1\}$. Then we prove by a case study on y that

  $$\mathsf{PV} \vdash s_0(\mathsf{RightRaw}(s_0(\mathsf{MakePair}(x, y)))) = \mathsf{ITE}(\mathsf{IsEps}(y), 100, s_0(s_0(z)))$$
  $$\mathsf{PV} \vdash s_0(\mathsf{RightRaw}(s_1(\mathsf{MakePair}(x, y)))) = \mathsf{ITE}(\mathsf{IsEps}(y), 10, s_0(s_0(z)))$$

  by Theorem 2.4.5. Notice that the RHS of the equations above are exactly $h_0(x, y, z)$ and $h_1(x, y, z)$.

- To prove that the RHS of Equation (2.47) is equal to the function recursively defined by g, h_0, h_1, the hard part is to prove that PV proves

  $$\begin{aligned}&\mathsf{ITE}_0^{(\varepsilon)}(\mathsf{FirstBit}(s_i(y)), \mathsf{PEnc}(s_i(y)), 1 \circ \mathsf{PEnc}(s_i(y)))\\ =&h_i(x, y, \mathsf{ITE}_0^{(\varepsilon)}(\mathsf{FirstBit}(y), \mathsf{PEnc}(y), 1 \circ \mathsf{PEnc}(y)))\end{aligned}$$

  for $i \in \{0, 1\}$. We can prove this by case study on y using Theorem 2.4.5, the fact that $\mathsf{PV} \vdash \mathsf{FirstBit}(s_j(s_i(y))) = \mathsf{FirstBit}(s_i(y))$, and Theorem 2.4.7.

The details of the proofs above are omitted.

Now we prove that $\mathsf{Right}(\mathsf{MakePair}(x,y)) = y$. Notice that PV proves the following sequence of calculation:

$$\begin{aligned} & \mathsf{Right}(\mathsf{MakePair}(x,y)) \\ = {} & \mathsf{PDec}(\mathsf{CleanLeft}(\mathsf{RightRaw}(\mathsf{MakePair}(x,y)))) && \text{(Unfolding Right)} \\ = {} & \mathsf{PDec}(\mathsf{CleanLeft}(\mathsf{ITE}_0^{(\varepsilon)}(\mathsf{FirstBit}(y), \mathsf{PEnc}(y), 1 \circ \mathsf{PEnc}(y)))). && (2.50) \end{aligned}$$

The last equation follows from Equation (2.47).

To further simplify Equation (2.50), we will perform a case study on “dirty” ITE using Theorem 2.4.7. We need to prove that

Recall that I said Theorem 2.4.7 will be useful :)

$$\mathsf{PDec}(\mathsf{CleanLeft}(\mathsf{PEnc}(y))) = y \quad (2.51)$$

$$\mathsf{PDec}(\mathsf{CleanLeft}(1 \circ \mathsf{PEnc}(y))) = y \quad (2.52)$$

By Equation (2.44) and (2.45), both of them are implied by $\mathsf{PDec}(\mathsf{PEnc}(y)) = y$, which is given by Proposition 2.5.3.

(*Correctness of* Left). We will prove that $\mathsf{PV} \vdash \mathsf{Left}(\mathsf{MakePair}(x,y)) = x$. By the definition axiom of Left, it suffices to show that PV proves:

$$\mathsf{TR}(\mathsf{TR}(\mathsf{ITR}(\mathsf{MakePair}(x,y), \mathsf{PEnc}(\mathsf{Right}(\mathsf{MakePair}(x,y)))))) = x.$$

By the correctness of Right, we know that $\mathsf{Right}(\mathsf{MakePair}(x,y)) = y$ and thus we need to prove in PV that:

$$\mathsf{TR}(\mathsf{TR}(\mathsf{ITR}(x \circ 11 \circ \mathsf{PEnc}(y), \mathsf{PEnc}(y)))) = x.$$

Note that by induction on y, it can be proved that:

Proposition 2.5.5. $\mathsf{PV} \vdash \mathsf{ITR}(x \circ y, y) = x$.

Thus, it suffices to prove in PV that $\mathsf{TR}(\mathsf{TR}(x \circ 11)) = x$, which can be proved by unfolding TR's. □

2.5.2 Tuples

To support making and unwinding tuples, we need to implement the following functions for every $k \in \mathbb{N}$ and $i \in [k]$:

- $\mathsf{MakeTuple}^{(k)}(x_1, \dots, x_k)$ intended to construct a tuple $\pi = (x_1, \dots, x_k)$;
- $\mathsf{UnwindTuple}_i^{(k)}(\pi) = x_i$ takes the i-th element.

We use the standard construction: a $(k+1)$-tuple $(x_1, \dots, x_k)$ is defined as a pair of x_1 and $(x_2, \dots, x_k)$. Formally, we define by induction on k that for $k \geq 3$:

$$\mathsf{MakeTuple}^{(k)}(x_1, x_2, \dots, x_k) = \mathsf{MakePair}(x_1, \mathsf{MakeTuple}^{(k-1)}(x_2, \dots, x_k)) \quad (2.53)$$

$$\mathsf{UnwindTuple}_1^{(k)}(\pi) = \mathsf{Left}(\pi) \quad (2.54)$$

$$\mathsf{UnwindTuple}_i^{(k)}(\pi) = \mathsf{UnwindTuple}^{(k-1)}(\mathsf{Right}(\pi)) \quad (2 \leq i \leq k) \quad (2.55)$$

and for $k = 2$, $\mathsf{MakeTuple}^{(2)} = \mathsf{MakePair}$, $\mathsf{UnwindTuple}_1^{(k)} = \mathsf{Left}$, $\mathsf{UnwindTuple}_2^{(k)} = \mathsf{Right}$. We can prove that:

Lemma 2.5.6. *For every $k \geq 2$ and $i \in [k]$, PV proves that:*

$$\mathsf{UnwindTuple}_i^{(k)}(\mathsf{MakeTuple}^{(k)}(x_1, \dots, x_k)) = x_i.$$

Proof Sketch. We prove by induction on k and applying Lemma 2.5.4. Notice that this induction is in meta-theory instead of PV. □

For simplicity, we will ignore the superscript (k) if there is no ambiguity. We will simply denote the tuple $\mathsf{MakeTuple}(x_1, \dots, x_k)$ as $\pi = (x_1, \dots, x_k)$, and the function $\mathsf{UnwindTuple}_i(\pi)$ as π_i.

2.6 Extensions on Recursion and Induction

In this section, we will propose a few extensions on the (limited) recursion rule for introducing PV functions, as well as the (structural) induction rule on the extensions of recursion.

2.6.1 Recursion on Multiple Variables

We first consider a version of recursion on multiple variables. As a motivating example, we will consider how to define a function $\mathsf{EQ}(x, y)$ that outputs 1 if and only if $x = y$, and outputs 0 otherwise. Intuitively we will define the function by considering the last bit of x and y:

$$\mathsf{EQ}(\varepsilon, \varepsilon) := 1 \tag{2.56}$$
$$\mathsf{EQ}(\varepsilon, s_i(y)) := 0 \quad (i \in \{0,1\}) \tag{2.57}$$
$$\mathsf{EQ}(s_i(x), \varepsilon) := 0 \quad (i \in \{0,1\}) \tag{2.58}$$
$$\mathsf{EQ}(s_i(x), s_j(y)) := \mathsf{EQ}(x, y) \quad (i, j \in \{0,1\}, i = j) \tag{2.59}$$
$$\mathsf{EQ}(s_i(x), s_j(y)) := 0 \quad (i, j \in \{0,1\}, i \neq j) \tag{2.60}$$

Recursion on multiple variables. Generalizing this example, we would like to have the following meta-theorem:

Theorem 2.6.1 ([Coo75]). *Let $g_{00}(\vec{x}), g_{01}(\vec{x}, y), g_{10}(\vec{x}, y)$ be PV functions, $h_\alpha(\vec{x}, y_1, y_2, z)$ and $k_\alpha(\vec{x}, y_1, y_2)$ for $\alpha \in \{0,1\}^2$ be PV functions. If*

$$\mathsf{PV} \vdash \mathsf{ITR}(h_\alpha(\vec{x}, y_1, y_2, z), z \circ k_\alpha(\vec{x}, y_1, y_2)) = 0$$

for every $\alpha \in \{0,1\}^2$, then there is a PV function $f(\vec{x}, y_1, y_2)$ such that the following equations are provable in PV:

- $f(\vec{x}, \varepsilon, \varepsilon) = g_{00}(\vec{x})$*;*
- $f(\vec{x}, \varepsilon, s_i(y)) = g_{01}(\vec{x}, s_i(y))$ *for* $i \in \{0,1\}$*;*[7]
- $f(\vec{x}, s_i(y), \varepsilon) = g_{10}(\vec{x}, s_i(y))$ *for* $i \in \{0,1\}$*;*

[7] Here we write $s_i(y)$ instead of y because otherwise we will need to ensure that $g_{01}(\vec{x}, \varepsilon) = g_{10}(\vec{x}, \varepsilon)$.

- $f(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2)) = h_{i_1 i_2}(\vec{x}, y_1, y_2, f(\vec{x}, y_1, y_2))$ *for* $i_1, i_2 \in \{0, 1\}$.

Similarly, we can propose a further generalization of the theorem where we perform simultaneous recursion on more than two variables, which can be proved similar to this meta-theorem.

To prove Theorem 2.6.1, we need to implement recursion on two variables using the limited recursion rule in PV that only allows recursion on a single variable. The idea is to introduce two auxiliary variables w_1, w_2 and define a function $f'(\vec{x}, y_1, y_2, w_1, w_2)$ that is supposed to satisfy that

$$f'(\vec{x}, y_1, y_2, w_1, w_2) = f(\vec{x}, \mathsf{ITR}(y_1, \mathsf{ITR}(w_1, w_2)), \mathsf{ITR}(y_2, \mathsf{ITR}(w_1, w_2)))$$

and define f' by recursion on the variable w_2. Finally, we will define

$$f(\vec{x}, y_1, y_2) := f'(\vec{x}, y_1, y_2, y_1 \circ y_2, y_1 \circ y_2)$$

and verify that all equations in Theorem 2.6.1 can be proved in PV.

Proof Sketch of Theorem 2.6.1. We will define a function $f'(\vec{x}, y_1, y_2, w_1, w_2)$ as discussed above. That is, we will use the limited recursion rule on the auxiliary variable w_2 using the functions g, h_0, h_1, where $g(\vec{x}, y_1, y_2, w_1)$

$$= \begin{cases} \varepsilon & \mathsf{Not}(\mathsf{Or}(\mathsf{IsEps}(\mathsf{ITR}(y_1, w_1)), \mathsf{IsEps}(\mathsf{ITR}(y_2, w_1)))) \\ g_{00}(\vec{x}) & \mathsf{And}(\mathsf{IsEps}(\mathsf{ITR}(y_1, w_1)), \mathsf{IsEps}(\mathsf{ITR}(y_2, w_1))) \\ g_{01}(\vec{x}, \mathsf{ITR}(y_2, w_1)) & \mathsf{And}(\mathsf{IsEps}(\mathsf{ITR}(y_1, w_1)), \mathsf{Not}(\mathsf{IsEps}(\mathsf{ITR}(y_2, w_1)))) \\ g_{10}(\vec{x}, \mathsf{ITR}(y_1, w_1)) & \mathsf{And}(\mathsf{Not}(\mathsf{IsEps}(\mathsf{ITR}(y_1, w_1))), \mathsf{IsEps}(\mathsf{ITR}(y_2, w_1))) \end{cases} \tag{2.61}$$

and for $i \in \{0, 1\}$, $h_i(\vec{x}, y_1, y_2, w_1, w_2, z)$

$$= \begin{cases} g(\vec{x}, \hat{y}_1, \hat{y}_2, \varepsilon) & \mathsf{Or}(\mathsf{IsEps}(\hat{y}_1), \mathsf{IsEps}(\hat{y}_2)) \\ h_{i_1 i_2}(\vec{x}, \mathsf{TR}(\hat{y}_1), \mathsf{TR}(\hat{y}_2), z) & (\mathsf{LastBit}(\hat{y}_1), \mathsf{LastBit}(\hat{y}_2)) = (i_1, i_2) \end{cases} \tag{2.62}$$

where $\hat{y}_j := \mathsf{ITR}(y_j, \mathsf{ITR}(w_1, s_i(w_2)))$ for $j \in \{1, 2\}$. Note that the case study in defining g, h_0, and h_1 can be done by the if-then-else function. We will define the function $k_i(\vec{x}, y_1, y_2, w_1, w_2)$ as

$$= \begin{cases} g(\vec{x}, \hat{y}_1, \hat{y}_2, \varepsilon) & \mathsf{Or}(\mathsf{IsEps}(\hat{y}_1), \mathsf{IsEps}(\hat{y}_2)) \\ k_{i_1 i_2}(\vec{x}, \mathsf{TR}(\hat{y}_1), \mathsf{TR}(\hat{y}_2)) & (\mathsf{LastBit}(\hat{y}_1), \mathsf{LastBit}(\hat{y}_2)) = (i_1, i_2) \end{cases} \tag{2.63}$$

Inequality for limited recursion. We need to verify that (g, h_0, h_1, k_0, k_1) satisfy that

$$\mathsf{PV} \vdash \mathsf{ITR}(h_i(\vec{x}, y_1, y_2, w_1, w_2, z), k_i(\vec{x}, y_1, y_2, w_1, w_2)) = 0$$

for $i \in \{0, 1\}$ to use the limited recursion rule. This is implied by the assumption, the fact that $\mathsf{PV} \vdash \mathsf{ITR}(x, x) = 0$, and the case study technique on ITE (see Theorem 2.4.6 and Remark 2.4.2).

Correctness. As mentioned above, we will define

$$f(\vec{x}, y_1, y_2) := f'(\vec{x}, y_1, y_2, y_1 \circ y_2, y_1 \circ y_2),$$

and it remains to verify that the equations in the theorem statement are provable in PV. We sketch the proof of each equation:

- To see that $f(\vec{x}, \varepsilon, \varepsilon) = g_{00}(\vec{x})$, notice that $f(\vec{x}, \varepsilon, \varepsilon)$ is equal to $g(\vec{x}, \varepsilon, \varepsilon, \varepsilon)$. By the definition of g, we will obtain (after unfolding $\mathsf{ITR}, \mathsf{IsEps}$ and And) that $g(\vec{x}, \varepsilon, \varepsilon, \varepsilon) = g_{00}(\vec{x})$.
- To see that $f(\vec{x}, \varepsilon, s_j(y)) = g_{01}(\vec{x}, s_j(y))$, notice that $f(\vec{x}, \varepsilon, s_j(y))$ is equal to

  $$f'(\vec{x}, \varepsilon, s_j(y), \varepsilon \circ s_j(y), \varepsilon \circ s_j(y)) = f'(\vec{x}, \varepsilon, s_j(y), s_j(y), s_j(y)).$$

  Since $\hat{y}_1 = \mathsf{ITR}(\varepsilon, s_j(y), s_j(y)) = \varepsilon$, we know (by unfolding ITE in the definition of h_j) that this is PV-provably equal to

  $$g(\vec{x}, \varepsilon, \mathsf{ITR}(s_j(y), \mathsf{ITR}(s_j(y), s_j(y))), \varepsilon) = g(\vec{x}, \varepsilon, s_j(y), \varepsilon)$$

  (The equation follows from that $\mathsf{ITR}(x, x) = \varepsilon$ and $\mathsf{ITR}(y, \varepsilon) = y$ are provable in PV.) By unfolding g and ITE's (in the definition of g), we can prove in PV that $g(\vec{x}, \varepsilon, s_j(y), \varepsilon) = g_{01}(\vec{x}, s_j(y))$.
- The case for $f(\vec{x}, s_j(y), \varepsilon) = g_{10}(\vec{x}, s_j(y))$ is similar to the case above.
- To see that $f(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2)) = h_{i_1 i_2}(\vec{x}, y_1, y_2, f(\vec{x}, y_1, y_2))$ for $i_1, i_2 \in \{0, 1\}$, notice that PV proves

  $$\begin{aligned} & f(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2)) && (2.64)\\ =\ & f'(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2), s_{i_1}(y_1) \circ s_{i_2}(y_2), s_{i_1}(y_1) \circ s_{i_2}(y_2)) && (2.65)\\ =\ & f'(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2), s_{i_2}(s_{i_1}(y_1) \circ y_2), s_{i_2}(s_{i_1}(y_1) \circ y_2)) && (2.66)\\ =\ & h_{i_2}(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2), s_{i_2}(s_{i_1}(y_1) \circ y_2), s_{i_1}(y_1) \circ y_2, z) && (2.67) \end{aligned}$$

  where

  $$\begin{aligned} z &:= f'(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2), s_{i_2}(s_{i_1}(y_1) \circ y_2), s_{i_1}(y_1) \circ y_2) && (2.68)\\ &= f'(\vec{x}, y_1, y_2, y_1 \circ y_2, y_1 \circ y_2) && (2.69)\\ &= f(\vec{x}, y_1, y_2). && (2.70) \end{aligned}$$

  Here, Equation (2.69) is non-trivial and will be deferred to the end of the proof. Also, notice that

  $$\begin{aligned} \hat{y}_1 &= \mathsf{ITR}(s_{i_1}(y_1), \mathsf{ITR}(s_{i_2}(s_{i_1}(y_1) \circ y_2), s_{i_2}(s_{i_1}(y_1) \circ y_2))) = s_{i_1}(y_1) && (2.71)\\ \hat{y}_2 &= \mathsf{ITR}(s_{i_2}(y_2), \mathsf{ITR}(s_{i_2}(s_{i_1}(y_1) \circ y_2), s_{i_2}(s_{i_1}(y_1) \circ y_2))) = s_{i_2}(y_2) && (2.72) \end{aligned}$$

  By unfolding h_{i_2} we can see that

  $$(2.67) = h_{i_1 i_2}(\vec{x}, y_1, y_2, z), \tag{2.73}$$

  which (together with Equation (2.70)) obtains the equation we need.

It remains to prove Equation (2.69). Indeed, we will prove a more general equation (which derives Equation (2.69) by z_1/y_1 and z_2/y_2).

Proposition 2.6.2. PV *proves the following equation*

$$f'(\vec{x}, s_{i_1}(z_1), s_{i_2}(z_2), s_{i_2}(s_{i_1}(y_1) \circ y_2), s_{i_1}(y_1) \circ y_2) = f'(\vec{x}, z_1, z_2, y_1 \circ y_2, y_1 \circ y_2).$$

Proof Sketch. Let $f_1(\vec{x}, z_1, z_2, y_1, y_2)$ and $f_2(\vec{x}, z_1, z_2, y_1, y_2)$ be the LHS and RHS of the equation, respectively. Notice that when y_1 and y_2 are substituted by ε, the equation is provable in PV by simply unfolding f'. We then prove that

$$f_1(\vec{x}, z_1, z_2, y_1, \varepsilon) = f_2(\vec{x}, z_1, z_2, y_1, \varepsilon). \tag{2.74}$$

To see this, notice that both sides of the equation are identical to the function recursively defined on y_1 using some PV functions g', h'_0, h'_1. Specifically, the base case is given by that $f_1(\vec{x}, z_1, z_2, \varepsilon, \varepsilon) = f_2(\vec{x}, z_1, z_2, \varepsilon, \varepsilon)$, and the induction step requires proving identities on ITR, such as $\mathsf{ITR}(z, \mathsf{ITR}(s_{i_2}(s_{i_1}(y_1) \circ y_2), s_{i_1}(y_1) \circ y_2)) = \mathsf{TR}(z)$.

Hopefully you are convinced that all these identities are PV provable...

Subsequently, we prove

$$f_1(\vec{x}, z_1, z_2, y_1, y_2) = f_2(\vec{x}, z_1, z_2, y_1, y_2)$$

by showing that both sides of the equation are identical to the function recursively defined on y_2 using some PV functions g'', h''_0, h''_1. This time, the base case is given by Equation (2.74), while the induction case also requires proving identities on ITR. □

This completes the proof. □

2.6.2 Induction on Multiple Variables

After defining the function EQ, we need to prove properties about it to use it in mathematical reasoning. For instance, we may need to prove that

$$\mathsf{PV} \vdash \mathsf{EQ}(x, y) = \mathsf{EQ}(y, x). \tag{2.75}$$

Since EQ is defined using Theorem 2.6.1 instead of the standard recursion rule in PV, we will need a structural induction rule for functions defined by recursion on multiple variables. Formally, we will need to prove:

Theorem 2.6.3. *Let $f_1(\vec{x}, y_1, y_2), f_2(\vec{x}, y_1, y_2)$ be two PV functions. Let $g_{00}(\vec{x})$, $g_{01}(\vec{x}, y)$, $g_{10}(\vec{x}, y)$, and $h_\alpha(\vec{x}, y_1, y_2, z)$, $\alpha \in \{0,1\}^2$ be PV functions. Suppose that for $j \in \{1, 2\}$, PV proves that*

- $f_j(\vec{x}, \varepsilon, \varepsilon) = g_{00}(\vec{x})$;
- $f_j(\vec{x}, \varepsilon, s_j(y)) = g_{01}(\vec{x}, s_j(y))$ *for* $j \in \{0, 1\}$;
- $f_j(\vec{x}, s_j(y), \varepsilon) = g_{10}(\vec{x}, s_j(y))$ *for* $j \in \{0, 1\}$;
- $f_j(\vec{x}, s_{i_1}(y_1), s_{i_2}(y_2)) = h_{i_1 i_2}(\vec{x}, y_1, y_2, f_j(\vec{x}, y_1, y_2))$ *for* $i_1, i_2 \in \{0, 1\}$.

Then $\mathsf{PV} \vdash f_1(\vec{x}, y_1, y_2) = f_2(\vec{x}, y_1, y_2)$.

See Equation (2.56) to (2.60)

To see that the theorem suffices to prove Equation (2.75), notice that for $f_1(x,y) := \mathsf{EQ}(x,y)$ and $f_2(x,y) = \mathsf{EQ}(y,x)$, both f_1 and f_2 PV-provably satisfy the precondition in Theorem 2.6.3 with $g_{00} = 1$, $g_{01} = 0$, $g_{10} = 0$, and

$$h_{ij}(x,y,z) = \begin{cases} z & i = j \\ 0 & i \neq j \end{cases},$$

therefore by Theorem 2.6.3 we know that $\mathsf{PV} \vdash f_1(x,y) = f_2(x,y)$.

Proof Sketch of Theorem 2.6.3. The proof idea of Theorem 2.6.3 closely follows the proof of Theorem 2.6.1. For each $j \in \{1,2\}$, we define

$$f_j'(\vec{x}, y_1, y_2, w_1, w_2) := f_j(\vec{x}, \mathsf{ITR}(y_1, \mathsf{ITR}(w_1, w_2)), \mathsf{ITR}(y_2, \mathsf{ITR}(w_1, w_2))). \tag{2.76}$$

We will prove that

$$\mathsf{PV} \vdash f_0'(\vec{x}, y_1, y_2, y_1 \circ y_2, w_2) = f_1'(\vec{x}, y_1, y_2, y_1 \circ y_2, w_2) \tag{2.77}$$

and thus the theorem follows by substituting w_2 with $y_1 \circ y_2$.

To prove Equation (2.77), we will use the induction rule in PV on the variable w_2. Concretely, we define $g'(\vec{x}, y_1, y_2) = g_{00}(\vec{x})$ and for $i \in \{0,1\}$, $h_i'(\vec{x}, y_1, y_2, w_2, z)$

$$= \begin{cases} g_{00}(\vec{x}) & \mathsf{And}(\mathsf{IsEps}(\mathsf{ITR}(y_1, \mathsf{ITR}(w_1, s_i(w_2)))), \\ & \quad \mathsf{IsEps}(\mathsf{ITR}(y_2, \mathsf{ITR}(w_1, s_i(w_2))))) = 1 \\ g_{01}(\vec{x}, \mathsf{ITR}(y_2, \mathsf{ITR}(w_1, s_i(w_2)))) & \mathsf{IsEps}(\mathsf{ITR}(y_1, \mathsf{ITR}(w_1, s_i(w_2)))) = 1 \\ g_{10}(\vec{x}, \mathsf{ITR}(y_1, \mathsf{ITR}(w_1, s_i(w_2)))) & \mathsf{IsEps}(\mathsf{ITR}(y_2, \mathsf{ITR}(w_1, s_i(w_2)))) = 1 \\ h_{i_1 i_2}(\vec{x}, \mathsf{TR}(\hat{y}_1), \mathsf{TR}(\hat{y}_2), z) & (\mathsf{LastBit}(\hat{y}_1), \mathsf{LastBit}(\hat{y}_2)) = (i_1, i_2) \end{cases}$$

where $w_1 := y_1 \circ y_2$, $\hat{y}_j := \mathsf{ITR}(y_j, w_1, s_i(w_2))$ for $j \in \{1,2\}$. It can be verified (similar to the proof of Theorem 2.6.3) that both LHS and RHS of Equation (2.77) are identical to the function recursively defined by $g'(\vec{x}, y_1, y_2), h_0(\vec{x}, y_1, y_2, w_2, z), h_1(\vec{x}, y_1, y_2, w_2, z)$ inductively on the variable w_2. □

Remark 2.6.1. Although we only consider the case for two variables, the same proof idea generalizes to induction on k variables for all $k \in \mathbb{N}$.

2.6.3 Application: Basic Arithmetic

Cheers! Feasible mathematicians are finally starting to study primary school mathematics.

Another important application of recursion and induction on multiple variables (i.e. Theorem 2.6.1 and 2.6.3) is to implement basic arithmetic operations of natural numbers such as addition and multiplication.

Encoding of natural numbers. Recall that natural numbers are not "first-class citizens" in the theory of PV — only binary strings have native support in PV. We will need to specify an encoding of natural numbers with binary strings.

Following [Coo75], we will use *dyadic encoding* where the leftmost bit is the most significant. Formally, let $[x]_{\mathbb{N}}$ be the number encoded by the string x, we define

$$[\varepsilon]_{\mathbb{N}} := 0,\ [s_0(x)]_{\mathbb{N}} := 2x+2,\ [s_1(x)]_{\mathbb{N}} := 2x+1.$$

This encoding method ensures a bijection between natural numbers and binary strings so that we will no longer need to verify whether a string encodes a valid number. For simplicity, we will identify $s_2(x)$ with $s_0(x)$; we also denote $s_2(x)$ by $x2$ and $s_1(x)$ by $x1$.

A downside is that ε means both "error" and 0 :(

Addition. To define addition $\mathsf{Add}(x,y)$ (i.e. $[\mathsf{Add}(x,y)]_{\mathbb{N}} = [x]_{\mathbb{N}} + [y]_{\mathbb{N}}$), we perform the standard algorithm on the dyadic encoding using Theorem 2.6.1:

$$\mathsf{Add}(x,\varepsilon) = \mathsf{Add}(\varepsilon,x) := x \tag{2.78}$$
$$\mathsf{Add}(x2,y2) := s_2(\mathsf{Succ}(\mathsf{Add}(x,y))) \tag{2.79}$$
$$\mathsf{Add}(x2,y1) = \mathsf{Add}(x1,y2) := s_1(\mathsf{Succ}(\mathsf{Add}(x,y))) \tag{2.80}$$
$$\mathsf{Add}(x1,y1) := s_2(\mathsf{Add}(x,y)) \tag{2.81}$$

where $\mathsf{Succ}(x)$ is the successor function $x \mapsto x+1$, whose definition is left as an exercise. Specifically, Add is defined from $g_{00} = \varepsilon, g_{01}(y) = g_{10}(y) = y$, $h_{00}(x,y,z) = s_2(\mathsf{Succ}(z))$, $h_{01}(x,y,z) = h_{10}(x,y,z) = s_1(\mathsf{Succ}(z))$, and $h_{11}(x,y,z) = s_2(z)$.

Note that to apply Theorem 2.6.1, we will also need to define functions k_α for $\alpha \in \{0,1\}$ and proves in PV that

$$\mathsf{ITR}(h_\alpha(x,y,z), z \circ k_\alpha(x,y)) = 0.$$

Indeed, it is not hard to verify that it suffices to define $k_{00}(x,y) = k_{01}(x,y) = k_{10}(x,y) = k_{11}(x,y) = 11$. (To see this, we will need that $\mathsf{PV} \vdash \mathsf{ITR}(\mathsf{Succ}(x), s_0(x)) = 0$, which is easy if Succ is defined properly.)

Proposition 2.6.4. *The following equations are provable in* PV*.*

- $\mathsf{Add}(x,y) = \mathsf{Add}(y,x)$
- $\mathsf{Add}(x,1) = \mathsf{Succ}(x)$
- $\mathsf{Add}(\mathsf{Add}(x,y),z) = \mathsf{Add}(x,\mathsf{Add}(y,z))$

The first equation can be proved similar to the proof of $\mathsf{EQ}(x,y) = \mathsf{EQ}(y,x)$. The second can be proved by unfolding Add using the definition axioms.

To prove the last equation, we will need to apply induction on three variables (see Theorem 2.6.3 and Remark 2.6.1). Specifically, let $g_{000} = 0$, $g_{001}(x,y) = g_{010}(x,y) = g_{100}(x,y) = \mathsf{Add}(x,y)$, $g_{011}(x) = g_{101}(x) = g_{110}(x) = x$, and some $h_\alpha(x,y,z,w)$ for $\alpha \in \{0,1\}^3$, both LHS and RHS of the equation are PV-provably equal to the function recursively defined from these g_α and h_α. For instance, for $\alpha = 111$, $h_{111}(x, y, z, w) := s_1(\mathsf{Succ}(w))$ and we can prove in PV that

$$\begin{aligned}\mathsf{Add}(\mathsf{Add}(x1, y1), z1) &= h_{111}(x, y, z, \mathsf{Add}(\mathsf{Add}(x, y), z)) \\ \mathsf{Add}(x1, \mathsf{Add}(y1, z1)) &= h_{111}(x, y, z, \mathsf{Add}(x, \mathsf{Add}(y, z))\end{aligned}$$

by unfolding Add and h_{111}. Note that other basic properties of addition can also be proved following a similar approach.

From now on, we will slightly abuse the notation to write $\mathsf{Add}(x, y)$ as $x + y$.

Multiplication. Similarly, we can define multiplication $\mathsf{Mul}(x, y)$ of two numbers. Let $2 \cdot x := \mathsf{Add}(x, x)$ and $4 \cdot x := \mathsf{Add}(2 \cdot x, 2 \cdot x)$. We define:

$$\mathsf{Mul}(x, \varepsilon) = \mathsf{Mul}(\varepsilon, x) = \varepsilon \tag{2.82}$$

$$\mathsf{Mul}(x1, y1) := 4 \cdot \mathsf{Mul}(x, y) + 2 \cdot x + 2 \cdot y + 1 \tag{2.83}$$

$$\mathsf{Mul}(x2, y1) := 4 \cdot \mathsf{Mul}(x, y) + 2 \cdot x + 4 \cdot y + 2 \tag{2.84}$$

$$\mathsf{Mul}(x1, y2) := 4 \cdot \mathsf{Mul}(x, y) + 4 \cdot x + 2 \cdot y + 2 \tag{2.85}$$

$$\mathsf{Mul}(x2, y2) := 4 \cdot \mathsf{Mul}(x, y) + 4 \cdot x + 4 \cdot y + 4 \tag{2.86}$$

More formally, Mul is recursively defined using Theorem 2.6.1 from $g_{00} = g_{01}(x) = g_{10}(x) = \varepsilon$, $h_{11}(x, y, z) = 4 \cdot z + 2 \cdot x + 2 \cdot y + 1$, $h_{01}(x, y, z) = 4 \cdot z + 2 \cdot x + 4 \cdot y + 2$, $h_{10}(x, y, z) = 4 \cdot z + 4 \cdot x + 2 \cdot y + 2$, and $h_{00}(x, y, z) = 4 \cdot z + 4 \cdot x + 4 \cdot y + 4$. We will need to ensure that for every $\alpha \in \{0, 1\}^2$, there is a function k_α such that

$$\mathsf{PV} \vdash \mathsf{ITR}(h_\alpha(x, y, z), z \circ k_\alpha(x, y)) = \varepsilon.$$

Indeed, we can define $k_\alpha(x, y) := 1111 \circ (x \circ 1111) \circ (y \circ 1111)$. To see that this suffices, we will need to prove that

Proposition 2.6.5. PV *proves the following equations:*

- $\mathsf{ITR}(\mathsf{Add}(x, y), x \circ y) = 0$
- $\mathsf{ITR}(2 \cdot x, x \circ 1111) = 0$
- $\mathsf{ITR}(4 \cdot x, x \circ 1111) = 0$

The first equation can be proved by induction on x and y using Theorem 2.6.3. The second and third equations can be proved by induction on x using the induction rule of PV. From now on, we will slightly abuse the notation to denote $\mathsf{Mul}(x, y)$ as $x \cdot y$ or xy.

Similar to the case for addition, basic properties of multiplication can be proved by induction on one or multiple variables. For instance:

Proposition 2.6.6. PV *proves the following equations:*

- $xy = yx$
- $(xy)z = x(yz)$
- $x(y + z) = xy + xz$

We sketch the proof of the third equation for completeness. We will prove the equation by induction on x, y, z (see Theorem 2.6.3 and Remark 2.6.1). The cases that at least one of x, y, z is ε are straightforward by unfolding Mul and Add. It suffices to show that there are $h_\alpha(x, y, z, w)$ for $\alpha \in \{0,1\}^3$ such that both sides of the equation satisfy the recursive relation specified by h_α. For instance, letting $\alpha = 111$, we will have

$$\begin{aligned} x1 \cdot (y1 + z1) &= x1 \cdot ((y+z)2) \\ &= 4 \cdot x(y+z) + 4 \cdot x + 2 \cdot (y+z) + 2; \\ x1 \cdot y1 + x1 \cdot z1 &= 4 \cdot xy + 2 \cdot x + 2 \cdot y + 1 + 4 \cdot xz + 2 \cdot x + 2 \cdot z + 1 \\ &= 4 \cdot (xy + xz) + 4 \cdot x + 2 \cdot (y+z) + 2; \end{aligned}$$

where the last equation follows from the commutativity and associativity of addition. Therefore, it suffices to define $h_{111}(x, y, z, w) = 4 \cdot z + 2 \cdot x + 2 \cdot (y+z) + 2$.

2.7 The Function EQ and Equality

An important application of the induction principle on multiple variables is to prove that the function EQ we defined before is indeed the characteristic function of the equality relation in PV. We will use the notion of conditional equation introduced in Section 2.4.4.

Lemma 2.7.1. $\mathsf{PV} \vdash \mathsf{EQ}(y_1, y_2) \Rightarrow y_1 = y_2$.

Proof. Unfolding the definition of conditional equations, we need to prove that $\mathsf{PV} \vdash \mathsf{ITE}(\mathsf{EQ}(y_1, y_2), y_2, y_1) = y_1$. We prove this by applying Theorem 2.6.3 on the variables y_1 and y_2. Let $f_1(x, y) = \mathsf{ITE}(\mathsf{EQ}(x, y), y, x)$ and $f_2(x, y) = x$. We define $g_{00} = 1$, $g_{10}(y) = y$, $g_{01}(y) = \varepsilon$, $h_{01}(y_1, y_2, z) = s_0(y_1)$, $h_{10}(y_1, y_2) = s_1(y_1)$, and $h_{ii}(y_1, y_2, z) = s_i(z)$ for $i \in \{0, 1\}$. Then both f_1 and f_2 are identical to the function recursively defined from $g_{i_1 i_2}$ and $h_{i_1 i_2}$.

We verify this for the LHS of the equation. The base cases (i.e. at least one of y_j is ε) are straightforward, so we will only consider the recursion case. Fix any $i_1, i_2 \in \{0, 1\}$, we need to prove that

$$\begin{aligned} &\mathsf{ITE}(\mathsf{EQ}(s_{i_1}(y_1), s_{i_2}(y_2)), s_{i_2}(y_2), s_{i_1}(y_1)) \\ &= h_{i_1 i_2}(y_1, y_2, \mathsf{ITE}(\mathsf{EQ}(y_1, y_2), y_2, y_1)). \end{aligned} \tag{2.87}$$

We prove this by a case study on whether $i_1 = i_2$. (Note that the case study happens in meta-theory.)

Suppose that $i_1 = i_2$, then by the definition equations of EQ we know that $\mathsf{EQ}(s_{i_1}(y_1), s_{i_2}(y_2)) = \mathsf{EQ}(y_1, y_2)$. Therefore, the LHS of Equation (2.87) is PV-provably equal to

$$\mathsf{ITE}(\mathsf{EQ}(y_1, y_2), s_{i_2}(y_2), s_{i_1}(y_1))$$

which is further PV-provably equal to

$$s_{i_1}(\mathsf{ITE}(\mathsf{EQ}(y_1, y_2)), y_2, y_1) = h_{i_1 i_2}(y_1, y_2, \mathsf{ITE}(\mathsf{EQ}(y_1, y_2), y_2, y_1)).$$

(Here, we use the property that $i_1 = i_2$.)

Suppose that $i_1 \neq i_2$, then by the definition equations of EQ we know that

$$\mathsf{EQ}(s_{i_1}(y_1), s_{i_2}(y_2)) = 0,$$

and hence by unfolding ITE we know that the LHS of Equation (2.87) is PV-provably equal to $s_{i_1}(y_1) = h_{i_1 i_2}(y_1, y_2, z)$ for any z. □

Theorem 2.7.2. *$\mathsf{PV} \vdash \mathsf{EQ}(s(\vec{x}), t(\vec{x})) = 1$ if and only if $\mathsf{PV} \vdash s(\vec{x}) = t(\vec{x})$.*

Proof. The ($\Leftarrow$) side is straightforward and we will only prove the ($\Rightarrow$) side. Suppose that $\mathsf{PV} \vdash \mathsf{EQ}(s(\vec{x}), t(\vec{x})) = 1$. By Lemma 2.7.1 and (L44) substitution we have

$$\mathsf{PV} \vdash \mathsf{EQ}(s(\vec{x}), t(\vec{x})) \Rightarrow s(\vec{x}) = t(\vec{x}).$$

Then $\mathsf{PV} \vdash s(\vec{x}) = t(\vec{x})$ by Modus Ponens (see Proposition 2.4.10). □

Remark 2.7.1. One remark to this theorem is that the ($\Rightarrow$) direction is proved in a *black-box* manner: The transformation from the proof of $\mathsf{EQ}(s(\vec{x}), t(\vec{x})) = 1$ to the proof of $s(\vec{x}) = t(\vec{x})$ does not read the proof; the only information needed is the proof is correct and the last line is $\mathsf{EQ}(s(\vec{x}), t(\vec{x})) = 1$.

2.8 Bibliographical and Other Remarks

The theory PV. Cook's theory PV [Coo75] can be viewed as a time-bounded version of Skolem's primitive recursive arithmetic PRA [Sko23], where the time-bound is achieved with the idea from Cobham's characterization [Cob65]. Both theories are equational and "logic-free", i.e., do not contain logical connectives or quantifiers, and therefore their consistencies are arguably more trustworthy than that of those with connectives and quantifiers. One notable application is that extensions of Skolem's theory PRA were used as the base theory for Gentzen's celebrated consistency proof of Peano Arithmetic [Gen36].

Cook [Coo75] also introduced an extension of PV with logical connectives (but no quantifiers), which will be explained in the next chapter. Cook called the theory PV_1, though PV_1 usually refers to the first-order extension of PV that has both connectives and quantifiers in bounded arithmetic literature (see, e.g., [Kra95a, KPT91]).

Buss's theories. Independently, Buss [Bus86] introduced a first-order theory S^1_2 by extending Parikh's theory $I\Delta_0$ [Par71]. The theory allows a stronger induction principle beyond Postulate 3 and is thus not known to be admissible in PV; indeed, S^1_2 is strictly stronger than PV unless the polynomial-time hierarchy collapses [Bus95b]. Interestingly, Buss's witnessing theorem [Bus86] (see also [Kra95a]) shows that $S^1_2(\mathsf{PV})$ (i.e. S^1_2 equipped with PV functions) is an conservative extension over PV — in terms of provable *equations*, S^1_2 is as strong as PV.

It is worth noting that a slight modification of T^0_2, the lowest level of Buss's T_2 hierarchy, is equivalent to Cook's theory PV [Jeř06].

Two-sorted theories. An alternative approach to define bounded theories is to consider "two-sorted" first-order logic, where the variables are either strings X or indices i (see, e.g., [CN10, CKKO21]). The distinction between strings and indices is necessary to develop theories corresponding to very weak complexity classes, say AC^0. The two-sorted version of PV is called VPV, and is known to be equivalent to PV in a strong sense (see [Kra19, Section 9]).

Feasible Mathematics Thesis. We refer interested readers to the introduction section of [Coo75] for discussions of the Verifiability Thesis. The Thesis of Feasible Mathematics is arguably weaker than Cook's original Verifiability Thesis as any theory satisfying the three postulates should immediately give the polynomial-time verifiability of the theory — our postulates ensure that the proof itself is "a uniform mean of verification" as considered by Cook [Coo75].

For instance, the postulate of induction induces a uniform verification method following "the chain of reasoning", which is of polynomial length.

In particular, It could be the case that the Thesis of Feasible Mathematics is correct but the Verifiability Thesis is incorrect if there is a new widely accepted "uniform mean of verification" (e.g., a new interpretation of "verification" from quantum computing) that is independent of the three postulates (and therefore also cannot be implemented in PV).

Programming in PV. Cook's 1975 paper [Coo75] does not prove the admissibility of basic programming functionalities such as pairing, tuples, or recursion on multiple variables in PV. A detailed exposition was given later by Cook and Urquhart [CU93]; the idea of implementing conditional equations with ITE (see Section 2.4.4) plays an important role in their construction.

Chapter 3

Basic Proof Theory of PV

In the previous chapter, we have already proved several meta-theorems that serve as an abstraction of particular proof tactics. However, the meta-theorems are still far from informal mathematical reasoning — this leaves a significant gap between informal feasible mathematics, where standard logical rules are allowed, and the "logic-free" theory PV.

In this chapter, we will develop more tools for *reasoning* in PV. We will define a natural proof system that is close to human reasoning, and in particular, is close to the informal notion of feasible mathematics. This will be an important step towards constructing more advanced algorithms and data structures as the results developed in this section will relieve us from writing long and tedious PV proofs.

3.1 A Predicate Logic and its Proof System

We define a proof system PV-PL that, intuitively, extends PV by allowing *logical connectives* $\{\to, \land, \lor, \neg\}$ and *conditional proofs*.

Syntax. We define the syntax of the system PV-PL as follows:

- An *atomic formula* of PV-PL is either $\bot$ (denoting contradiction) or an equation in the language of PV. A *formula* of PV-PL is either an atomic formula or a composition of atomic formulas using the logical connective $\to$ for implication. This is without loss of generality, as we can define other propositional logic connectives from $\{\bot, \to\}$, for instance:

$$\neg\varphi := \varphi \to \bot; \quad \varphi \lor \psi := \neg\varphi \to \psi; \quad \varphi \land \psi := \neg(\neg\varphi \lor \neg\psi).$$

We will denote PV-PL formulas using Greek letters.

- An *assertion* of PV-PL is of form $\Gamma \vdash \varphi$, where Γ is a finite sequence of formulas and φ is a formula. Γ is called the *antecedents*, and φ is called the *consequence*.
- We say that x is a *variable* of an atomic formula is x is a variable of the PV equation. Similarly, x is a variable of a formula if it is a variable of any atomic formula inside it, and is called a variable of an assertion if it is a variable of any formula of the assertion.

- A *deduction rule* of PV-PL is written of form
$$\frac{\Gamma_1 \vdash \varphi_1 \quad \dots \quad \Gamma_k \vdash \varphi_k}{\Gamma \vdash \varphi},$$
where the assertions above the line are called *premises* and the assertion below the line is called the *conclusion.* An *axiom* of PV-PL is a deduction rule with no premise.
- A *proof* of a PV-PL assertion $\Gamma \vdash \varphi$ is a tree, where each leaf is an axiom, each internal node is a deduction rule, and the root is $\Gamma \vdash \varphi$. A PV-PL assertion is said to be *provable* if there is a proof of it.

Interpretation of assertions. Suppose that $\vec{x}$ contains all variables occurred in $\Gamma \vdash \varphi$, the intuitive interpretation of the assertion is that for every $\vec{n}$ running over the universe, if $\alpha[\vec{x}/\vec{n}]$ is true for all $\alpha \in \Gamma$, then $\varphi[\vec{x}/\vec{n}]$ is also true. That is:

- Suppose x appears in both Γ and φ, they are considered as the occurrences of the same variable, instead of different variables.
- Variables in an assertion are considered to be universally quantified.

We can define models (and the standard model) and interpretations of assertions similar to the definition of models and interpretations of PV, which is left as an exercise.

Deduction rules. There are three groups of deduction rules. Firstly, we have *structural rules* that deal with the antecedents.

- (W). The weakening rule is used to introduce a dummy condition in the antecedent of an assertion:
$$\frac{\Gamma \vdash \varphi}{\Gamma, \alpha \vdash \varphi}.$$
- (C). The contraction rule removes redundant conditions in the antecedent:
$$\frac{\Gamma, \alpha, \alpha \vdash \varphi}{\Gamma, \alpha \vdash \varphi}$$
- (P). The permutation rule allows an arbitrary permutation of conditions in the antecedent:
$$\frac{\Gamma, \alpha, \beta, \Delta \vdash \varphi}{\Gamma, \beta, \alpha, \Delta \vdash \varphi}.$$

Note that the contraction and permutation rules essentially make the antecedent a set rather than a sequence. Some authors (see, e.g., [TS00]) directly define the antecedent as a set (so that both rules are embedded) or a multiset (so that the permutation rule is embedded). We will not do that, as we will eventually translate PV-PL proofs back to PV proofs, and sets or multisets are not yet supported in PV.

Sets and multisets will be available in the next chapter, but we have to avoid circular reasoning...

We have the following logical rules that deal with assumptions, logical connectives, equations, and variables in the formulas.

- (A). This axiom is used to apply an assumption:
$$\frac{}{\Gamma, \alpha \vdash \alpha}.$$

- $(\to_i)$. This rule is used to introduce an implication connective to the conclusion, formally:
$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}.$$
- $(\to_e)$. This rule is used to eliminate an implication connective in the conclusion, formally:
$$\frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$
- $(\bot_e)$. This rule is used to eliminate a contradiction in the conclusion, which captures the standard "proof by contradiction" tactic:
$$\frac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi},$$
where $\neg\varphi$ is a shorthand of $\varphi \to \bot$.
- (V). This is the substitution rule for variables. Let x be a variable and t be a term that does not contain x. Then for any formula φ,
$$\frac{\Gamma \vdash \varphi}{\Gamma[x/t] \vdash \varphi[x/t]},$$
where $\alpha[x/t]$ denotes the formula obtained by substituting all occurrences of x in α with t, and $\Gamma[x/t] := \{\alpha \in \Gamma \mid \alpha[x/t]\}$.
- $(=_r)$. This is the reflexivity axiom of equality: $\Gamma \vdash x = x$.
- $(=_s)$. This is the symmtricity axiom of equality: $\Gamma, x = y \vdash y = x$.
- $(=_t)$. This is the transitivity axiom of equality: $\Gamma, x = y, y = z \vdash x = z$.
- $(=_/)$. This is the substitution axiom of equality: $\Gamma, x = y \vdash t[z/x] = t[z/y]$.

Exercise: Define and prove the Explosion Rule from $(\bot_e)$ and (W).

We stress that the substitution rule (V) for variables is valid because the variables in an PV-PL assertion are considered *universally* quantified. This rule essentially means that if we can prove that for all x, $\Gamma \vdash \alpha$ is true, we can substitute x in both the antecedent and the conclusion to any term t. In particular, the conclusion of the deduction rule $\Gamma[x/t] \vdash \alpha[x/t]$ could be weaker than $\Gamma \vdash \alpha$.

Finally, we have non-logical axioms that allow us to include equations in PV and some useful facts. Concretely:

- (PV). This axiom is used to apply a PV-provable equation. Concretely, for every PV provable equation e, we have $\Gamma \vdash e$.
- $(\mathrm{D}\varepsilon)$. This axiom distinguishes the empty string ε and any non-empty string. For $i \in \{0, 1\}$, we have $\Gamma \vdash \varepsilon \neq s_i(x)$, where for all terms t_1 and t_2, $t_1 \neq t_2$ is a shorthand of $\neg(t_1 = t_2)$.
- $(\mathrm{D}i)$. This axiom distinguishes 0 and 1: $\Gamma \vdash s_0(x) \neq s_1(y)$.

It can be verified that $(\mathrm{D}\varepsilon)$ and $(\mathrm{D}i)$ are not necessary and can be deduced from (PV). We choose to include them as they are quite useful :)

Recall that the main reason that PV only includes a restricted version of structural induction rather than the standard version is that it is not clear how to define *conditional proofs* in PV. As we have already introduced the notion of assertions, we can now formulate the standard structural induction scheme as:

- (Ind$_n$). Suppose that t_1, t_2 are terms and $x_1, \cdots, x_n$ are variables, $n \in \mathbb{N}$. Then we have the following rule:

$$\frac{\begin{array}{c}\forall j \in [n] : \ \Gamma \vdash t_1[x_j/\varepsilon] = t_2[x_j/\varepsilon]; \quad \forall i_1 \ldots, i_n \in \{0,1\} : \\ \Gamma, t_1 = t_2 \vdash t_1[x_1/s_{i_1}(x_1), x_n/s_{i_n}(x_n)] = t_2[x_1/s_{i_1}(x_1), \ldots, x_n/s_{i_n}(x_n)]\end{array}}{\Gamma \vdash t_1 = t_2}.$$

  Note that the quantification over j and $i_1, \ldots, i_n$ is running in the meta-theory, instead of the proof system. That is, if we have all $2^n + n$ assumptions above the line, where $j \in [n]$ and $i_1, \ldots, i_n \in \{0,1\}$, we can prove the conclusion below the line.

We note that (Ind$_n$) can be used to implement the method of the case study (see Theorem 2.4.5) by simply ignoring the induction hypothesis $t_1 = t_2$, where "ignoring the induction hypothesis" can be formally done by the weakening rule (W).

Interpretation of the deduction rules. One may read a natural deduction proof from the root to the bottom as a goal-directed proof search. We use the elimination rule of $\bot_e$ (i.e. proof by contradiction) as an example. To prove that $\Gamma \vdash \alpha$, we can assume, towards a contradiction, that α is false and prove a contradiction, i.e., deduce $\bot$ from $\Gamma, \neg\alpha$.

For instance, the rule $\frac{Y}{X}$ can be read as: To prove the assertion X, we only need to prove Y...

Similarly, the induction rule (Ind$_n$) can be interpreted as follows. Suppose that we want to prove $\Gamma \vdash t_1 = t_2$. It suffices to prove that

- We can prove the identity if one of $x_1, \ldots, x_n$ is an empty string. That is, for every $j \in [n]$, we can prove $t_1 = t_2$ from Γ if we substitute x_j/ε.
- From Γ and $t_1 = t_2$, we can prove the identity $t_1 = t_2$ if we append a bit to each of $x_1, \ldots, x_n$. That is, for every $i_1, \ldots, i_n \in \{0,1\}$, we can deduce $t_1[x_j/s_{i_j}(x_j)\ (\forall j)] = t_2[x_j/s_{i_j}(x_j)\ (\forall j)]$ from $t_1 = t_2$ and Γ.

This exactly captures the informal version of structural induction as we discussed in the first section.

The substitution rule (V) for variables, in the interpretation, is the tactic of proof by *generalization*. Suppose that we want to prove a fact $\varphi[x/t]$ from $\Gamma[x/t]$, the substitution rule (V) suggests that we can prove a more general form φ from Γ, where we pick a *fresh* variable x to replace some occurrences of the term t. Note that it is said to be more general as from $\Gamma \vdash \varphi$ we can deduce $\Gamma[x/t'] \vdash \varphi[x/t']$ for *any* term t' instead of just for $t = t'$.

Note that since the goal-directed proof search interpretation is closer to the standard writing style of mathematical reasoning, we will mostly write PV-PL proofs in this way. Nevertheless, it should be straightforward to translate such proofs into a proof tree in PV-PL.

The rule of cut. One important proof tactic in informal mathematics is to propose a *lemma*, prove that the lemma suffices to derive the conclusion, and then prove the lemma. This is called the rule of *cut*, formally denoted as

$$(\text{Cut}) : \quad \frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash \psi}{\Gamma \vdash \psi}$$

where $\Gamma \vdash \psi$ is the final goal, and φ is the “lemma”, or called the *cut formula*. Indeed, the rule of cut is *admissible* in that it can be implemented by the axioms and rules in PV-PL. Concretely, we can use the following proof tree:

$$\dfrac{\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\,(\to_i) \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\,(\to_e)$$

to simulate the rule of cut.

3.2 Warmup: Reasoning in PV-PL

We first provide two examples to demonstrate the power of the system PV-PL. The first example considers a basic property of strings: If $s_i(x) = s_i(y)$, then $x = y$. The second example considers a form of the correctness of EQ (see Lemma 2.7.1).

Example 3.2.1. We will show that $s_i(x) = s_i(y) \vdash x = y$ for $i \in \{0, 1\}$. To see this, we first apply the substitution rule of equality $(=_/)$ to prove that

$$s_i(x) = s_i(y) \vdash \mathsf{TR}(s_i(x)) = \mathsf{TR}(s_i(y)).$$

Therefore by the rule of cut, it suffices to prove that

$$s_i(x) = s_i(y), \mathsf{TR}(s_i(x)) = \mathsf{TR}(s_i(y)) \vdash x = y.$$

Note that we can prove by the definition axiom of TR (using the rule (PV)) and weakening (to introduce dummy conditions) we can prove that

$$s_i(x) = s_i(y), \mathsf{TR}(s_i(x)) = \mathsf{TR}(s_i(y)) \vdash x = \mathsf{TR}(s_i(x)),$$

and therefore by the rule of cut, it suffices to prove that

$$s_i(x) = s_i(y), \mathsf{TR}(s_i(x)) = \mathsf{TR}(s_i(y)), x = \mathsf{TR}(s_i(x)) \vdash x = y.$$

We can introduce $\mathsf{TR}(s_i(y)) = y$ in the antecedent using the rule of cut as above, and it suffices to prove that

$$s_i(x) = s_i(y), \mathsf{TR}(s_i(x)) = \mathsf{TR}(s_i(y)), x = \mathsf{TR}(s_i(x)), \mathsf{TR}(s_i(y)) = y \vdash x = y.$$

Then by the substitution (generalization) rule (V), it suffices to prove

$$s_i(x) = s_i(y), z_1 = z_3, z_2 = z_1, z_3 = z_4 \vdash z_2 = z_4.$$

The rest of the proof is straightforward applications of the transitivity of equality and the rule of cut, which is left as an exercise.

Example 3.2.2. As an example to show the power of the induction rule in PV-PL, we consider the correctness of equality (see Lemma 2.7.1), that is:

$$\vdash \mathsf{EQ}(y_1, y_2) = 1 \to y_1 = y_2,$$

where the conditional equations are replaced by the logical connective → that is natively supported in PV-PL.

Implementing the rules require a full unwinding of Theorem 3.3.4, which is painful... I can tell because I have tried :(

To prove this, we first apply the introduction rule of → to move the assumption into the antecedent, and therefore it suffices to prove $\mathsf{EQ}(y_1, y_2) = 1 \vdash y_1 = y_2$. We perform induction on y_1 and y_2 simultaneously using (Ind_2). This introduces the following subgoals:

- (Base Case 1). $\mathsf{EQ}(\varepsilon, y_2) = 1 \vdash \varepsilon = y_2$. To prove this, we apply a case study on y_2 using the induction rule and the weakening rule, which introduces the following sub-goals:
  - (Base Case 1.*i*). $\mathsf{EQ}(\varepsilon, \varepsilon) = 1 \vdash \varepsilon = \varepsilon$. This follows from a weakening (that removes the assumption), a generalization using (V) with $t = \varepsilon$, and applying the rule of reflexivity.
  - (Base Case 1.*ii*). Let $i \in \{0, 1\}$, we need to prove $\mathsf{EQ}(\varepsilon, s_i(x)) = 1 \vdash \varepsilon = s_i(x)$. Recall that by the definition axiom of EQ we have that $\mathsf{PV} \vdash \mathsf{EQ}(\varepsilon, s_i(x)) = 0$, and therefore by the ($\mathsf{PV}$) rule as well as weakening we can obtain that

    $$\mathsf{EQ}(\varepsilon, s_i(x)) = 1 \vdash \mathsf{EQ}(\varepsilon, s_i(x)) = 0.$$

    By the rule of cut, it suffices to prove that

    $$\mathsf{EQ}(\varepsilon, s_i(x)) = 1, \mathsf{EQ}(\varepsilon, s_i(x)) = 0 \vdash \varepsilon = s_i(x).$$

    By the explosion rule (which can be simulated by the elimination rule of ⊥ and weakening), it suffices to prove that

    $$\mathsf{EQ}(\varepsilon, s_i(x)) = 1, \mathsf{EQ}(\varepsilon, s_i(x)) = 0 \vdash \bot. \tag{3.1}$$

    From the antecedent we can deduce that $0 = \mathsf{EQ}(\varepsilon, s_i(x))$ (by symmetricity) and subsequently $0 = 1$ (by transitivity). Also, from the axiom (D*i*) we know that $0 \neq 1$, i.e., $\vdash 0 = 1 \to \bot$. Therefore, by applying the elimination rule of → we can conclude Equation (3.1), which concludes the sub-goal.
- (Base Case 2). $\mathsf{EQ}(y_1, \varepsilon) = 1 \vdash y_1 = \varepsilon$. The proof is almost identical to the proof of the previous case and is therefore omitted.
- (Base Case 3). $\mathsf{EQ}(s_i(y_1), s_j(y_2)) = 1, y_1 = y_2 \vdash s_i(y_1) = s_j(y_2)$, where $i = j$. This can be proved by weakening (to remove the first assumption) and then applying the substitution rule of equality ($=_/$).

- (Base Case 4). $\mathsf{EQ}(s_i(y_1), s_j(y_2)) = 1, y_1 = y_2 \vdash s_i(y_1) = s_j(y_2)$, where $i \neq j$. In this case, we know from the definition rule that $\mathsf{EQ}(s_i(y_1), s_j(y_2)) = 0$. The rest of the proof is similar to that of Base Case 1, which is left as an exercise.

Through these examples, one can notice that an informal proof almost directly translates to a natural deduction proof in PV-PL. In the rest of the note, we may write informal proofs that can be translated to PV-PL proofs straightforwardly.

Remark 3.2.1. One can verify that the PV-PL rules for logical connectives are complete for propositional logic in the sense that if φ can be derived from Γ only through propositional logic deduction, then $\Gamma \vdash \varphi$ is PV-PL provable. Moreover, this is also true for connectives defined from $\{\to, \bot\}$ such as $\wedge$ and $\vee$.

We say that a rule is *admissible* in PV-PL if it can be implemented by PV-PL proof trees. In particular, the following rules for $\wedge$ and $\vee$ can be implemented by PV-PL proof trees.

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}(\wedge_e^1) \quad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}(\wedge_e^2) \quad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}(\wedge_i)$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}(\vee_i^1) \quad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}(\vee_i^2) \quad \frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \alpha \quad \Gamma, \psi \vdash \alpha}{\Gamma \vdash \beta}(\vee_e)$$

Remark 3.2.2. We also note that the axiom of excluded middle, i.e., $\vdash \varphi \vee \neg\varphi$, can be proved by the $(\bot_e)$ rule using the following proof tree:

$$\dfrac{\dfrac{(\varphi \vee \neg\varphi) \to \bot \vdash (\varphi \to \bot) \to \bot \qquad (\varphi \vee \neg\varphi) \to \bot \vdash \varphi \to \bot}{(\varphi \vee \neg\varphi) \to \bot \vdash \bot}(\to_e)}{\varphi \vee \neg\varphi}(\bot_e)$$

The proofs for the two leaves are similar. Notice that the second leaf node can be proved as follows:

$$\dfrac{\dfrac{(\varphi \vee \neg\varphi) \to \bot, \varphi \vdash \varphi \vee \neg\varphi \qquad (\varphi \vee \neg\varphi) \to \bot, \varphi, \varphi \vee \neg\varphi \vdash \bot}{(\varphi \vee \neg\varphi) \to \bot, \varphi \vdash \bot}(\text{Cut})}{(\varphi \vee \neg\varphi) \to \bot \vdash \varphi \to \bot}(\to_i)$$

where the two leaves can be easily proved by the rules for disjunction and implication and if left as an exercise.

3.3 The Translation Theorem

Now we are ready to formulate and prove a theorem that connects PV-PL and PV. By the (PV) rule in PV-PL we know that a provable equation in PV is also provable in PV-PL, and therefore PV-PL is an extension of PV-PL. Moreover, we can prove that the extension does not introduce any provable equation that is not provable in PV. Formally:

Theorem 3.3.1 (implicit in [Coo75]). *Let e be an equation in* PV*. Then $\vdash e$ is provable in* PV-PL *if and only if* PV $\vdash e$*.*

Such an extension T' of a theory T that does not introduce new provable theorems that can be formulated in the language of a theory T is called a *conservative extension.* In particular, the theorem suggests that PV-PL is a conservative extension of the theory PV.

Indeed, we will prove a stronger result, called the *translation theorem*, that provides an explicit embedding of PV-PL assertions $\Gamma \vdash \varphi$ into PV equations $[\Gamma \vdash \varphi]_{\mathsf{PV}}$ such that a proof of φ can be translated back to a PV proof of $[\Gamma \vdash \varphi]_{\mathsf{PV}}$.

An extension of conditional equations. To define the embedding $[\cdot]_{\mathsf{PV}}$, we need to introduce an extension of conditional equations. Since $t \Rightarrow e$ itself is an equation, we can also talk about a "conditional conditional equation" $t_1 \Rightarrow (t_2 \Rightarrow e)$, or even with more $(\Rightarrow)$'s. We may remove the parentheses by assuming that $\Rightarrow$ is right-associative.

We slightly extend the notation. For a sequence of terms $t_c^{(1)}, \ldots, t_c^{(\ell)}$, $\ell \in \mathbb{N}$, we define

$$t_c^{(1)}, \ldots, t_c^{(\ell)} \Rightarrow t_1 = t_2$$

as

$$t_c^{(1)} \Rightarrow \ldots \Rightarrow t_c^{(\ell)} \Rightarrow t_1 = t_2,$$

which intuitively means that the conjunction of all conditions $t_c^{(1)}, \ldots, t_c^{(\ell)}$ imply $t_1 = t_2$.

We use $\mathsf{RHS}[e]$ and $\mathsf{LHS}[e]$ to represent the RHS and LHS of the conditional equation e. Note that they are not PV functions but meta-functions (in the meta-theory) dealing with formulas. We use $e_1 \equiv e_2$ to denote that the terms (or equations) e_1 and e_2 are the same terms (or equations).

Nevertheless, they should be feasible assuming reasonable encoding of PV equations.

The PV Translation. Let $\Gamma \vdash \varphi$ be an assertion, we define the PV translation of it, denoted by $[\Gamma \vdash \varphi]_{\mathsf{PV}}$, as a PV equation defined as follows:

- (*Translation of Assertion*): Let $\Gamma = (\alpha_1, \ldots, \alpha_n)$, $n \in \mathbb{N}$. We define

  $$[\Gamma \vdash \varphi]_{\mathsf{PV}} := \text{“}[\alpha_n]_{\mathsf{PV}} \Rightarrow [\alpha_{n-1}]_{\mathsf{PV}} \Rightarrow \ldots \Rightarrow [\alpha_1]_{\mathsf{PV}} \Rightarrow [\varphi]_{\mathsf{PV}} = 1\text{”},$$

  where $[\alpha]_{\mathsf{PV}}$ is the PV translation of formulas that will be defined later, and $t_c \Rightarrow t_1 = t_2$ denotes conditional equation. In particular, if $n \in \mathbb{N}$, we have

  $$[\Gamma \vdash \varphi]_{\mathsf{PV}} := [\varphi]_{\mathsf{PV}} = 1.$$

- (*Translation of Atomic Formula*). We define

  $$\begin{aligned} &[\bot]_{\mathsf{PV}} := 0, \\ &[t = s]_{\mathsf{PV}} := \text{“}\mathsf{EQ}(t, s)\text{”}. \end{aligned}$$

- (*Translation of the Connective*). We define

  $$[\varphi \to \psi]_{\mathsf{PV}} := \mathsf{ITE}([\varphi]_{\mathsf{PV}}, [\psi]_{\mathsf{PV}}, 1).$$

Proposition 3.3.2. *Let φ be a formula. Then* $\mathsf{PV} \vdash \mathsf{IsNotEps}([\varphi]_{\mathsf{PV}}) = 1$.

Proof Sketch. We perform induction (in the meta-theory) on the outermost connective of φ. If φ is an atomic formula, then either $\varphi = \bot$ or φ is a PV equation. In the first case it suffices to prove that $\mathsf{PV} \vdash \mathsf{IsNotEps}(0) = 1$, which follows from a simple unfolding of $\mathsf{IsNotEps}$, while in the latter case it suffices to prove that $\mathsf{PV} \vdash \mathsf{IsNotEps}(\mathsf{EQ}(x, y)) = 1$, which follows from a simultaneous induction on x and y using Theorem 2.6.3.

Suppose that $\varphi := \psi_1 \to \psi_2$. From the induction hypothesis, we know that $\mathsf{PV} \vdash \mathsf{IsNotEps}([\psi_1]_{\mathsf{PV}}) = 1$ and $\mathsf{IsNotEps}([\psi_2]_{\mathsf{PV}}) = 1$. We will need to prove that

$$\mathsf{PV} \vdash \mathsf{IsNotEps}(\mathsf{ITE}([\psi_1]_{\mathsf{PV}}, [\psi_2]_{\mathsf{PV}}, 1)) = 1. \tag{3.2}$$

Indeed, we will prove that

$$\mathsf{PV} \vdash \mathsf{IsNotEps}(x) \Rightarrow \mathsf{IsNotEps}(y) \Rightarrow \mathsf{IsNotEps}(\mathsf{ITE}(x, y, 1)) = 1. \tag{3.3}$$

If this is provable, we can derive Equation (3.2) by the substitution $x/[\psi_1]_{\mathsf{PV}}$, $y/[\psi_2]_{\mathsf{PV}}$ and applying Modus Ponens (see Proposition 2.4.10).

It remains to prove Equation (3.3). We perform a case study on x and y using Theorem 2.4.5, where all 9 cases can be proved by simply unfolding of $\mathsf{IsNotEps}$ and ITE. The details are omitted. □

Proposition 3.3.3. *Let φ be a formula. Then* $\mathsf{PV} \vdash \mathsf{TR}([\varphi]_{\mathsf{PV}}) = \varepsilon$.

The proof is similar to the proposition above. The detail is omitted.

The translation theorem claims that the PV translation of an assertion preserves its provability. Formally:

Theorem 3.3.4 (Translation Theorem)**.** *Let $\Gamma \vdash \varphi$ be a* $\mathsf{PV\text{-}PL}$ *assertion. Then $\Gamma \vdash \varphi$ is provable in* $\mathsf{PV\text{-}PL}$ *if and only if* $\mathsf{PV} \vdash [\Gamma \vdash \varphi]_{\mathsf{PV}}$.

To see that the translation theorem implies Theorem 3.3.1, notice that if we take $\Gamma = \varnothing$ and $\varphi := s = t$, we will have that $[\Gamma \vdash \varphi]_{\mathsf{PV}} =$ “$\mathsf{EQ}(s, t) = 1$. Therefore, suppose that $\vdash s = t$ is provable in $\mathsf{PV\text{-}PL}$, we have that $\mathsf{PV} \vdash \mathsf{EQ}(s, t) = 1$, which implies that $\mathsf{PV} \vdash s = t$ by Lemma 2.7.1.

Road map to the proof of the translation theorem. The proof of the translation theorem is highly constructive. Indeed, we will show that each deduction rule in $\mathsf{PV\text{-}PL}$ is PV-*admissible*, in the sense that if the PV translation of all premises of the rule are PV-provable, then the PV translation of the conclusion is also PV-provable; this implies the proof of translation theorem by structural induction on the $\mathsf{PV\text{-}PL}$ proof tree.

In the rest of this chapter, we will prove the admissibility of all $\mathsf{PV\text{-}PL}$ rules, which concludes the translation theorem.

3.4 Admissibility of Structural Rules

We prove the admissibility of the structural rules, including the weakening, contraction, and permutation rules that deal with the antecedents.

3.4.1 Admissibility of Weakening

Lemma 3.4.1. *The weakening rule is admissible. That is, for any assertion $\Gamma \vdash \varphi$, suppose that $\mathsf{PV} \vdash [\Gamma \vdash \varphi]_{\mathsf{PV}}$, then $\mathsf{PV} \vdash [\Gamma, \alpha \vdash \varphi]_{\mathsf{PV}}$.*

Let $\Gamma = (\beta_1, \dots, \beta_n)$, $n \in \mathbb{N}$. By the definition of the PV translation, we know that it suffices to prove that

$$\frac{\mathsf{PV} \vdash [\beta_n]_{\mathsf{PV}} \Rightarrow \dots \Rightarrow [\beta_1]_{\mathsf{PV}} \Rightarrow [\varphi]_{\mathsf{PV}} = 1}{\mathsf{PV} \vdash [\alpha]_{\mathsf{PV}} \Rightarrow [\beta_n]_{\mathsf{PV}} \Rightarrow \dots \Rightarrow [\beta_1]_{\mathsf{PV}} \Rightarrow [\varphi]_{\mathsf{PV}} = 1}.$$

Moreover, it suffices to prove that:

Proposition 3.4.2. *Let t_c, t_1, t_2 be terms. If $\mathsf{PV} \vdash t_1 = t_2$ and $\mathsf{PV} \vdash \mathsf{IsNotEps}(t_c) = 1$, $\mathsf{PV} \vdash t_c \Rightarrow t_1 = t_2$.*

To see this, we can treat $[\beta_n]_{\mathsf{PV}} \Rightarrow \dots \Rightarrow [\beta_1]_{\mathsf{PV}} \Rightarrow [\varphi]_{\mathsf{PV}} = 1$ as the equation $s = t$ and $[\alpha]_{\mathsf{PV}}$ as the term t', and that $\mathsf{PV} \vdash \mathsf{IsNotEps}([\alpha]_{\mathsf{PV}}) = 1$ by Proposition 3.3.2.

Furthermore, the following proposition implies the fact above by the substitution z/t', x/s, y/t, and Modus Ponens (see Proposition 2.4.10).

Proposition 3.4.3. $\mathsf{PV} \vdash \mathsf{IsNotEps}(z) \Rightarrow \mathsf{EQ}(x, y) \Rightarrow z \Rightarrow x = y$*.*

It remains to prove the proposition.

Proposition 3.4.4. PV *proves that*

- $\mathsf{LHS}[s_1(z) \Rightarrow x = y] = \mathsf{LHS}[x = y]$
- $\mathsf{RHS}[s_1(z) \Rightarrow x = y] = \mathsf{RHS}[x = y]$

Proof Sketch. Both equations can be proved by a straightforward unfolding of ITE in the definition of conditional equations. □

Proof of Proposition 3.4.3. We can prove this by a case study on z using Theorem 2.4.5. In the case of z/ε, $\mathsf{IsNotEps}(\varepsilon) = 0$, and thus we prove the equation by the Explosion Rule (see Proposition 2.4.11). Otherwise, for $i \in \{0, 1\}$, we need to prove the equation after the substitution $z/s_i(z)$. In either case, we know that $\mathsf{IsNotEps}(s_i(z)) = 1$, and thus by Proposition 3.4.4

$$\mathsf{PV} \vdash \mathsf{LHS}[\mathsf{IsNotEps}(s_i(z)) \Rightarrow e] = \mathsf{LHS}[e],$$
$$\mathsf{PV} \vdash \mathsf{RHS}[\mathsf{IsNotEps}(s_i(z)) \Rightarrow e] = \mathsf{RHS}[e],$$

where e is defined as the equation "$\mathsf{EQ}(x, y) \Rightarrow s_i(z) \Rightarrow x = y$". Subsequently, it suffices to prove $\mathsf{LHS}[e] = \mathsf{RHS}[e]$, i.e., the equation

$$e \equiv \text{“}\mathsf{EQ}(x, y) \Rightarrow s_i(z) \Rightarrow x = y\text{”}.$$

For the case that $i = 0$, notice that the conditional equation $s_0(z) \Rightarrow x = y$ is provable by the Explosion Rule (see Proposition 2.4.11), and thus by Proposition 3.4.4 we know that e is also provable.

For the case that $i = 1$, let $e' := s_1(z) \Rightarrow x = y$, notice that

$$\begin{aligned} e &\equiv \text{“}\mathsf{LHS}[e] = \mathsf{RHS}[e]\text{”} \\ &\equiv \text{“}\mathsf{ITE}(\mathsf{EQ}(x,y), \mathsf{RHS}[e'], \mathsf{LHS}[e']) = \mathsf{LHS}[e']\text{”} \\ &\equiv \text{“}\mathsf{ITE}(\mathsf{EQ}(x,y), x, \mathsf{ITE}(s_1(z), y, x)) = \mathsf{ITE}(s_1(z), y, x)\text{”}. \end{aligned}$$

By unfolding ITE, it suffices to prove that $\mathsf{ITE}(\mathsf{EQ}(x,y), x, y) = y$, i.e., $\mathsf{EQ}(x,y) \Rightarrow y = x$. This can be proved following the proof of the correctness of EQ, i.e., $\mathsf{EQ}(x,y) \Rightarrow x = y$ (see Lemma 2.7.1). □

3.4.2 Admissibility of Contraction

Lemma 3.4.5. *The contraction rule is admissible. That is, for any assertion $\Gamma, \alpha \vdash \varphi$, suppose that $\mathsf{PV} \vdash [\Gamma, \alpha, \alpha \vdash \varphi]_{\mathsf{PV}}$, then $\mathsf{PV} \vdash [\Gamma, \alpha \vdash \varphi]_{\mathsf{PV}}$.*

Similar to the case for weakening, we know by unfolding the definition of the PV translation that it suffices to prove the following meta-theorem of PV:

Proposition 3.4.6. *Let t_c, t_1, t_2 be terms. Suppose that $\mathsf{PV} \vdash \mathsf{IsNotEps}(t_c) = 1$ and $\mathsf{PV} \vdash t_c \Rightarrow t_1 = t_2$, then $\mathsf{PV} \vdash t_c \Rightarrow t_c \Rightarrow t_1 = t_2$.*

To prove the proposition, we need to further extend the syntactic of conditional equations to allow equations (as well as conditional equations) to appear in the antecedent of a conditional equation, rather than only in the conclusion of a conditional equation. Let e be an equation, we use $[e]_{\mathsf{EQ}}$ to denote the term $\mathsf{EQ}(\mathsf{LHS}[e], \mathsf{RHS}[e])$. We define the conditional equation $e \Rightarrow s = t$ as the equation $[e]_{\mathsf{EQ}} \Rightarrow s = t$. Similarly, the equation

$$\text{“}(e_1 \Rightarrow e_2) \Rightarrow s = t\text{”} \equiv \text{“}[[e_1]_{\mathsf{EQ}} \Rightarrow \mathsf{LHS}[e_2] = \mathsf{RHS}[e_2]]_{\mathsf{EQ}} \Rightarrow s = t''.$$

We can therefore allow the formulation of an arbitrary composition of equations and the condition symbol $\Rightarrow$.

By substitution and Modus Ponens (see Proposition 2.4.10), it is easy to verify that the following proposition implies Proposition 3.4.6.

Proposition 3.4.7. $\mathsf{PV} \vdash \mathsf{IsNotEps}(z) \Rightarrow (x = y) \Rightarrow z \Rightarrow x = y$.

Proof Sketch. The proof is done by a case study on z using Theorem 2.4.5. We will only demonstrate the case for $z/s_1(z)$, while the other two cases z/ε and $z/s_0(z)$ are left as an exercise.

By the definition of the conditional equation, we know that $s_1(z) \Rightarrow x = y$ is the equation $\mathsf{ITE}(s_1(z), y, x) = x$. It then suffices to prove that

$$\mathsf{PV} \vdash \mathsf{EQ}(x,y) \Rightarrow \mathsf{ITE}(s_1(z), y, x) = x,$$

which is the equation

$$\mathsf{PV} \vdash \mathsf{ITE}(\mathsf{EQ}(x,y), x, \mathsf{ITE}(s_1(z), y, x)) = \mathsf{ITE}(s_1(z), y, x).$$

(Recall that we can add a dummy condition $\mathsf{IsNotEps}(z)$ by Proposition 3.4.2.)

Note that $\mathsf{PV} \vdash \mathsf{ITE}(s_1(z), y, x) = y$. Therefore, it suffices to prove that $\mathsf{PV} \vdash \mathsf{ITE}(\mathsf{EQ}(x,y), x, y) = y$, which is exactly $\mathsf{PV} \vdash \mathsf{EQ}(x,y) \Rightarrow y = x$. This can be proved similar to the proof of Lemma 2.7.1. □

3.4.3 Admissibility of Permutation

Lemma 3.4.8. *The permutation rule is admissible. That is, for any assertion* $\Gamma, \alpha, \beta, \Delta \vdash \varphi$*, suppose that* $\mathsf{PV} \vdash [\Gamma, \alpha, \beta, \Delta \vdash \varphi]_{\mathsf{PV}}$*, then* $\mathsf{PV} \vdash [\Gamma, \beta, \alpha, \Delta \vdash \varphi]_{\mathsf{PV}}$.

Again, to prove the admissibility of permutation, it suffices to prove the following meta-theorem of PV:

Proposition 3.4.9. *Let* $s_1, \dots, s_n, t_c^1, t_c^2, t_1, t_2$ *be terms,* $n \in \mathbb{N}$*. Suppose that* $\mathsf{PV} \vdash \mathsf{IsNotEps}(t_c^i) = 1$ *for* $i \in \{0, 1\}$*, and that* $\mathsf{PV} \vdash s_n \Rightarrow \dots \Rightarrow s_1 \Rightarrow t_c^1 \Rightarrow t_c^2 \Rightarrow t_1 = t_2$*, then*

$$\mathsf{PV} \vdash s_n \Rightarrow \dots \Rightarrow s_1 \Rightarrow t_c^2 \Rightarrow t_c^1 \Rightarrow t_1 = t_2.$$

This can be easily derived from the following proposition:

Proposition 3.4.10. *Let* $n \in \mathbb{N}$*. Then* PV *proves*

$$\begin{aligned}&\mathsf{IsNotEps}(z_1) \Rightarrow \mathsf{IsNotEps}(z_2) \Rightarrow (w_n \Rightarrow \dots \Rightarrow w_1 \Rightarrow z_1 \Rightarrow z_2 \Rightarrow x = y)\\ &\quad \Rightarrow (w_n \Rightarrow \dots \Rightarrow w_1 \Rightarrow z_2 \Rightarrow z_1 \Rightarrow x = y).\end{aligned}$$

We will prove this by a case study on z using Theorem 2.4.5. The cases for z_1/ε or z_2/ε are easy. For instance, if we substitute z_2/ε, we have that $\mathsf{IsNotEps}(z_2) = 0$ and thus by the Explosion Rule (see Proposition 2.4.11),

$$\begin{aligned}\mathsf{PV} \vdash \mathsf{IsNotEps}(\varepsilon) &\Rightarrow (w_n \Rightarrow \dots \Rightarrow w_1 \Rightarrow z_1 \Rightarrow \varepsilon \Rightarrow x = y)\\ &\Rightarrow (w_n \Rightarrow \dots \Rightarrow w_1 \Rightarrow \varepsilon \Rightarrow z_1 \Rightarrow x = y).\end{aligned}$$

We can further add a dummy condition $\mathsf{IsNotEps}(z_1)$ by Proposition 3.4.2.

Now we consider the case for $z_1/s_i(z_1)$ and $z_2/s_j(z_2)$. We will prove that

$$\begin{aligned}\mathsf{PV} \vdash (w_n &\Rightarrow \dots \Rightarrow w_1 \Rightarrow s_i(z_1) \Rightarrow s_j(z_2) \Rightarrow x = y)\\ &\Rightarrow (w_n \Rightarrow \dots \Rightarrow w_1 \Rightarrow s_j(z_2) \Rightarrow s_i(z_1) \Rightarrow x = y).\end{aligned}$$

Again, if $i = 0$ or $j = 0$, we can complete the proof by the Explosion Rule (see Proposition 2.4.11) and adding dummy conditions using Proposition 3.4.2. Therefore, it suffices to deal with the case for $i = j = 1$.

This will be proved by induction on n in the meta-theory. Suppose that $n = 0$, we will need to prove that:

Proposition 3.4.11. $\mathsf{PV} \vdash (s_1(z_1) \Rightarrow s_1(z_2) \Rightarrow x = y) \Rightarrow s_1(z_1) \Rightarrow s_1(z_2) \Rightarrow x = y.$

By unfolding all the definitions, this equation will essentially reduce to the correctness of equality. The details are omitted.

Now we consider the induction case. That is, if

$$\begin{aligned}\mathsf{PV} \vdash (w_n &\Rightarrow \dots \Rightarrow w_1 \Rightarrow s_i(z_1) \Rightarrow s_j(z_2) \Rightarrow x = y)\\ &\Rightarrow (w_n \Rightarrow \dots \Rightarrow w_1 \Rightarrow s_j(z_2) \Rightarrow s_i(z_1) \Rightarrow x = y),\end{aligned}$$

then

$$\begin{aligned}\mathsf{PV} \vdash (w_{n+1} \Rightarrow \ldots \Rightarrow w_1 \Rightarrow s_i(z_1) \Rightarrow s_j(z_2) \Rightarrow x = y)\\ \Rightarrow (w_{n+1} \Rightarrow \ldots \Rightarrow w_1 \Rightarrow s_j(z_2) \Rightarrow s_i(z_1) \Rightarrow x = y),\end{aligned}$$

With the substitution $x/[w_n \Rightarrow \ldots \Rightarrow w_1 \Rightarrow s_i(z_1) \Rightarrow s_j(z_2) \Rightarrow x = y]_{\mathsf{EQ}}$, $y_1/\mathsf{LHS}[w_n \Rightarrow \ldots \Rightarrow w_1 \Rightarrow s_j(z_2) \Rightarrow s_i(z_1) \Rightarrow x = y]$, $y_2/\mathsf{RHS}[w_n \Rightarrow \ldots \Rightarrow w_1 \Rightarrow s_j(z_2) \Rightarrow s_i(z_1) \Rightarrow x = y]$, w/w_{n+1} and Modus Ponens (see Proposition 2.4.10), it suffices to prove the "transitivity" of ⇒, formalized as:

Proposition 3.4.12. $\mathsf{PV} \vdash (x \Rightarrow y_1 = y_2) \Rightarrow (w \Rightarrow x) \Rightarrow w \Rightarrow y_1 = y_2$.

This can be proved by applying a case study on w and x using Theorem 2.4.5, and using the correctness of EQ (see Lemma 2.7.1). The detail is omitted and left as an exercise.

3.5 Admissibility of Axioms for Equality

Now we consider the admissibility of axioms for equality, including the logical axioms $(=_r)$, $(=_s)$, $(=_t)$, and $(=_/)$, as well as the non-logical axioms (PV), (Dε), and (Di).

3.5.1 Non-logical Axioms about Equality

The non-logical axioms are relatively straightforward, so we quickly scan over them.

Lemma 3.5.1. *The* (PV) *axiom is admissible. That is, for any* Γ *and any* PV *provable equation* e, $\mathsf{PV} \vdash [\Gamma \vdash e]_{\mathsf{PV}}$.

Proof. Let $e \equiv t_1 = t_2$. It suffices to prove that $\mathsf{PV} \vdash [\vdash e]_{\mathsf{PV}}$ and apply the admissibility of weakening. Notice that $[\vdash e]_{\mathsf{PV}} \equiv$ "$\mathsf{EQ}(t_1, t_2) = 1$". Since $\mathsf{PV} \vdash t_1 = t_2$, we know by the correctness of EQ (see Lemma 2.7.1) that $\mathsf{EQ}(t_1, t_2) = 1$. □

Lemma 3.5.2. *The* (Dε) *axiom is admissible. That is, for any* Γ *and* $i \in \{0, 1\}$, $\mathsf{PV} \vdash [\Gamma \vdash \varepsilon \neq s_i(x)]_{\mathsf{PV}}$.

Proof. It suffices to prove that $\mathsf{PV} \vdash [\vdash \varepsilon \neq s_i(x)]_{\mathsf{PV}}$ and apply the admissibility of weakening. Notice that $[\vdash \varepsilon \neq s_i(x)]_{\mathsf{PV}} \equiv$ "$\mathsf{EQ}(\varepsilon, s_i(x)) \Rightarrow 0 = 1$". By the definition axioms of EQ we know that this conditional equation PV-provably evaluates to $0 \Rightarrow 0 = 1$, which is provable in PV by the Explosion Rule (see Proposition 2.4.11). □

Lemma 3.5.3. *The* (Di) *axiom is admissible. That is, for any* Γ, $\mathsf{PV} \vdash [\Gamma \vdash s_0(x) \neq s_1(y)]$.

Proof. It suffices to prove that $\mathsf{PV} \vdash [\vdash s_0(x) \neq s_1(y)]_{\mathsf{PV}}$ and apply the admissibility of weakening. Notice that $[\vdash s_0(x) \neq s_1(y)]_{\mathsf{PV}} \equiv$ "$\mathsf{EQ}(s_0(x), s_1(y)) \Rightarrow 0 = 1$". By the definition axioms of EQ we know that this conditional equation PV-provably evaluates to $0 \Rightarrow 0 = 1$, which is provable in PV by the Explosion Rule (see Proposition 2.4.11). □

3.5.2 Logical Axioms for Equality

Lemma 3.5.4. *The reflexivity rule* $(=_r)$ *is admissible. That is, for any* Γ*,* $\mathsf{PV} \vdash [\Gamma \vdash x = x]_{\mathsf{PV}}$.

Proof. It suffices to prove that $\mathsf{PV} \vdash [\vdash x = x]_{\mathsf{PV}}$ and apply the admissibility of weakening. Notice that $[\vdash x = x]_{\mathsf{PV}} \equiv$ "$\mathsf{EQ}(x, x) = 1$", which is provable in PV by a simple induction on x. □

Lemma 3.5.5. *The symmetricity rule* $(=_s)$ *is admissible. That is, for any* Γ*,* $\mathsf{PV} \vdash [\Gamma, x = y \vdash y = x]_{\mathsf{PV}}$.

Proof Sketch. It suffices to prove that $\mathsf{PV} \vdash [x = y \vdash y = x]_{\mathsf{PV}}$ and apply the admissibility of weakening. Notice that $[x = y \vdash y = x]_{\mathsf{PV}} \equiv$ "$\mathsf{EQ}(x, y) \Rightarrow \mathsf{EQ}(y, x) = 1$", or equivalently:

$$\mathsf{ITE}(\mathsf{EQ}(x, y), 1, \mathsf{EQ}(y, x)) = \mathsf{EQ}(y, x).$$

This can be proved by simultaneous induction on x, y using Theorem 2.6.3, which is similar to the proof of the correctness of EQ (see Lemma 2.7.1). □

Lemma 3.5.6. *The transitivity rule* $(=_t)$ *is admissible. That is, for any* Γ*,* $\mathsf{PV} \vdash [\Gamma, x = y, y = z \vdash x = z]_{\mathsf{PV}}$.

Proof Sketch. It suffices to prove that $\mathsf{PV} \vdash [x = y, y = z \vdash x = z]_{\mathsf{PV}}$ and apply the admissibility of weakening. Notice that

$$\begin{aligned}
&[x = y, y = z \vdash x = z]_{\mathsf{PV}} \\
\equiv\ &\text{“}\mathsf{EQ}(y, z) \Rightarrow \mathsf{EQ}(x, y) \Rightarrow \mathsf{EQ}(x, z) = 1\text{”} \\
\equiv\ &\text{“}\mathsf{EQ}(y, z) \Rightarrow \mathsf{ITE}(\mathsf{EQ}(x, y), 1, \mathsf{EQ}(x, z)) = \mathsf{EQ}(x, z)\text{”} \\
\equiv\ &\text{“}\mathsf{ITE}(\mathsf{EQ}(y, z), \mathsf{EQ}(x, z), \mathsf{ITE}(\mathsf{EQ}(x, y), 1, \mathsf{EQ}(x, z))) = \mathsf{ITE}(\mathsf{EQ}(x, y), 1, \mathsf{EQ}(x, z))\text{”}.
\end{aligned}$$

To see that this is provable in PV, we will perform induction simultaneously on x, y, and z using Theorem 2.6.3 (see Remark 2.6.1). Specifically, we will show that both sides of the equation are identical to the function $f(x, y, z)$ recursively defined by the equations:

$$\begin{aligned}
&f(\varepsilon, y, z) := \mathsf{ITE}(\mathsf{EQ}(\varepsilon, y), 1, \mathsf{EQ}(\varepsilon, z)), \quad f(x, \varepsilon, z) := \mathsf{ITE}(\mathsf{EQ}(x, \varepsilon), 1, \mathsf{EQ}(x, z)), \\
&\quad f(x, y, \varepsilon) := \mathsf{ITE}(\mathsf{EQ}(x, y), 1, \mathsf{EQ}(x, \varepsilon))
\end{aligned}$$

$$f(s_i(x), s_j(y), s_k(z)) := \begin{cases} f(x, y, z) & i = j = k \\ \mathsf{ITE}(\mathsf{EQ}(x, y), 1, 0) & i = j \land i \neq k \\ \mathsf{EQ}(x, z) & i = k \land i \neq j \\ 0 & j = k \land i \neq j \end{cases}$$

Note that for cases where one of x, y, z is substituted by ε, we need to further perform induction on the remaining variables. The details are omitted and left as an exercise. □

Before proving the admissibility of the substitution rule for equality, we first prove a technical lemma showing the correctness of EQ in conditional equations.

Proposition 3.5.7. $\mathsf{PV} \vdash \mathsf{IsNotEps}(x) \Rightarrow (x \Rightarrow y = z) \Rightarrow x \Rightarrow \mathsf{EQ}(y, z) = 1$. *Moreover, if for any terms* t_c, t_1, t_2*, if* $\mathsf{PV} \vdash \mathsf{IsNotEps}(t_c)$ *and* $\mathsf{PV} \vdash t_c \Rightarrow t_1 = t_2$*, then* $\mathsf{PV} \vdash t_c \Rightarrow \mathsf{EQ}(t_1, t_2) = 1$.

Proof Sketch. The "moreover" part follows straightforwardly, so we will only prove the first part. To show that $\mathsf{PV} \vdash \mathsf{IsNotEps}(x) \Rightarrow (x \Rightarrow y = z) \Rightarrow x \Rightarrow \mathsf{EQ}(y, z) = 1$, we perform a case study on z using Theorem 2.4.5. The case for z/ε can be proved using the Explosion Rule (see Proposition 2.4.11), and the case for $z/s_0(z)$ can be proved using the Explosion Rule by adding dummy conditions using Proposition 3.4.2.

For the case $z/s_1(z)$, the conditional equation PV-provably evaluates to $\mathsf{EQ}(y, z) \Rightarrow \mathsf{EQ}(y, z) = 1$, which can be proved using the fact that

$$\mathsf{PV} \vdash \mathsf{IsNotEps}(x) \Rightarrow \mathsf{TR}(x) = \varepsilon \Rightarrow x \Rightarrow x = 1$$

(which can be proved by a simple case study) and that $\mathsf{PV} \vdash \mathsf{IsNotEps}(\mathsf{EQ}(y, z)) = 1$ and $\mathsf{PV} \vdash \mathsf{TR}(\mathsf{EQ}(y, z)) = \varepsilon$ (both of which can be proved by simultaneous induction on y and z. □

Lemma 3.5.8. *The substitution rule* $(=_/)$ *for equality is admissible. That is, for any* Γ*,* $\mathsf{PV} \vdash [\Gamma, x = y \vdash t[z/x] = t[z/y]]_{\mathsf{PV}}$.

Proof Sketch. Again, it suffices to prove that $\mathsf{PV} \vdash [x = y \vdash t[z/x] = t[z/y]]_{\mathsf{PV}}$, or equivalently:

$$\mathsf{PV} \vdash \mathsf{EQ}(x, y) \Rightarrow \mathsf{EQ}(t[z/x], t[z/y]) = 1.$$

Indeed, we will prove a stronger result (by Proposition 3.5.7) that

$$\mathsf{PV} \vdash \mathsf{EQ}(x, y) \Rightarrow t[z/x] = t[z/y].$$

Suppose that $z, \vec{w}$ are the variables that occurred in t, and $f_t(z, \vec{w}) = t$ is the function defined from t using the composition rule in PV. It suffices to prove that

$$\mathsf{PV} \vdash \mathsf{EQ}(x, y) \Rightarrow f_t(x, \vec{w}) = f_t(y, \vec{w}),$$

or equivalently $\mathsf{PV} \vdash \mathsf{ITE}(\mathsf{EQ}(x, y), f_t(y, \vec{w}), f_t(x, \vec{w})) = f_t(x, \vec{w})$.

Auxiliary functions. Let $\mathsf{EQL}(x, y)$ be the function that outputs 1 (resp. 0) if and only if x and y are of the same length. It can be defined, for instance, by simultaneous recursion on x and y, provided that it proves

$$\mathsf{EQL}(\varepsilon, s_i(y)) = \mathsf{EQL}(s_j(x), \varepsilon) = 0, \quad \mathsf{EQL}(s_i(x), s_j(y)) = \mathsf{EQL}(x, y).$$

Let $\mathsf{Suf}(x, y)$ be the function that outputs the suffix of x of length $|y|$, that is:

$$\mathsf{Suf}(x, \varepsilon) := \varepsilon, \quad \mathsf{Suf}(\varepsilon, y) := \varepsilon$$
$$\mathsf{Suf}(s_i(x), s_j(y)) := s_i(\mathsf{Suf}(x, y)).$$

It can be proved by induction on x that $\mathsf{Suf}(x, y \circ x) = x$. Moreover, we can prove the following properties of Suf:

Proposition 3.5.9. PV *proves the following equations:*

- $\mathsf{EQL}(y,z) \Rightarrow \mathsf{Suf}(x,y) = \mathsf{Suf}(x,z)$.
- $\mathsf{Suf}(x, y \circ x) = x$.
- $\mathsf{Suf}(x, s_i(y)) = \mathsf{LastBit}(\mathsf{ITR}(x,y)) \circ \mathsf{Suf}(x,y)$, $i \in \{0,1\}$.
- $\mathsf{Suf}(x, x \circ y) = x$.

Hopefully I'm not making any stupid mistake here :)

Proof Sketch. The first bullet can be proved by simultaneous induction on y and z using Theorem 2.6.3. The second bullet can be proved by induction on x. The third bullet can be proved by simultaneous induction on x and y using Theorem 2.6.3, where we would need the first bullet as well as $\mathsf{EQL}(s_0(y), s_1(y)) = 1$ and Modus Ponens (see Proposition 2.4.10) to prove that $\mathsf{Suf}(x, s_0(y)) = \mathsf{Suf}(x, s_1(y))$. The last bullet can be proved by induction on y and using the second and the third bullets to prove that $\mathsf{Suf}(x, x\circ\varepsilon) = x$ and $\mathsf{Suf}(x, x \circ s_i(y)) = \mathsf{Suf}(x, x\circ y)$. □

A generalized equation. We will show that PV proves:

$$\mathsf{ITE}(\mathsf{EQ}(\mathsf{Suf}(x,z), \mathsf{Suf}(y,z)), f_t(\mathsf{ITR}(x,z)\circ \mathsf{Suf}(y,z), \vec{w}), f_t(x,\vec{w})) = f_t(x,\vec{w}). \tag{3.4}$$

To see that this suffices, we can substitute $z/x\circ y$, so that the LHS of the equation is PV-provably equal to $\mathsf{ITE}(\mathsf{EQ}(x,y), f_t(y,\vec{w}), f_t(x,\vec{w}))$ by Proposition 3.5.9.

It remains to prove Equation (3.4). We will perform induction on the auxiliary variable z using the induction rule in PV. Concretely, we will show that both sides of the equation are identical to the function $g(z,x,y,\vec{w})$ recursively defined as

$$g(\varepsilon, x, y, \vec{w}) = f_t(x,\vec{w}),$$
$$g(s_i(z), x, y, \vec{w}) = \begin{cases} f_t(x,\vec{w}) & \mathsf{EQ}(\mathsf{LastBit}(\mathsf{ITR}(x,z)), \mathsf{LastBit}(\mathsf{ITR}(y,z))) = 1; \\ g(z,x,y,\vec{w}) & \text{otherwise.} \end{cases}$$

The RHS of Equation (3.4) is identical to g.

Useful properties. Now we show that this is also true for the LHS, starting from proving a few properties that will help simplify Equation (3.4) with $z/s_i(z)$. Notice that by Proposition 3.5.9:

$$\begin{aligned} &\mathsf{EQ}(\mathsf{Suf}(x,s_i(z)), \mathsf{Suf}(y,s_i(z))) \\ &= \mathsf{EQ}(\mathsf{LastBit}(\mathsf{ITR}(x,z))\circ \mathsf{Suf}(x,z), \mathsf{LastBit}(\mathsf{ITR}(y,z))\circ\mathsf{Suf}(y,z)). \end{aligned} \tag{3.5}$$

Moreover, by simultaneous induction on x and y using Theorem 2.6.3, we can prove that $\mathsf{EQ}(\mathsf{LastBit}(z)\circ x, \mathsf{LastBit}(w)\circ y) = \mathsf{And}(\mathsf{EQ}(\mathsf{LastBit}(z), \mathsf{LastBit}(w)), \mathsf{EQ}(x,y))$, which implies that

$$(3.5) = \mathsf{And}(\mathsf{EQ}(\mathsf{LastBit}(\mathsf{ITR}(x,z)), \mathsf{LastBit}(\mathsf{ITR}(y,z))), \mathsf{EQ}(\mathsf{Suf}(x,z), \mathsf{Suf}(y,z))).$$

Also, notice that

$$\begin{aligned} &\mathsf{ITR}(x,s_i(z))\circ\mathsf{Suf}(y,s_i(z)) \\ &= \mathsf{ITR}(x,s_i(z))\circ(\mathsf{LastBit}(\mathsf{ITR}(y,z))\circ\mathsf{Suf}(y,z)). && \text{(Proposition 3.5.9)} \\ &= \mathsf{ITR}(x,s_i(z))\circ\mathsf{LastBit}(\mathsf{ITR}(y,z))\circ\mathsf{Suf}(y,z) \end{aligned}$$

Proof of Equation (3.4). Note that it is clear that the LHS of Equation (3.4) with the substitution z/ε is identical to $g(\varepsilon, x, y, \vec{w})$ by unfolding. We will now simplify Equation (3.4) with $z/s_i(z)$. The strategy is to perform a case study on whether $\mathsf{LastBit}(\mathsf{ITR}(x,z)) = \mathsf{LastBit}(\mathsf{ITR}(y,z))$. Formally, let z' be a fresh variable, and we will prove that

$$\begin{aligned}
&\begin{cases} f_t(\mathsf{TR}(z') \circ \mathsf{LastBit}(z'') \circ \mathsf{Suf}(y,z), \vec{w}) & \begin{array}{l}\mathsf{And}(\mathsf{EQ}(\mathsf{LastBit}(z'), \mathsf{LastBit}(z'')),\\ \quad \mathsf{EQ}(\mathsf{Suf}(x,z), \mathsf{Suf}(y,z))) = 1\end{array} \\ f_t(x, \vec{w}) & \text{otherwise} \end{cases} \\
=&\begin{cases} \mathsf{ITE}(\mathsf{EQ}(\mathsf{Suf}(x,z), \mathsf{Suf}(y,z)), f_t(z' \circ \mathsf{Suf}(y,z), \vec{w}), f_t(x, \vec{w})) & \mathsf{EQ}(\mathsf{LastBit}(z'), \mathsf{LastBit}(z'')) = 1; \\ f_t(x, \vec{w}) & \text{otherwise}; \end{cases} \\
\equiv&\begin{cases} \mathsf{LHS}[(3.4)] & \mathsf{EQ}(\mathsf{LastBit}(z'), \mathsf{LastBit}(z'')) = 1; \\ f_t(x, \vec{w}) & \text{otherwise}; \end{cases}
\end{aligned}$$

To see that this suffices, we can substitute

$$z'/\mathsf{ITR}(x,z), \quad z''/\mathsf{ITR}(y,z)$$

so that according to the properties we proved above, the equation shows exactly that the LHS of Equation (3.4) is identical to the function g recursively defined as above.

Finally, we prove the equation above by a case study on z' and z'' using Theorem 2.4.5. In all cases that $\mathsf{LastBit}(z') \neq \mathsf{LastBit}(z'')$, both sides of the consequence of the conditional equation evaluates to $f_t(x, \vec{w})$. Now we consider the cases that $\mathsf{LastBit}(z') = \mathsf{LastBit}(z'')$, for instance, with the substitution $z'/s_0(z')$ and $z''/s_0(z'')$. In such case, the LHS of the equation evaluates to

$$\begin{cases} f_t(z' \circ 0 \circ \mathsf{Suf}(y,z), \vec{w}) & \mathsf{EQ}(\mathsf{Suf}(x,z), \mathsf{Suf}(y,z))) = 1; \\ f_t(x, \vec{w}) & \text{otherwise}, \end{cases}$$

while the RHS evaluates to

$$\begin{aligned}
&\mathsf{ITE}(\mathsf{EQ}(\mathsf{Suf}(x,z), \mathsf{Suf}(y,z)), f_t(s_0(z') \circ \mathsf{Suf}(y,z), \vec{w}), f_t(x, \vec{w})) \\
=&\begin{cases} f_t(s_0(z') \circ \mathsf{Suf}(y,z), \vec{w}) & \mathsf{EQ}(\mathsf{Suf}(x,z), \mathsf{Suf}(y,z))) = 1; \\ f_t(x, \vec{w}) & \text{otherwise}. \end{cases}
\end{aligned}$$

Since $z' \circ 0 = s_0(z')$, we can prove that they are identical by a case study on ITE using Theorem 2.4.6. This completes the proof. □

3.6 Admissibility of Logical Rules

Now we will prove the admissibility of logical rules, including the assumption axiom (V), the substitution rule (V), and the introduction and elimination rules for logical connectives $\{\rightarrow, \bot\}$.

3.6.1 Assumption and Substitution

Lemma 3.6.1. *The assumption rule* (A) *is admissible. That is, for any* Γ *and formula* α, $\mathsf{PV} \vdash [\Gamma, \alpha \vdash \alpha]_{\mathsf{PV}}$.

Proof Sketch. Again, it suffices to prove the admissibility of $\alpha \vdash \alpha$, i.e., $\mathsf{PV} \vdash [\alpha \vdash \alpha]_{\mathsf{PV}}$. Notice that $[\alpha \vdash \alpha]_{\mathsf{PV}} \equiv$ "$[\alpha]_{\mathsf{PV}} \Rightarrow [\alpha]_{\mathsf{PV}} = 1$", which is provable in PV by the fact that

$$\mathsf{PV} \vdash \mathsf{IsNotEps}(x) \Rightarrow (\mathsf{TR}(x) = \varepsilon) \Rightarrow x \Rightarrow x = 1$$

and that $\mathsf{PV} \vdash \mathsf{IsNotEps}([\alpha]_{\mathsf{PV}})$ (see Proposition 3.3.2) and $\mathsf{PV} \vdash \mathsf{TR}([\alpha]_{\mathsf{PV}}) = \varepsilon$ (see Proposition 3.3.3) using Modus Ponens (see Proposition 2.4.10). □

Lemma 3.6.2. *The substitution rule* (V) *is admissible. That is, for any assertion* $\Gamma \vdash \alpha$ *such that* $\mathsf{PV} \vdash [\Gamma \vdash \alpha]_{\mathsf{PV}}$, *we have* $\mathsf{PV} \vdash [\Gamma[z/t] \vdash \alpha[z/t]]_{\mathsf{PV}}$ *for any term* t.

Proof Sketch. It suffices to prove that substitution commutes with the PV translation, i.e., $[\Gamma[z/t] \vdash \alpha[z/t]]_{\mathsf{PV}} \equiv [\Gamma \vdash \alpha]_{\mathsf{PV}}[z/t]$, so that the lemma follows directly from the substitution rule (L4) in PV.

We first prove that substitution commutes with conditional equations, i.e., $(s_c \Rightarrow s_1 = s_2)[z/t] \equiv s_c[z/t] \Rightarrow s_1[z/t] = s_2[z/t]$, which follows directly from the definition of conditional equations. Let $\Gamma = (\beta_1, \dots, \beta_n)$. Notice that

$$\begin{aligned} & [\Gamma \vdash \alpha]_{\mathsf{PV}}[z/t] \\ \equiv\ & [\beta_n]_{\mathsf{PV}}[z/t] \Rightarrow \dots \Rightarrow [\beta_1]_{\mathsf{PV}}[z/t] \Rightarrow [\alpha]_{\mathsf{PV}}[z/t] = 1 \\ \equiv\ & [\Gamma[z/t] \vdash \alpha[z/t]]_{\mathsf{PV}}, \end{aligned}$$

where the first equality follows from applying the fact that substitution commutes with conditional equations for n times, and the second equality follows from the definition of the translation $[\cdot]_{\mathsf{PV}}$. □

3.6.2 Rules of Implication

Next, we show that the introduction ($\rightarrow_i$) and elimination ($\rightarrow_e$) rules of implication are PV admissible.

Lemma 3.6.3. *The introduction rule of implication* ($\rightarrow_i$) *is admissible. That is, for any* Γ *and formulas* φ, ψ, *suppose that* $\mathsf{PV} \vdash [\Gamma, \varphi \vdash \psi]_{\mathsf{PV}}$, *then* $\mathsf{PV} \vdash [\Gamma \vdash \varphi \rightarrow \psi]_{\mathsf{PV}}$.

Proof Sketch. By the admissibility of the permutation rule, it suffices to prove that $\mathsf{PV} \vdash [\varphi, \Gamma \vdash \psi]_{\mathsf{PV}}$ implies that $\mathsf{PV} \vdash [\Gamma \vdash \varphi \rightarrow \psi]_{\mathsf{PV}}$.

Let $\Gamma = (\beta_1, \dots, \beta_n)$, $n \in \mathbb{N}$. By the assumption and the definition of PV translation, we have that

$$\mathsf{PV} \vdash [\beta_n]_{\mathsf{PV}} \Rightarrow \dots \Rightarrow [\beta_1]_{\mathsf{PV}} \Rightarrow [\varphi]_{\mathsf{PV}} \Rightarrow [\psi]_{\mathsf{PV}} = 1. \tag{3.6}$$

and we need to prove that

$$\mathsf{PV} \vdash [\beta_n]_{\mathsf{PV}} \Rightarrow \dots \Rightarrow [\beta_1]_{\mathsf{PV}} \Rightarrow \mathsf{ITE}([\varphi]_{\mathsf{PV}}, [\psi]_{\mathsf{PV}}, 1) = 1. \tag{3.7}$$

Proposition 3.6.4. *Let $n \in \mathbb{N}$. Then* PV *proves*

$$\begin{aligned}&\mathsf{IsNotEps}(x) \Rightarrow \mathsf{IsNotEps}(y) \Rightarrow \mathsf{TR}(x) = \varepsilon \Rightarrow \mathsf{TR}(y) = \varepsilon \Rightarrow\\&\mathsf{IsNotEps}(z_n) \Rightarrow \ldots \Rightarrow \mathsf{IsNotEps}(z_1) \Rightarrow\\&(z_n \Rightarrow \ldots \Rightarrow z_1 \Rightarrow x \Rightarrow y = 1) \Rightarrow z_n \Rightarrow \ldots \Rightarrow z_1 \Rightarrow \mathsf{ITE}(x, y, 1) = 1.\end{aligned}$$

To see that this proposition suffices, notice that with the substitution $x/[\varphi]_{\mathsf{PV}}$, $y/[\psi]_{\mathsf{PV}}$, $z_i/[\beta_i]_{\mathsf{PV}}$ for $i \in [n]$ and Proposition 3.3.2 and 3.3.3, we obtain that

$$\begin{aligned}&([\beta_n]_{\mathsf{PV}} \Rightarrow \ldots \Rightarrow [\beta_1]_{\mathsf{PV}} \Rightarrow [\varphi]_{\mathsf{PV}} \Rightarrow [\psi]_{\mathsf{PV}} = 1)\\&\quad \Rightarrow [\beta_n]_{\mathsf{PV}} \Rightarrow \ldots \Rightarrow [\beta_1]_{\mathsf{PV}} \Rightarrow \mathsf{ITE}([\varphi]_{\mathsf{PV}}, [\psi]_{\mathsf{PV}}, 1) = 1,\end{aligned}$$

where the antecedent (and thus its $[\cdot]_{\mathsf{EQ}}$ translation) is provable in PV. Therefore, by Modus Ponens (see Proposition 2.4.10), we can obtain Equation (3.7).

Proof of Proposition 3.6.4. It remains to prove Proposition 3.6.4. Similar to the proof of Proposition 3.4.10, we will prove the conditional equation by induction on n (in the meta-theory).

Consider the case for $n = 0$. That is:

$$\begin{aligned}&\mathsf{IsNotEps}(x) \Rightarrow \mathsf{IsNotEps}(y) \Rightarrow \mathsf{TR}(x) = \varepsilon \Rightarrow \mathsf{TR}(y) = \varepsilon \Rightarrow\\&x \Rightarrow y = 1 \Rightarrow \mathsf{ITE}(x, y, 1) = 1.\end{aligned}$$

To prove this, we perform a case analysis on x and y using Theorem 2.4.5, where all cases are straightforward.

It remains to consider the induction case. That is, assuming that PV proves

$$\begin{aligned}&\mathsf{IsNotEps}(x) \Rightarrow \mathsf{IsNotEps}(y) \Rightarrow \mathsf{TR}(x) = \varepsilon \Rightarrow \mathsf{TR}(y) = \varepsilon \Rightarrow\\&\mathsf{IsNotEps}(z_n) \Rightarrow \ldots \Rightarrow \mathsf{IsNotEps}(z_1) \Rightarrow\\&(z_n \Rightarrow \ldots \Rightarrow z_1 \Rightarrow x \Rightarrow y = 1) \Rightarrow z_n \Rightarrow \ldots \Rightarrow z_1 \Rightarrow \mathsf{ITE}(x, y, 1) = 1,\end{aligned}$$

we will show that PV proves

$$\begin{aligned}&\mathsf{IsNotEps}(x) \Rightarrow \mathsf{IsNotEps}(y) \Rightarrow \mathsf{TR}(x) = \varepsilon \Rightarrow \mathsf{TR}(y) = \varepsilon \Rightarrow\\&\mathsf{IsNotEps}(z_{n+1}) \Rightarrow \ldots \Rightarrow \mathsf{IsNotEps}(z_1) \Rightarrow\\&(z_{n+1} \Rightarrow \ldots \Rightarrow z_1 \Rightarrow x \Rightarrow y = 1) \Rightarrow z_{n+1} \Rightarrow \ldots \Rightarrow z_1 \Rightarrow \mathsf{ITE}(x, y, 1) = 1,\end{aligned}$$

We perform a case analysis on x and y twice (i.e. we consider what is the last and the second last bits of x and y). All cases are trivial except for the case $x/1$ and $y/0$, in which it suffices to prove that

$$\begin{aligned}&\mathsf{IsNotEps}(z_{n+1}) \Rightarrow \ldots \Rightarrow \mathsf{IsNotEps}(z_1) \Rightarrow\\&(z_{n+1} \Rightarrow \ldots \Rightarrow z_1 \Rightarrow 1 \Rightarrow 0 = 1) \Rightarrow z_{n+1} \Rightarrow \ldots \Rightarrow z_1 \Rightarrow 0 = 1.\end{aligned} \tag{3.8}$$

We perform a case study on z_{n+1}. The only non-trivial case is that $z_{n+1}/s_1(z_{n+1})$, Let e be the conditional equation $z_n \Rightarrow \ldots \Rightarrow z_1 \Rightarrow 0 = 1$. With the substitution $z_{n+1}/s_1(z_{n+1})$, Equation (3.8) PV-provably evaluates to

$$\begin{aligned}&\mathsf{IsNotEps}(z_n) \Rightarrow \ldots \Rightarrow \mathsf{IsNotEps}(z_1) \Rightarrow \\ &(z_n \Rightarrow \ldots \Rightarrow z_1 \Rightarrow 1 \Rightarrow 0 = 1) \Rightarrow z_n \Rightarrow \ldots \Rightarrow z_1 \Rightarrow 0 = 1, \end{aligned} \tag{3.9}$$

which follows from the assumption (with the same substitution $x/1$ and $y/0$). This completes the proof. □

Lemma 3.6.5. *The elimination rule of implication* $(\to_e)$ *is admissible. That is, for any* Γ *and formula* φ, ψ, *suppose that* $\mathsf{PV} \vdash [\Gamma \vdash \varphi \to \psi]_{\mathsf{PV}}$ *and* $\mathsf{PV} \vdash [\Gamma \vdash \varphi]$, *we have that* $\mathsf{PV} \vdash [\Gamma \vdash \psi]$.

Proof Sketch. We can deal with the antecedent Γ similar to the proof of the admissibility of $(\to_i)$; for simplicity of presentation, we will only demonstrate the admissibility in case that $\Gamma = \varnothing$. By the definition of the PV translation, we will need to derive $\mathsf{PV} \vdash [\psi]_{\mathsf{PV}}$ from $\mathsf{PV} \vdash \mathsf{ITE}([\varphi]_{\mathsf{PV}}, [\psi]_{\mathsf{PV}}, 1) = 1$ and $\mathsf{PV} \vdash [\varphi]_{\mathsf{PV}} = 1$. Therefore, it suffices to prove that:

$$\mathsf{PV} \vdash \mathsf{IsNotEps}(y) \Rightarrow (\mathsf{TR}(y) = \varepsilon) \Rightarrow (x = 1) \Rightarrow (\mathsf{ITE}(x, y, 1) = 1) \Rightarrow y = 1. \tag{3.10}$$

To see that this suffices, notice that we can substitute $x/[\varphi]_{\mathsf{PV}}$, $y/[\psi]_{\mathsf{PV}}$, and apply Modus Ponens (see Proposition 2.4.10).

To prove Equation (3.10), we perform a case analysis on x and y using Theorem 2.4.5, and all cases can be proved directly by unfolding with the Explosion Rule (see Proposition 2.4.11) and the trick of adding dummy conditions (see Proposition 3.4.2). □

3.6.3 Rules of Contradiction

The last logical rule we need to consider is the elimination rule of $\bot$, i.e., "proof by contradiction".

Lemma 3.6.6. *The elimination rule* $(\bot_e)$ *of* $\bot$ *is admissible. That is, for any* Γ *and any formula* α, *suppose that* $\mathsf{PV} \vdash [\Gamma, \neg\alpha \vdash \bot]_{\mathsf{PV}}$, *then* $\mathsf{PV} \vdash [\Gamma \vdash \alpha]_{\mathsf{PV}}$.

Proof Sketch. We will prove that the axiom of double negation elimination is admissible in PV, that is:

$$\mathsf{PV} \vdash [\Gamma \vdash \neg\neg\alpha \to \alpha]_{\mathsf{PV}}.$$

To see that this suffices, notice that the elimination rule $(\bot_e)$ can be simulated by the following proof tree using double negation elimination using the introduction and elimination rules of $\to$ that have already proved to be admissible:

$$\dfrac{\dfrac{}{\Gamma \vdash \neg\neg\alpha \to \alpha} \quad \dfrac{\Gamma, \neg\alpha \vdash \bot}{\Gamma \vdash \neg\neg\alpha}\,(\to_i)}{\Gamma \vdash \alpha}\,(\to_e)$$

Moreover, by the admissibility of weakening (W), it suffices to prove the admissibility of the case $\Gamma = \varnothing$. That is,

$$\begin{aligned} & \mathsf{PV} \vdash [\vdash \neg\neg\alpha \to \alpha]_{\mathsf{PV}} \\ \iff & \mathsf{PV} \vdash \mathsf{ITE}(\mathsf{ITE}(\mathsf{ITE}([\alpha]_{\mathsf{PV}}, 0, 1), 0, 1), [\alpha]_{\mathsf{PV}}, 1) = 1. \end{aligned}$$

Furthermore, it suffices to prove the more general version:

$$\mathsf{PV} \vdash \mathsf{IsNotEps}(x) \Rightarrow \mathsf{ITE}(\mathsf{ITE}(\mathsf{ITE}(x, 0, 1), 0, 1), x, 1) = 1,$$

which can be proved by a case study on x using Theorem 2.4.5. □

3.7 Admissibility of Structural Induction

Finally, we prove that the structural induction rule is admissible in PV.

I guess there is no need to "recall", dear fellow feasible mathematicians :)

Recall that in the definition of PV, we only provide a restricted version of structural induction in the sense that if two functions are both identical to a recursively defined feasible function, we can conclude that the two functions are identical. The induction rule in PV-PL, however, captures the generic structural induction principle as discussed in Postulate 3. Therefore, the feasibility of the induction rule in PV-PL shows that the restricted structural induction rule in PV suffices to capture the informal notion of structural induction.

Remark 3.7.1. For simplicity, we will only prove the feasibility of (Ind_2); nevertheless, the admissibility of (Ind_n) clearly follows from the same technique.

Lemma 3.7.1. *The structural induction rule* (Ind_2) *is admissible. That is, for any terms* t_1, t_2, Γ, *and variables* x_1, x_2, *suppose that* PV *proves*

- $[\Gamma \vdash t_1[x_j/\varepsilon] = t_2[x_j/\varepsilon]]_{\mathsf{PV}}$, *where* $j \in \{1, 2\}$.
- $[\Gamma, t_1 = t_2 \vdash t_1[x_1/s_{i_1}(x_1), x_2/s_{i_2}(x_2)] = t_2[x_1/s_{i_1}(x_1), x_2/s_{i_2}(x_2)]]_{\mathsf{PV}}$, *where* $i_1, i_2 \in \{0, 1\}$.

Then $\mathsf{PV} \vdash [\Gamma \vdash t_1 = t_2]_{\mathsf{PV}}$.

Proof. By the admissibility of the permutation rule, we can change the second bullet to $[t_1 = t_2, \Gamma \vdash t_1[x_1/s_{i_1}(x_1), x_2/s_{i_2}(x_2)] = t_2[x_1/s_{i_1}(x_1), x_2/s_{i_2}(x_2)]]_{\mathsf{PV}}$. Moreover, we can (without loss of generality) replace t_1, t_2 by the PV functions symbols f_1, f_2 defined by $f_i(x_1, x_2, \vec{w}) = t_i$, $i \in \{1, 2\}$, where $\vec{w}$ denote other variables in t_1, t_2. For simplicity, we will ignore $\vec{w}$ in the proof and write $f_i(x_1, x_2)$.

Let $\Gamma = (\beta_1, \dots, \beta_n)$. We will prove the admissibility in PV by induction on n (in the meta-theory). Concretely, we will prove that for any PV terms $u_1, \dots, u_n$, if PV proves

- $\mathsf{IsNotEps}(u_i) = 1$ for any $i \in [n]$,
- $u_n \Rightarrow \dots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(\varepsilon, x_2), f_2(\varepsilon, x_2)) = 1$,
- $u_n \Rightarrow \dots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(x_1, \varepsilon), f_2(x_1, \varepsilon)) = 1$,
- $u_n \Rightarrow \dots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)) \Rightarrow \mathsf{EQ}(f_1(s_{i_1}(x_1), s_{i_2}(x_2)), f_2(s_{i_1}(x_1), s_{i_2}(x_2))) = 1$, for any $i_1, i_2 \in \{0, 1\}$,

Then PV proves $u_n \Rightarrow \dots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)) = 1$. (Note that by instantiating $u_i := [\beta_i]_{\mathsf{PV}}$ for $i \in [n]$ we can derive the lemma.)

Hint: Introduce a provable equation, e.g., $\varepsilon = \varepsilon$ in the antecedent using cut.

Base case. We first consider the case $n = 1$. The case for $\Gamma = \varnothing$ can be derived from this case, which is left as an exercise. Note that we need to prove that if PV proves for any PV term u,

- $\mathsf{IsNotEps}(u) = 1$,
- $u \Rightarrow \mathsf{EQ}(f_1(\varepsilon, x_2), f_2(\varepsilon, x_2)) = 1$,
- $u \Rightarrow \mathsf{EQ}(f_1(x_1, \varepsilon), f_2(x_1, \varepsilon)) = 1$,
- $u \Rightarrow \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)) \Rightarrow \mathsf{EQ}(f_1(s_{i_1}(x_1), s_{i_2}(x_2)), f_2(s_{i_1}(x_1), s_{i_2}(x_2))) = 1$, for any $i_1, i_2 \in \{0, 1\}$,

then $\mathsf{PV} \vdash u \Rightarrow \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)) = 1$. That is:

$$\mathsf{ITE}(u, 1, \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2))) = \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)). \tag{3.11}$$

We will prove the alternative equation:

$$\mathsf{ITE}(u, \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)), 1) = 1 \tag{3.12}$$

Hint: Prove a more general conditional equation and apply Modus Ponens.

in PV, and it is left as an exercise to prove that this suffices.

We will prove the equation by simultaneous induction on x_1 and x_2 using Theorem 2.6.3. Let $g_{00} = g_{01}(x) = g_{10}(y) = 1$, and $h_{i_1 i_2}(x, y, z) = \mathsf{ITE}(z, 1, 0)$ for $i_1, i_2 \in \{0, 1\}$. It is easy to see that the RHS of $(\star)$ is identical to the function defined recursively from $g_{j_1 j_2}$ and $h_{i_1 i_2}$. Moreover, we can easily see that PV proves:

- $\mathsf{ITE}(u, \mathsf{EQ}(f_1(\varepsilon, \varepsilon), f_2(\varepsilon, \varepsilon)), 1) = g_{00} = 1$,
- $\mathsf{ITE}(u, \mathsf{EQ}(f_1(\varepsilon, s_j(x_2)), f_2(\varepsilon, s_j(x_2))), 1) = g_{01}(s_j(x_2)) = 1$,
- $\mathsf{ITE}(u, \mathsf{EQ}(f_1(s_j(x_1), \varepsilon), f_2(s_j(x_1), \varepsilon)), 1) = g_{01}(s_j(x_1)) = 1$,

from the assumption. Therefore, it suffices to prove that

$$\begin{aligned} &\mathsf{ITE}(u, \mathsf{EQ}(f_1(s_{i_1}(x_1), s_{i_2}(x_2)), f_2(s_{i_1}(x_1), s_{i_2}(x_2))), 1) \\ &= \mathsf{ITE}(\mathsf{ITE}(u, \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)), 1), 1, 0). \end{aligned} \tag{3.13}$$

We will first prove a more general result:

Proposition 3.7.2. $\mathsf{PV} \vdash (x \Rightarrow y \Rightarrow z = 1) \Rightarrow \mathsf{ITE}(x, z, 1) = \mathsf{ITE}(\mathsf{ITE}(x, y, 1), 1, 0)$.

This can be proved by a simple case study on x, y, and z using Theorem 2.4.5. Finally, with the substitution

$$\begin{aligned} &x/u \\ &y/\mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)) \\ &z/\mathsf{EQ}(f_1(s_{i_1}(x_1), s_{i_2}(x_2)), f_2(s_{i_1}(x_1), s_{i_2}(x_2))) \end{aligned}$$

we can see that the antecedent of Proposition 3.7.2 is exactly the second bullet of the assumption, and the consequence is exactly Equation (3.13) that we need to prove. This concludes the base case by Modus Ponens (see Proposition 2.4.10).

Induction case. Our strategy is to "merge" u_{n+1} and u_n in the antecedents of the conditional equation so that it reduces to the case for $|\Gamma| = n$. Concretely, we will prove the following meta-theorem of PV:

Proposition 3.7.3. *Let t_1, t_2, t_c, t'_c be terms such that $\mathsf{PV} \vdash \mathsf{IsNotEps}(t_c) = 1$ and $\mathsf{PV} \vdash \mathsf{IsNotEps}(t'_c) = 1$. Then $\mathsf{PV} \vdash t_c \Rightarrow t'_c \Rightarrow t_1 = t_2$ if and only if $\mathsf{PV} \vdash \mathsf{And}(t_c, t'_c) \Rightarrow t_1 = t_2$.*

To see that this suffices, notice that by applying the meta-theorem on both the premise and the conclusion of the rule, it suffices to derive from

$$\begin{aligned}
&\mathsf{PV} \vdash \mathsf{IsNotEps}(u_i) = 1 \quad (i \in [n-1]) \\
&\mathsf{PV} \vdash \mathsf{IsNotEps}(\mathsf{And}(u_{n+1}, u_n)) = 1 \\
&\mathsf{PV} \vdash \mathsf{And}(u_{n+1}, u_n) \Rightarrow u_{n-1} \Rightarrow \ldots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(\varepsilon, x_2), f_2(\varepsilon, x_2)) = 1 \\
&\mathsf{PV} \vdash \mathsf{And}(u_{n+1}, u_n) \Rightarrow u_{n-1} \Rightarrow \ldots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(x_1, \varepsilon), f_2(x_1, \varepsilon)) = 1 \\
&\mathsf{PV} \vdash \mathsf{And}(u_{n+1}, u_n) \Rightarrow u_{n-1} \Rightarrow \ldots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)) \Rightarrow \\
&\qquad \mathsf{EQ}(f_1(s_{i_1}(x_1), s_{i_2}(x_2)), f_2(s_{i_1}(x_1), s_{i_2}(x_2))) = 1 \quad (i_1, i_2 \in \{0, 1\})
\end{aligned}$$

that $\mathsf{PV} \vdash \mathsf{And}(u_{n+1}, u_n) \Rightarrow u_{n-1} \Rightarrow \ldots \Rightarrow u_1 \Rightarrow \mathsf{EQ}(f_1(x_1, x_2), f_2(x_1, x_2)) = 1$, which is true by the induction hypothesis and the fact that if $\mathsf{PV} \vdash \mathsf{IsNotEps}(u) = 1$ and $\mathsf{PV} \vdash \mathsf{IsNotEps}(u') = 1$, then $\mathsf{PV} \vdash \mathsf{IsNotEps}(\mathsf{And}(u, u')) = 1$.

It remains to prove Proposition 3.7.3. We will only prove the $(\Rightarrow)$ directly, and the converse can be proved using the same approach. Consider the following more general version:

Proposition 3.7.4. $\mathsf{PV} \vdash \mathsf{IsNotEps}(z_1) \Rightarrow \mathsf{IsNotEps}(z_2) \Rightarrow (z_1 \Rightarrow z_2 \Rightarrow x = y) \Rightarrow \mathsf{And}(z_1, z_2) \Rightarrow x = y$.

This can be proved by a case study on z_1 and z_2 using Theorem 2.4.5. The cases for z_1/ε, z_2/ε, $z_1/s_0(z_1)$, and $z_2/s_0(z_2)$ can be proved by the Explosion Rule (see Proposition 2.4.11). For the case $z_1/s_1(z_1)$ and $z_2/s_1(z_2)$, we can unfold all the definitions and apply the correctness of EQ (see Lemma 2.7.1).

To see that this implies the $(\Rightarrow)$ direction of Proposition 3.7.3, we can substitute z_1/t_c, z_2/t'_c, x/t_1, y/t_2. We can remove the outermost three antecedents using the assumption and Modus Ponens (see Proposition 2.4.10), which derives $\mathsf{And}(t_c, t'_c) \Rightarrow t_1 = t_2$ and completes the proof. □

3.8 Extensions of Rules

Now we show a few extensions of the rules in PV-PL that may be helpful in formalizing informal mathematical proofs.

Rewrite of formulas. We will also consider a stronger form of $(=_/)$, denoted by $(\hat{=}_/)$, together with its converse $(\hat{=}_/^{-1})$. Both of the rules will be referred as "rewrite rules" as it can be used to rewrite a formula based on an equation in the antecedent.

Let $\Gamma, x = y \vdash \varphi$ be an assertion. Suppose that φ' is obtained from φ by replacing *some* occurrences of x to y, then the axioms $(\hat{=}_{/})$ and its converse $(\hat{=}_{/}^{-1})$ are defined as

$$(\hat{=}_{/}) : \quad \frac{}{\Gamma, x = y, \varphi \vdash \varphi'}; \quad (\hat{=}_{/}^{-1}) : \quad \frac{}{\Gamma, x = y, \varphi' \vdash \varphi}.$$

We call φ the substitution formula of the rule.

Note that $(\hat{=}_{/})$ and its converse simulate each other by the cut rule and the symmetricity of equality; the following proof tree simulates $(\hat{=}_{/}^{-1})$ using $(\hat{=}_{/})$:

$$\frac{\dfrac{}{\Gamma, x = y \vdash y = x}\,(=_s) \qquad \dfrac{}{\Gamma, x = y, y = x, \varphi' \vdash \varphi}\,(\hat{=}_{/})}{\Gamma, x = y, \varphi' \vdash \varphi}\,(\mathrm{Cut})$$

(All structural rules are omitted.)

Lemma 3.8.1. *$(\hat{=}_{/})$ and $(\hat{=}_{/}^{-1})$ are admissible in PV-PL.*

Proof Sketch. We prove by induction on the depth of the substitution formula φ.

Base case. Suppose that φ is atomic, i.e., it is either $\bot$ or a PV equation $u = v$. Note that the case for $\varphi \equiv \bot$ is trivial by the assumption rule (A). Consider the case for $\varphi \equiv$ "$u = v$" and $\varphi' \equiv$ "$u' = v'$", and it suffices to prove the admissibility of $(\hat{=}_{/})$ as $(\hat{=}_{/}^{-1})$ can be simulated by $(\hat{=}_{/})$ with substitution formula of the same depth. Let z be a fresh variable and $\hat{u}$ and $\hat{v}$ be the terms satisfying that

$$\hat{u}[z/y] = u', \hat{u}[z/x] = u, \ \hat{v}[z/y] = v', \hat{v}[z/x] = v.$$

Notice that we can prove $\Gamma, x = y, u = v \vdash u = u'$ by the proof tree (where all structural rules are omitted):

$$\frac{}{\Gamma, x = y, u = v \vdash \hat{u}[z/x] = \hat{u}[z/y]}\,(=_{/})$$

Similarly, we can prove that $\Gamma, x = y, u = v \vdash v = v'$. Therefore by the cut rule and the symmetricity and transitivity of equality, we can prove $\Gamma, x = y, u = v \vdash u' = v'$.

Induction case. Suppose that φ is of form $\beta \to \alpha$ and $\varphi' \equiv \beta' \to \alpha'$. By the induction hypothesis, we know that both $(\hat{=}_{/})$ and $(\hat{=}_{/}^{-1})$ are admissible if the substitution formula is α, α', β or β'. It suffices to prove the admissibility of $(\hat{=}_{s})$ with φ being the substitution formula, i.e.,

$$\Gamma, x = y, \beta \to \alpha \vdash \beta' \to \alpha'.$$

By the introduction rule of implication, it suffices to prove that $\Gamma, x = y, \beta \to \alpha, \beta' \vdash \alpha'$. We then apply the cut rule with β being the cut formula, so that it generates two subgoals to resolve:

(*Subgoal* 1): $\Gamma, x = y, \beta \to \alpha, \beta' \vdash \beta$. This is provable by the induction hypothesis, as we can apply $(\hat{=}_{/}^{-1})$ with β' being the substitution formula.

(*Subgoal* 2): $\Gamma, x = y, \beta \to \alpha, \beta', \beta \vdash \alpha'$. We apply the cut rule again with α being the cut formula, so that it further generates two subgoals:

(*Subgoal 2.1*): $\Gamma, x = y, \beta \to \alpha, \beta', \beta \vdash \alpha$. Notice that this can be proved by Modus Ponens on two assumptions, which can be implemented by the elimination rule of $\to$:

This is usually called the `apply` tactic in proof assistants.

$$\dfrac{\dfrac{}{\Gamma, x = y, \beta \to \alpha, \beta', \beta \vdash \beta \to \alpha}\,(\mathrm{A}) \qquad \dfrac{}{\Gamma, x = y, \beta \to \alpha, \beta', \beta \vdash \beta}\,(\mathrm{A})}{\Gamma, x = y, \beta \to \alpha, \beta', \beta \vdash \alpha}\,(\to_e)$$

(*Subgoal 2.2*): $\Gamma, x = y, \beta \to \alpha, \beta', \beta, \alpha \vdash \alpha'$. This is provable by the induction hypothesis, as we can apply $(\hat{=}_{/})$ with α being the substitution formula.

This completes all subgoals and leads to the lemma. □

Comprehension of PV translation. Recall that the PV translation could translate a formula φ in $\mathsf{PV\text{-}PL}$ into a PV term $[\varphi]_{\mathsf{PV}}$. We now show that we can prove the equivalence of a formula and its PV translation in $\mathsf{PV\text{-}PL}$, formalized as the following comprehension axiom (CPV) and its converse (C$^{-1}\mathsf{PV}$):

$$(\mathrm{C}\mathsf{PV}):\ \dfrac{}{\Gamma, \varphi \vdash [\varphi]_{\mathsf{PV}} = 1}; \qquad (\mathrm{C}^{-1}\mathsf{PV}):\ \dfrac{}{\Gamma, [\varphi]_{\mathsf{PV}} = 1 \vdash \varphi}.$$

Lemma 3.8.2. *Both* (CPV) *and* (C$^{-1}\mathsf{PV}$) *are admissible in* $\mathsf{PV\text{-}PL}$.

Proof Sketch. We prove this by induction on the depth of the formula φ in the comprehension axiom. The case for $\varphi = \bot$ is trivial. Suppose that φ is of form $u = v$, then (CPV) follows from the rewrite rule as well as $\mathsf{PV} \vdash \mathsf{EQ}(x, x) = 1$, while (C$^{-1}\mathsf{PV}$) follows from the correctness of EQ as demonstrated in Example 3.2.2.

Now we consider the case that φ is of form $\beta \to \alpha$, so that it suffices to prove in $\mathsf{PV\text{-}PL}$ that:

- $\Gamma, \beta \to \alpha \vdash \mathsf{ITE}([\beta]_{\mathsf{PV}}, [\alpha]_{\mathsf{PV}}, 1) = 1$,
- $\Gamma, \mathsf{ITE}([\beta]_{\mathsf{PV}}, [\alpha]_{\mathsf{PV}}, 1) = 1 \vdash \beta \to \alpha$.

The proofs of these two assertions are similar, so we only explain the first bullet.

By the cut rule, we first prove that $\Gamma, \beta \to \alpha \vdash [\beta]_{\mathsf{PV}} = 1 \to [\alpha]_{\mathsf{PV}} = 1$, which reduces to $\Gamma, \beta \to \alpha, [\beta]_{\mathsf{PV}} = 1 \vdash [\alpha]_{\mathsf{PV}} = 1$. By the induction hypothesis, we know that $\Gamma, \beta \to \alpha, [\beta]_{\mathsf{PV}} = 1 \vdash \beta$ and subsequently

$$\Gamma, \beta \to \alpha, [\beta]_{\mathsf{PV}} = 1 \vdash \alpha.$$

Using the cut rule it suffices to prove that $\Gamma, \beta \to \alpha, \alpha, [\beta]_{\mathsf{PV}} = 1 \vdash [\alpha]_{\mathsf{PV}}$, which again follows from the induction hypothesis. □

Induction on formulas. One primary application of the comprehension axioms is to generalize the induction principle (Ind$_n$) from equations to formulas. Suppose that φ is a formula and z is a variable, we have

$$(\mathrm{Ind}_1^{\varphi}):\ \dfrac{\Gamma \vdash \varphi[z/\varepsilon] \quad \Gamma, \varphi \vdash \varphi[z/s_0(z)] \quad \Gamma, \varphi \vdash \varphi[z/s_1(z)]}{\Gamma \vdash \varphi}$$

Lemma 3.8.3. $(\mathrm{Ind}_1^{\varphi})$ *is admissible in* PV-PL.

Proof Sketch. By the cut rule and the comprehension axiom, it suffices to prove that $\Gamma \vdash [\varphi]_{\mathsf{PV}} = 1$ from the premises. By the induction rule (Ind_1) in PV-PL, it suffices to prove that

- $\Gamma \vdash [\varphi]_{\mathsf{PV}}[z/\varepsilon] = 1$
- $\Gamma, [\varphi]_{\mathsf{PV}} = 1 \vdash [\varphi]_{\mathsf{PV}}[z/s_i(z)] = 1$ for $i \in \{0, 1\}$.

It is easy to see that substitution commutes with the PV translation, and therefore it is equivalent to proving that

- $\Gamma \vdash [\varphi[z/\varepsilon]]_{\mathsf{PV}} = 1$,
- $\Gamma, [\varphi]_{\mathsf{PV}} = 1 \vdash [\varphi[z/s_i(z)]]_{\mathsf{PV}} = 1$ for $i \in \{0, 1\}$,

which can be proved from the premises using the comprehension axioms. □

Remark 3.8.1. Though we only wrote down the induction principle with one variable for simplicity of presentation, the same proof clearly generalizes to the case for multiple variables.

3.9 Bibliographical and Other Remarks

The theory PV-PL is almost identical to the theory PV_1 that Cook introduced in [Coo75]. We use the name PV-PL as nowadays the name PV_1 usually refers to the first-order extension of PV (see, e.g., [Kra95a, KPT91, Kra19]). The theory PV-PL can be viewed as the universal fragment of PV_1, and thus by the subformula property (see, e.g., [TS00, Section 4.2]), PV_1 is a conservative extension of PV-PL.

The theory PV-PL is defined by a natural deduction system, where the consequences of assertions are formulas. Alternatively, one can define the theory using a sequent calculus system (similar to Buss's theory T_2^0 and S_2^1 [Bus86]) where the consequences of assertions are sequences of formulas. We choose the natural deduction formulation as it is closer to human reasoning.

Another proof of Theorem 3.3.1 is to apply Buss's witnessing theorem [Bus86], which relies on a form of Gentzen's cut elimination theorem (see, e.g., [TS00, Chapter 4]) for the theory S_2^1.

As far as we know, the direct embedding of PV-PL assertions to PV equation and the translation theorem (see Theorem 3.3.4) is not explicitly stated in literature. Arguably, the proof is more straightforward than proofs using the cut elimination theorem. It might be worth noting that the direct translation proof avoids the super-polynomial proof size blowup of the cut elimination theorem (see, e.g., [Ngu07] and [TS00, Chapter 5]). This could be an advantage if the sizes of the PV-PL and PV proofs are important.

Chapter 4

Advanced Tools in PV

With the stronger reasoning system PV-PL and the translation theorem (see Theorem 3.3.4), we continue the investigation of the programming language built from Cook's theory PV.

In this chapter, we will develop advanced programming and mathematical tools, which significantly extend our capability of programming and reasoning about programs. This provides further evidence for the Feasible Mathematics Thesis. In more detail, we will provide:

1. Constructions of advanced data structures such as lists and maps with PV (or PV-PL) provable correctness. In addition, we will prove meta-theorems that allow recursion and induction on lists just like the limited recursion and induction rules over strings.

2. A simple imperative programming language IMP(PV) and a proof system for its functionality based on Hoare logic. We prove that IMP(PV) programs can be compiled back to PV functions, and proofs about the functionality of programs using Hoare logic can be compiled back to PV proofs.

3. A simulation of Turing machines by IMP(PV) programs (and subsequently by PV functions). This concludes Theorems 2.1.1 and 2.3.2.

4. A development of *feasible set theory*, which allows set-theoretic operations (e.g. union and intersection) on finite sets encoded by enumeration. We show that proofs in the first-order logic over finite sets (i.e. allowing quantifiers $\forall x \in S$ and $\exists x \in S$) can be translated back to PV proofs.

Throughout this chapter, we will write "pseudo-proofs", i.e., goal-targeted mathematical proofs in natural language that could be directly translated to a proof tree in PV-PL, and subsequently a proof in PV by the translation theorem.

4.1 Lists

Next, we implement *lists*, i.e., a sequence of unbounded length. The list is arguably the most important data structure in *functional programming* and will greatly simplify the formalization of algorithms in PV.

4.1.1 Definition of Lists

We first implement basic functions to form and unfold a list: $\mathsf{Append}(\ell, x)$ (also denoted by $\ell :: x$) appends a string x to the end of a list ℓ, $\mathsf{Head}(\ell)$, $\mathsf{Tail}(\ell)$ and proves in PV that[1]

$$\mathsf{Head}(\varepsilon) = \varepsilon, \qquad \mathsf{Head}(\ell :: x) = x \tag{4.1}$$

$$\mathsf{Tail}(\varepsilon) = \varepsilon, \qquad \mathsf{Tail}(\ell :: x) = \ell \tag{4.2}$$

Also, we will prove that the description length of a list does not grow super-polynomially so that we can create a list of polynomial length. Formally:

$$\mathsf{PV} \vdash \mathsf{ITR}(\ell :: x, \ell \circ 11 \circ \mathsf{PEnc}(x)) = \varepsilon. \tag{4.3}$$

(We will see how this equation will be useful later.)

Indeed the construction is similar to tuples. We define:

$$\mathsf{Append}(\ell, x) := \mathsf{MakePair}(\ell, x),\ \mathsf{Head}(\ell) = \mathsf{Right}(\ell),\ \mathsf{Tail}(\ell) = \mathsf{Left}(\ell).$$

The proof of Equation (4.1) and (4.2) naturally follows from Lemma 2.5.4. Therefore, it suffices to verify Equation (4.3):

Lemma 4.1.1. $\mathsf{PV} \vdash \mathsf{ITR}(\ell :: x, \ell \circ (11 \circ \mathsf{PEnc}(x))) = \varepsilon$.

Proof. By unfolding the definition of Append (i.e. $\ell :: x$), subsequently unfolding MakePair, and applying the associativity of concatenation (see Proposition 2.1.4), it suffices to prove that

$$\mathsf{PV} \vdash \mathsf{ITR}(\ell \circ 11 \circ \mathsf{PEnc}(x), \ell \circ 11 \circ \mathsf{PEnc}(x)) = \varepsilon. \tag{4.4}$$

This follows from $\mathsf{PV} \vdash \mathsf{ITR}(x, x) = \varepsilon$ using (L4) substitution, and $\mathsf{ITR}(x, x) = \varepsilon$ can be proved in PV using Proposition 2.4.1 and Proposition 2.1.3. □

Generating a list. As an example, we will define a function $\mathsf{GenList}(v, y)$ that generates a list of length $|y|$ consists of v. Formally, it will satisfy that

$$\mathsf{PV} \vdash \mathsf{GenList}(v, \varepsilon) = \varepsilon \tag{4.5}$$

$$\mathsf{PV} \vdash \mathsf{GenList}(v, s_i(y)) = \mathsf{GenList}(v, y) :: v \tag{4.6}$$

We will define the function using limited recursion on the variable y. Let $g(v) = \varepsilon$, $h_i(v, y, z) = z :: v$, and $k_i(v, y) = 11 \circ \mathsf{PEnc}(v)$, we can verify that

Proposition 4.1.2. $\mathsf{PV} \vdash \mathsf{ITR}(h_i(v, y, z), z \circ k_i(v, y)) = \varepsilon$ *for* $i \in \{0, 1\}$.

Proof. Unfolding h_i, k_i, we will notice that this is exactly Lemma 4.1.1. □

[1] Here we encode the empty list using ε; notice that the empty list ε is different from a single-element list consisting of ε (i.e. $\varepsilon :: \varepsilon$).

Therefore, we can define a function f from limited recursion rule using the functions (g, h_0, h_1, k_0, k_1). It is easy to verify that if we define $\mathsf{GenList}(v, y) := f(v, y)$, it will satisfy Equation (4.5) and (4.6).

We also note that by composing $\mathsf{GenList}$ and $\#$, we can initiate a list of arbitrary polynomial lengths. For instance, $\mathsf{GenList}(v, y \# y \# y)$ will generate a list of length $|y|^3$ consisting of v.

4.1.2 Recursion on Lists

Now, we state a meta-theorem that allows us to define functions recursively over lists. Using this tool, we can define useful functions over lists, such as the length of a list and the concatenation of lists.

Theorem 4.1.3 (Recursion on Lists)**.** *Let $g(\vec{x})$, $h(\vec{x}, u, \ell, z)$, $k(\vec{x}, u, \ell)$ be* PV *functions satisfying that*

$$\mathsf{PV} \vdash \mathsf{ITR}(h(\vec{x}, u, \ell, z), z \circ k(\vec{x}, u, \ell)) = \varepsilon.$$

Then there is a PV *function $f(\vec{x}, \ell)$ such that* PV *proves:*

$$f(\vec{x}, \varepsilon) = g(\vec{x}) \tag{4.7}$$

$$f(\vec{x}, \ell :: u) = h(\vec{x}, u, \ell, f(\vec{x}, \ell)) \tag{4.8}$$

We defer the proof of the theorem to the end of this chapter as it is quite technical. Here we will first define some functions on lists using this meta-theorem. Note that the existence of the length upper bound k in all the examples below is obvious, and is left as an exercise.

Type-checking. We can define a function using Theorem 4.1.3 that performs type-checking on lists. Let $g = 1$ and

$$h(u, \ell, z) = \mathsf{And}(\mathsf{EQ}(\ell, \mathsf{Tail}(\ell) :: \mathsf{Head}(\ell)), z),$$

we can apply Theorem 4.1.3 to define a function $\mathsf{IsList}'(\ell)$ such that

$$\mathsf{PV} \vdash \mathsf{IsList}'(\varepsilon) = 1 \tag{4.9}$$

$$\mathsf{PV} \vdash \mathsf{IsList}'(\ell :: u) = \mathsf{And}(\mathsf{Or}(\mathsf{IsEps}(\ell), \mathsf{EQ}(\ell, \mathsf{Tail}(\ell) :: \mathsf{Head}(\ell))), \mathsf{IsList}'(\ell)). \tag{4.10}$$

We can define $\mathsf{IsList}(\ell) = \mathsf{And}(\mathsf{Or}(\mathsf{IsEps}(\ell), \mathsf{EQ}(\ell, \mathsf{Tail}(\ell) :: \mathsf{Head}(\ell))), \mathsf{IsList}'(\ell))$.

Proposition 4.1.4. $\mathsf{PV}\text{-}\mathsf{PL}$ *proves* $\vdash \mathsf{IsList}(\varepsilon) = 1$ *and* $\mathsf{IsList}(\ell) = 1 \vdash \mathsf{IsList}(\ell :: u) = 1$.

Proof Sketch. Both assertions can be proved by simple unfolding. □

Length. Similarly, we can define a function using Theorem 4.1.3 that outputs the length of a list in unary. Concretely:

$$\mathsf{PV} \vdash \mathsf{Len}(\varepsilon) = 0 \tag{4.11}$$

$$\mathsf{PV} \vdash \mathsf{Len}(\ell :: u) = s_0(\mathsf{Len}(\ell)) \tag{4.12}$$

Concatenation. We can also define the concatenation of two lists ℓ_1 and ℓ_2 by recursion on ℓ_2. Concretely:

$$\mathsf{PV} \vdash \mathsf{Concat}(\ell_1, \varepsilon) = \ell_1 \tag{4.13}$$

$$\mathsf{PV} \vdash \mathsf{Concat}(\ell_1, \ell_2 :: u) = \mathsf{Concat}(\ell_1, \ell_2) :: u \tag{4.14}$$

For simplicity, we will use $\ell_1 \mathbin{+\!+} \ell_2$ to denote $\mathsf{Concat}(\ell_1, \ell_2)$.

Map. We can define a *functional*[2] Map_f that produces a function given any PV function symbol $f(x, \vec{w})$. Intuitively, $\mathsf{Map}_f(\ell)$ will apply the function f to each element in ℓ and return the new list. That is:

$$\begin{aligned} &\mathsf{PV} \vdash \mathsf{Map}_f(\varepsilon) = \varepsilon \\ &\mathsf{PV} \vdash \mathsf{Map}_f(\ell :: u) = \mathsf{Map}_f(\ell) :: f(u, \vec{w}) \end{aligned}$$

Find. Another functional that will be particularly important in our development of other data structures is to find the rightmost element in a list satisfying a given property. Let $f(x, \vec{w})$ be a PV function symbol, $\mathsf{Find}_f(\ell)$ will satisfy that

$$\begin{aligned} &\mathsf{PV} \vdash \mathsf{Find}_f(\varepsilon) = \varepsilon \\ &\mathsf{PV} \vdash \mathsf{Find}_f(\ell :: u) = \mathsf{ITE}(f(u, \vec{w}), s_0(u), \mathsf{Find}_f(\ell)) \end{aligned}$$

(Note that we output $s_0(u)$ instead of u as otherwise we cannot distinguish between the case that $\mathsf{Find}_f(\ell)$ does not find the element, or find the element $\varepsilon \in \ell$ satisfying that $f(\varepsilon, \vec{w})$.)

In particular, we can define the membership query of an element in a list as $\mathsf{Find}_x(\ell) := \mathsf{IsNotEps}(\mathsf{Find}_{f_x}(\ell))$ with $f_x(u) = \mathsf{EQ}(x, u)$. For simplicity, we write $[x \in \ell]$ as the abbreviation of $\mathsf{Find}_x(\ell) = 1$. It is left as an exercise that $\mathsf{PV}\text{-}\mathsf{PL}$ proves:

$$\begin{aligned} &\vdash \neg[x \in \varepsilon] \\ &\vdash [x \in \ell :: x] \\ [x \in \ell] &\vdash [x \in \ell :: u] \end{aligned}$$

This is certainly not a complete list of functions we can define using Theorem 4.1.3. The fun puzzle of defining other functions on lists is left to the readers.

4.1.3 Induction on Lists

We will then prove an induction principle on lists that as an analogy of the induction rule (Ind_n) in $\mathsf{PV}\text{-}\mathsf{PL}$. For simplicity, we will only consider induction on one list:

Theorem 4.1.5 (Inducition on Lists)**.** *Let u be a variable with no occurrences in Γ, t_1, t_2, and ℓ be a variable with no occurrences in Γ. The following rule is admissible in* $\mathsf{PV}\text{-}\mathsf{PL}$*:*

$$\frac{\Gamma \vdash t_1[\ell/\varepsilon] = t_2[\ell/\varepsilon] \quad \Gamma, \mathsf{IsList}(\ell) = 1, t_1 = t_2 \vdash t_1[\ell/\ell :: u] = t_2[\ell/\ell :: u]}{\Gamma, \mathsf{IsList}(\ell) = 1 \vdash t_1 = t_2}$$

[2] Functional should be considered as a notation in the meta-theory, rather than a symbol in PV.

The proof of the theorem is deferred to Section 4.6.

Note that a stronger version of induction is admissible, namely induction on an arbitrary PV-PL formula:

Corollary 4.1.6. *Let u be a variable with no occurrences in Γ, t_1, t_2, and ℓ be a variable with no occurrences in Γ. The following induction rule for lists is admissible in* PV-PL*:*

$$\frac{\Gamma \vdash \varphi[\ell/\varepsilon] \quad \Gamma, \mathsf{IsList}(\ell) = 1, \varphi \vdash \varphi[\ell/\ell :: u]}{\Gamma, \mathsf{IsList}(\ell) = 1 \vdash \varphi}$$

The proof utilizes the admissibility of the comprehension axioms (see Lemma 3.8.2) and is similar to that of Lemma 3.8.3, which is left as an exercise.

Example 4.1.1. We now show how to prove the transitivity of concatenation of lists using Corollary 4.1.6. Our goal is to prove in PV-PL that:

$$\mathsf{IsList}(\ell_3) = 1 \vdash (\ell_1 \mathbin{++} \ell_2) \mathbin{++} \ell_3 = \ell_1 \mathbin{++}(\ell_2 \mathbin{++} \ell_3).$$

We prove this by induction on ℓ_3 using Corollary 4.1.6, which generalizes the following two subgoals.

(*Subgoal 1*). $\vdash (\ell_1 \mathbin{++} \ell_2) \mathbin{++} \varepsilon = \ell_1 \mathbin{++}(\ell_2 \mathbin{++} \varepsilon)$. This can be easily proved by the definition equation of ++.

(*Subgoal 2*). We need to prove that

$$\begin{aligned} &\mathsf{IsList}(\ell_3) = 1, (\ell_1 \mathbin{++} \ell_2) \mathbin{++} \ell_3 = \ell_1 \mathbin{++}(\ell_2 \mathbin{++} \ell_3) \\ \vdash\ &(\ell_1 \mathbin{++} \ell_2) \mathbin{++}(\ell_3 :: u) = \ell_1 \mathbin{++}(\ell_2 \mathbin{++}(\ell_3 :: u)) \end{aligned}$$

By the definition equations of ++ (see Equation (4.13) and (4.14)), we can prove that the LHS of the consequence is equal to $((\ell_1 \mathbin{++} \ell_2) \mathbin{++} \ell_3) :: u$, which is subsequently equal to $(\ell_1 \mathbin{++}(\ell_2 \mathbin{++} \ell_3)) :: u$ by rewriting using the assumption that $(\ell_1 \mathbin{++} \ell_2) \mathbin{++} \ell_3 = \ell_1 \mathbin{++}(\ell_2 \mathbin{++} \ell_3)$ (see Lemma 3.8.1).

Similarly, we can prove by the definition equations of ++ that the RHS of the consequence is equal to $\ell_1 \mathbin{++}((\ell_2 \mathbin{++} \ell_3) :: u)$, and is subsequently equal to $(\ell_1 \mathbin{++}(\ell_2 \mathbin{++} \ell_3)) :: u$. Therefore, we can prove

$$(\ell_1 \mathbin{++} \ell_2) \mathbin{++}(\ell_3 :: u) = \ell_1 \mathbin{++}(\ell_2 \mathbin{++}(\ell_3 :: u))$$

by the symmetricity and transitivity of equality. This completes the proof.

4.1.4 Dictionaries

Building on lists, we will implement *dictionaries*, i.e., the data structure maintaining key-value relation that can be updated. In particular, we can use a dictionary to simulate an array of length ℓ by fixing the keys to be $[\ell]$.

Definition of dictionaries. We will implement a dictionary using a list containing a list of pairs maintaining key-value relations. Concretely, we define the functions

$\mathsf{Lookup}(\gamma, x)$ and $\mathsf{Update}(\gamma, x, y)$ as follows:

$$\mathsf{Lookup}(\gamma, x) := \mathsf{Right}(\mathsf{TR}(\mathsf{Find}_f(\gamma))), \quad f(\tau, x) = \mathsf{EQ}(\mathsf{Left}(\tau), x); \tag{4.15}$$

$$\mathsf{Update}(\gamma, x, y) := \gamma :: \mathsf{MakePair}(x, y). \tag{4.16}$$

Intuitively, $\mathsf{Lookup}(\gamma, x)$ finds the value corresponding to the key x by searching for the rightmost (i.e. latest) pair of the form $(x, \cdot)$, and $\mathsf{Update}(\gamma, x, y)$ adds a new key-value relation (x, y) to the dictionary γ. Note that we encode an empty dictionary using an empty list ε.

Correctness. We will prove the correctness of dictionaries formalized by the following PV-PL assertions:

$$\Gamma \vdash \mathsf{Lookup}(\varepsilon, x) = \varepsilon; \tag{4.17}$$

$$\Gamma, \mathsf{Lookup}(\gamma, x) = y, x' \neq x \vdash \mathsf{Lookup}(\mathsf{Update}(\gamma, x', y'), x) = y; \tag{4.18}$$

$$\Gamma \vdash \mathsf{Lookup}(\mathsf{Update}(\gamma, x, y), x) = y. \tag{4.19}$$

All these three assertions can be proved in PV-PL by simply unfolding all the functions; here, we will demonstrate the proof of Equation (4.18), and the rest of the proof is left as an exercise.

Proposition 4.1.7. PV-PL *proves Equation* (4.18).

Proof. By unfolding Update, the consequence of the assertion can be changed to $\mathsf{Lookup}(\gamma :: \mathsf{MakePair}(x', y')) = y$, and by further unfolding Lookup and Find_f, it suffices to prove from $\Gamma, \mathsf{Lookup}(\gamma, x) = y, x' \neq x$ that

$$\mathsf{Right}(\mathsf{TR}(\mathsf{ITE}(\mathsf{EQ}(\mathsf{Left}(\mathsf{MakePair}(x', y')), x), s_0(\mathsf{MakePair}(x', y')), \mathsf{Find}_f(\gamma))) = y,$$

where $f(\tau, x) := \mathsf{EQ}(\mathsf{Left}(\tau), x)$. By applying the correctness of Left and MakePair (using the cut rule), it suffices to prove (from the same antecedent) that

$$\mathsf{Right}(\mathsf{TR}(\mathsf{ITE}(\mathsf{EQ}(x', x), s_0(\mathsf{MakePair}(x', y')), \mathsf{Find}_f(\gamma)))) = y.$$

Note that by the correctness of EQ (see, e.g., Example 3.2.2) in PV-PL and the property $x' \neq x$ in the antecedent, we can conclude that $\mathsf{EQ}(x', x) = 0$, and therefore by the cut rule, it suffices to prove the equation above from the antecedent $\Gamma, \mathsf{Lookup}(\gamma, x) = y, x' \neq x, \mathsf{EQ}(x', x) = 0$. Using the substitution/generalization rule, it suffices to prove

$$\mathsf{Right}(\mathsf{TR}(\mathsf{ITE}(z, s_0(\mathsf{MakePair}(x', y')), \mathsf{Find}_f(\gamma)))) = y$$

from $\Gamma, \mathsf{Lookup}(\gamma, x) = y, x' \neq x, z = 0$, and subsequently we can change the consequence to

$$\mathsf{Right}(\mathsf{TR}(\mathsf{ITE}(0, s_0(\mathsf{MakePair}(x', y')), \mathsf{Find}_f(\gamma)))) = y$$

using the rewriting rule (see Lemma 3.8.1). Therefore, by unfolding ITE, it suffices to prove that $\mathsf{Right}(\mathsf{TR}(\mathsf{Find}_f(\gamma))) = y$, which, by the definition equation of Lookup (see Equation (4.15)) is provably identical to $\mathsf{Lookup}(\gamma, x) = y$. This is available in the assumption and thus concludes the proof. □

Type-checking. Similar to lists, we can define a function $\mathsf{IsDict}(\gamma)$ that verifies whether a string correctly encodes a dictionary. Concretely, given any string γ, our algorithm first checks whether γ is a list using $\mathsf{IsList}(\gamma)$. Let $f(x)$ be defined as

$$f(x) := \mathsf{Not}(\mathsf{EQ}(\mathsf{MakePair}(\mathsf{Left}(x), \mathsf{Right}(x)), x)),$$

our algorithm further checks whether $\mathsf{IsEps}(\mathsf{Find}_f(\gamma)) = 1$. The algorithm accepts if it passes both checks.

Proposition 4.1.8. PV-PL *proves the following assertions:*

- $\vdash \mathsf{IsDict}(\varepsilon) = 1$.
- $\mathsf{IsDict}(\gamma) = 1 \vdash \mathsf{IsDict}(\mathsf{Update}(\gamma, x, v)) = 1$.

4.2 Simulation of Imperative Programming Languages

We now demonstrate how to simulate ordinary imperative programming languages using functions in PV and reason about programs.

After all, most programmers rely on imperative programming languages in their minds, I guess...

4.2.1 Syntax of IMP(PV)

We introduce a simple untyped imperative programming language IMP(PV) that extends PV functions with variables and control statements. We first define the syntax of IMP(PV).

- (*Expressions*). An expression is an arbitrary PV term.
- (*Commands*). Commands in IMP(PV) is defined by the following BNF:

$$\begin{aligned}\mathsf{Cmd} := {} & \textbf{skip} \mid \textbf{let}\ x := \mathsf{Exp} \mid \mathsf{Cmd}; \mathsf{Cmd} \\ & \mid \textbf{if}\ \mathsf{Exp}\ \textbf{then}\ \mathsf{Cmd}\ \textbf{else}\ \mathsf{Cmd} \\ & \mid \textbf{for}\ x := \mathsf{Exp};\ x \neq \varepsilon;\ x := \mathsf{TR}(x)\ \textbf{do}\ \mathsf{Cmd}\end{aligned}$$

  where x denotes an arbitrary variable name.

For simplicity, we will assume that all variables are global variables (i.e. structural statements do not create namespaces). The semantics of an IMP(PV) in the standard model should be clear.

To ensure that the program terminates in polynomial time, we further introduce two restrictions to IMP(PV) programs:

- (Read-only restriction). The variable x used as the index variable of a for-loop statement is read-only within the loop, i.e., it is neither assigned using the **let** statement within the loop nor the index variable of an inner for-loop statement.
- (Length restriction). Let P be an IMP(PV) program satisfying the read-only restriction, and ω be a fresh variable, an *ω-length-restricted* program is of the form (P, ω), which intuitively means that all variables and expressions are subject to

the length upper bound ω. An expression that evaluates to a string y longer than ω will be rounded to $\mathsf{Suf}(y, \omega)$, i.e., the rightmost $|\omega|$-bits of y. (Nevertheless, we allow the intermediate computation within an expression to temporarily exceed the length restriction, as it will not cause any trouble.)

Arguably, these two restrictions should usually not trouble programmers: The read-only restriction is enforcing a good programming habit, and the length restriction also occurs in real-world programming languages due to the word length of built-in data types. I guess...?

4.2.2 Formal Semantic via Hoare Logic

To define the semantics of IMP(PV) and reason about programs, we introduce a Hoare logic for IMP(PV).

Let φ_1, φ_2 be PV-PL formulas and $c \in \mathsf{Cmd}$ be a command, a *Hoare tuple* is of the form $\{\varphi_1\}\ c\ \{\varphi_2\}$ indicating (intuitively) that if φ_1 is true before the execution of c, φ_2 is true after the execution of c. The formula φ_1 is called the *precondition*, and φ_2 is called the *postcondition*.

The semantic of IMP(PV) is given by the following axioms and rules:

- Consequence rule: Suppose that PV-PL proves $\varphi_1 \vdash \varphi_1'$ and $\varphi_2' \vdash \varphi_2$, then

$$\frac{\{\varphi_1'\}\ c\ \{\varphi_2'\}}{\{\varphi_1\}\ c\ \{\varphi_2\}}$$

  This rule is used to weaken the precondition and strengthen the postcondition.

- Empty statement axiom scheme:

$$\frac{}{\{\varphi\}\ \textbf{skip}\ \{\varphi\}}$$

- Assignment axiom scheme:

$$\frac{}{\{\varphi[x/e]\}\ \textbf{let}\ x := e\ \{\varphi\}}$$

- Rule of composition:

$$\frac{\{\varphi_1\}\ c_1\ \{\psi\} \qquad \{\psi\}\ c_2\ \{\varphi\}}{\{\varphi_1\}\ c_1; c_2\ \{\varphi_2\}}$$

- Rule of if-then-else:[3]

$$\frac{\{\varphi_1 \land \mathsf{LastBit}(e) = 1\}\ c_1\ \{\varphi_2\} \qquad \{\varphi_1 \land \mathsf{LastBit}(e) \neq 1\}\ c_0\ \{\varphi_2\}}{\{\varphi_1\}\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_0\ \{\varphi_2\}}$$

- Rule of for-loop: For $i \in \{0, 1\}$, we have

$$\frac{\{\varphi[x/s_i(x)]\}\ c\ \{\varphi\}}{\{\varphi[x/e]\}\ \textbf{for}\ x := e;\ x \neq \varepsilon;\ x := \mathsf{TR}(x)\ \textbf{do}\ c\ \{\varphi[x/\varepsilon]\}}$$

  This rule allows us to reason about the for-loop using a loop invariant φ maintained throughout the loop.

[3] $\varphi \land \varphi_2$ is indeed defined using connectives $\{\to, \neg\}$ in PV-PL.

We say that a Hoare tuple $\{\varphi_1\}\ c\ \{\varphi_2\}$ is provable if there is a proof tree using the axioms and rules above concluding $\{\varphi_1\}\ c\ \{\varphi_2\}$.

The rules of Hoare logic provide an intuitive reasoning framework on the behavior of $\mathsf{IMP}(\mathsf{PV})$ programs. The interpretation of the rules should be clear. We will use the following example to show the application of the Hoare rules.

Example 4.2.1. Consider the following $\mathsf{IMP}(\mathsf{PV})$ program for calculating the length of a list by iteratively removing the rightmost element:

```
LenITR := let $z := \ell$;
          let $y := \varepsilon$;
          for $x := \ell;\ x \neq \varepsilon;\ x := \mathsf{TR}(x)$ do
           if $\mathsf{IsEps}(z)$ then skip else let $y := s_0(y)$; let $z := \mathsf{Tail}(z)$
```

We can formalize its correctness as the Hoare tuple

$$\{\mathsf{IsList}(\ell) = 1\}\ \mathsf{LenITR}\ \{y = \mathsf{Len}(\ell)\}, \tag{4.20}$$

where Len is the function defined by Equation (4.11) and (4.12).

It can be proved using the loop invariant:

$$\varphi := \bigwedge \{\mathsf{IsList}(z) = 1, \mathsf{Len}(z) \circ y = \mathsf{Len}(\ell), \mathsf{ITR}(x, \ell) = \varepsilon, z = \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, x))\},$$

where $\mathsf{ITRL}(\ell, y)$ is the function defined as

$$\mathsf{ITRL}(\ell, \varepsilon) := \ell, \quad \mathsf{ITRL}(\ell, s_i(y)) := \mathsf{Tail}(\mathsf{ITRL}(\ell, y)) \quad (i \in \{0, 1\}).$$

More formally, let LenITR_3 be the for-loop in LenITR, we will prove the Hoare tuple $\{\varphi[x/\ell]\}\ \mathsf{LenITR}_3\ \{\varphi[x/\varepsilon]\}$. It is left as an exercise that Equation (4.20) can be derived from this Hoare tuple.

To prove $\{\varphi[x/\ell]\}\ \mathsf{LenITR}_3\ \{\varphi[x/\varepsilon]\}$, notice that LenITR_3 is a for-loop, and therefore we can apply the *rule of for-loop* so that it suffices to prove that

$$\{\varphi[x/s_i(x)]\}\ \textbf{if}\ \mathsf{IsEps}(z)\ \textbf{then skip else let}\ y := s_0(y);\ \textbf{let}\ z := \mathsf{Tail}(z)\ \{\varphi\}.$$

Subsequently, we can apply the *rule of if-then-else*, so that it suffices to prove the following two subgoals:

- (*Subgoal 1*). $\{\varphi[x/s_i(x)] \wedge \mathsf{LastBit}(\mathsf{IsEps}(z)) = 1\}$ **skip** $\{\varphi\}$
- (*Subgoal 2*). $\{\varphi[x/s_i(x)] \wedge \mathsf{LastBit}(\mathsf{IsEps}(z)) \neq 1\}$ **let** $y := s_0(y)$; **let** $z := \mathsf{Tail}(z)$ $\{\varphi\}$.

To prove (*Subgoal 1*), it suffices to prove in $\mathsf{PV}\text{-}\mathsf{PL}$ that

$$\varphi[x/s_i(x)] \wedge \mathsf{LastBit}(\mathsf{IsEps}(z)) = 1 \vdash \varphi,$$

apply the *consequence rule*, and subsequently apply the *empty statement axiom.* The PV-PL proof of the assertion above is easy and left as an exercise.

To prove (*Subgoal 2*), it suffices to prove in PV-PL that

$$\varphi[x/s_i(x)] \wedge \mathsf{LastBit}(\mathsf{IsEps}(z)) \neq 1 \vdash \varphi[z/\mathsf{Tail}(z)][y/s_0(y)],$$

apply the *consequence rule* to weaken the precondition to $\varphi[z/\mathsf{Tail}(z)][y/s_0(y)]$, and subsequently apply the *rule of composition* (with $\psi := \varphi[z/\mathsf{Tail}(z)]$) and the *assignment axiom* in both branches generated by the *rule of composition.* The proof of the assertion above is tedious but straightforward.

Similarly, we can define the Hoare logic of ω-length-restricted PV programs, where the *assignment axiom scheme* and the *rule of for-loop* are modified by replacing all occurrences of the expression e in the preconditions and postconditions to $\mathsf{Suf}(e, \omega)$. We will denote a Hoare tuple for ω-length-restricted programs by $\{\varphi_1\}$ (P, ω) $\{\varphi_2\}$.

It is left as an exercise that for the example above, we can pick an appropriate length upper bound ω such that we can still prove the correctness of the (ω-length-restricted) program.

4.2.3 Compiling Programs to PV Functions

Next, we show that it is possible to compile any (length-restricted) IMP(PV) program back to a PV function in a way that preserves the semantics of the function computed by the program.

Let (P, ω) be an ω-length-restricted IMP(PV) program. We define its PV translation, denoted also by $[P]_{\mathsf{PV}}$, as a PV function $f_P(\omega, \pi)$ follows.

- (*Context storage*). Suppose that $k \in \mathbb{N}$ variables are used in P named $x_1, \ldots, x_k$, we will use a k-tuple π of length k to store the context, i.e., $\mathsf{UnwindTuple}_i(\pi)$ stores the value of x_k. For simplicity, we will denote $\mathsf{UnwindTuple}_i(\pi)$ simply by π_i.

- (*Expressions*). For an expression e in P, where the variables are from $x_1, \ldots, x_k$, we define its PV translation with respect to the context π, denoted by $[e]^{\pi,\omega}_{\mathsf{PV}}$, as $\mathsf{Suf}(e[x_i/\pi_i\ \forall i \in [k]], \omega)$. We may omit ω in the superscript if there is no ambiguity.

- (*Empty statement*). We define $[\textbf{skip}]_{\mathsf{PV}}(\omega, \pi) := \pi$, i.e., the identity function.

- (*Assignment*). We define $[\textbf{let}\ x_i := e]_{\mathsf{PV}}(\omega, \pi)$ be the function that outputs the tuple $\mathsf{MakeTuple}(\pi_1, \ldots, \pi_{i-1}, [e]^{\pi}_{\mathsf{PV}}, \pi_{i+1}, \ldots, \pi_k)$.

- (*Composition*). We define $[c_1;\ c_2]_{\mathsf{PV}}(\omega, \pi) := [c_2]_{\mathsf{PV}}([c_1]_{\mathsf{PV}}(\pi))$, i.e., the PV translation commutes with composition of IMP(PV) programs.

- (*If-then-else*). We define the PV translation of if-then-else statement as

$$[\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_0]_{\mathsf{PV}}(\pi) := \mathsf{ITE}_0^{(\varepsilon)}([e]^{\pi}_{\mathsf{PV}}, [c_1]_{\mathsf{PV}}(\pi), [c_2]_{\mathsf{PV}}(\pi)).$$

Recall that $\mathsf{ITE}_0^{(\varepsilon)}(c, u, v)$ is the "dirty" if-then-else function, where the "else"-branch is chosen when $c = \varepsilon$.

- (*For Loop*). We define $[\textbf{for } x_i := e;\ x_i \neq \varepsilon;\ x_i := \mathsf{TR}(x_i) \textbf{ do } c]_{\mathsf{PV}}$ as the function $f(\omega, \pi)$ constructed below:
  - Let $f_{\mathsf{iter}}(\omega, \pi) := \mathsf{MakeTuple}(\pi'_1, \dots, \pi'_{i-1}, \mathsf{TR}(\pi'_i), \pi'_{i+1}, \dots, \pi'_k)$, where $\pi' = [c]_{\mathsf{PV}}(\omega, \pi)$.
  - Let $g(\omega, \pi) = \pi$, $h_i(\omega, \pi, y, z) = \pi'$, $k_i(\omega, \pi, y, z) = \pi''$, $i \in \{0, 1\}$, where

    $$\begin{aligned} \pi' &:= \mathsf{MakeTuple}^{(k)}(\mathsf{Suf}(f_{\mathsf{iter}}(\omega, z)_1, \omega), \dots, \mathsf{Suf}(f_{\mathsf{iter}}(\omega, z)_k, \omega)) \\ \pi'' &:= \mathsf{MakeTuple}^{(k)}(\omega, \omega, \dots, \omega) \end{aligned}$$

    It can be proved in PV that $\mathsf{ITR}(h_i(\omega, \pi, y, z), z \circ k_i(\omega, \pi, y, z)) = \varepsilon$ by the properties of $\mathsf{MakeTuple}$ and Suf, which is left as an exercise. Therefore, by the recursion rule of PV, there is a function f_{loop} satisfying that

    $$\begin{aligned} f_{\mathsf{loop}}(\omega, \pi, \varepsilon) &= g(\omega, \pi) \\ f_{\mathsf{loop}}(\omega, \pi, s_i(y)) &= h_i(\omega, \pi, y, f_{\mathsf{loop}}(\omega, \pi, y)) \end{aligned}$$

  - Let $f(\omega, \pi) = f_{\mathsf{loop}}(\omega, \mathsf{MakeTuple}(\pi_1, \dots, \pi_{i-1}, [e]^{\pi}_{\mathsf{PV}}, \pi_{i+1}, \dots, \pi_k), [e]^{\pi}_{\mathsf{PV}})$.

The following lemma shows that the PV translation of $\mathsf{IMP}(\mathsf{PV})$ programs correctly preserves the length restriction on variables:

Lemma 4.2.1 (Length Restriction)**.** *For any ω-length-restricted* $\mathsf{IMP}(\mathsf{PV})$ *program* (P, ω)*,* PV-PL *proves*

$$\pi = [P]_{\mathsf{PV}}(\omega, \mathsf{MakeTuple}(\mathsf{Suf}(x_1, \omega), \dots, \mathsf{Suf}(x_k, \omega)) \vdash \pi_i = \mathsf{Suf}(\pi_i, \omega)$$

for $i \in [k]$, where π_i is an abbreviation of $\mathsf{UnwindTuple}_i(\pi)$*.*

Proof Sketch. We will prove this by structural induction on the program P. All cases will essentially reduce to the PV equation that

$$\mathsf{Suf}(\mathsf{Suf}(x, \omega), \omega) = \mathsf{Suf}(x, \omega),$$

which is provable in PV by induction on x and w simultaneously using the (Ind$_2$) rule of PV-PL. □

4.2.4 Translating of Hoare Proofs to PV Proofs

It should be easy to see that in the standard model, the semantic of an ω-length-restricted $\mathsf{IMP}(\mathsf{PV})$ program is identical to its PV translation. In addition, we will now prove that Hoare proofs can also be translated back into PV proofs — providing a strong and intuitive tool to reason about $\mathsf{IMP}(\mathsf{PV})$ programs.

Hopefully feasible mathematicians will be satisfied with writing even less formal (pseudo) proofs after developing this result :)

Translation of Hoare tuples. We first demonstrate the formalization of Hoare tuples as PV-PL assertions. Let φ be a formula with variables $x_1, \ldots, x_k$, we define the translation of φ in the context π, denoted as φ^π as the formula obtained by replacing each x_i with $\pi_i (= \mathsf{UnwindTuple}_i^{(k)}(\pi))$. Then a Hoare tuple $\{\varphi_1\}\ (P, \omega)\ \{\varphi_2\}$ for ω-length-restricted programs is formalized as the PV-PL assertion

$$\Omega, \varphi_1 \vdash \varphi_2^{[c]_{\mathsf{PV}}(\mathsf{MakeTuple}(x_1,\ldots,x_k))},$$

where $\Omega := \{x_1 = \mathsf{Suf}(x_1, \omega), \ldots, x_k = \mathsf{Suf}(x_k, \omega)\}$. This assertion will be denoted by $[\{\varphi_1\}\ (P, \omega)\ \{\varphi_2\}]_{\mathsf{PV\text{-}PL}}$. In the rest of the chapter, we will use Ω as the abbreviation of $\{x_1 = \mathsf{Suf}(x_1, \omega), \ldots, x_k = \mathsf{Suf}(x_k, \omega)\}$ for simplicity.

Theorem 4.2.2. *Let (P, ω) be a length-restricted* $\mathsf{IMP}(\mathsf{PV})$ *program. Suppose that there is a Hoare proof of the tuple $\{\varphi_1\}\ (P, \omega)\ \{\varphi_2\}$, there is a* PV-PL *proof the the assertion*

$$\Omega, \varphi_1 \vdash \varphi_2^{[P]_{\mathsf{PV}}(\omega, \mathsf{MakeTuple}(x_1,\ldots,x_k))}.$$

Admissibility of Hoare rules. With the translation of Hoare tuples to PV-PL assertion, we can translate Hoare rules and axioms into PV-PL rules and axioms by translating both premises and the conclusion into PV-PL assertions. Since Hoare logic is also formulated as a natural deduction type system, it remains to prove that all Hoare rules (under this translation) are admissible in PV-PL.

We start by showing that the *consequence rule* is admissible.

Lemma 4.2.3. *The consequence rule is admissible in* PV-PL*:*

$$\frac{\varphi_1 \vdash \varphi_1' \quad \varphi_2' \vdash \varphi_2 \quad \Omega, \varphi_1' \vdash \varphi_2'^{[c]_{\mathsf{PV}}(\mathsf{MakeTuple}(x_1,\ldots,x_k))}}{\Omega, \varphi_1 \vdash \varphi_2^{[c]_{\mathsf{PV}}(\mathsf{MakeTuple}(x_1,\ldots,x_k))}}$$

Proof. Note that from the premises, it suffices to derive that

$$\varphi_2'^{[c]_{\mathsf{PV}}(\mathsf{MakeTuple}(x_1,\ldots,x_k))} \vdash \varphi_2^{[c]_{\mathsf{PV}}(\mathsf{MakeTuple}(x_1,\ldots,x_k))}$$

and apply the transitivity of PV-PL assertions.

Recall that $\varphi_2'^{[c]_{\mathsf{PV}}(\mathsf{MakeTuple}(x_1,\ldots,x_k))}$ (resp. $\varphi_2^{[c]_{\mathsf{PV}}(\mathsf{MakeTuple}(x_1,\ldots,x_k))}$) is obtained from φ_2' (resp. φ_2) by substituting

$$x_i / \mathsf{UnwindTuple}_i(\mathsf{MakeTuple}(x_1, \ldots, x_k))$$

for each $i \in [k]$. Therefore, this assertion can be proved from the premise $\varphi_2' \vdash \varphi_2$ directly using the substitution rule (V) of PV-PL. □

The proofs for the admissibility of other rules are similar, so we will only demonstrate the proofs of the *assignment axiom scheme* and *rule of for-loop.*

Lemma 4.2.4. *The assignment axiom scheme is admissible in* PV-PL*:*

$$\frac{}{\Omega, \varphi[x_i / \mathsf{Suf}(e, \omega)] \vdash \varphi^{[\textbf{\textit{let}}\ x_i := e]_{\mathsf{PV}}(\omega, \mathsf{MakeTuple}(x_1,\ldots,x_k))}}$$

Proof. By the definition of the PV translation of length-restricted $\mathsf{IMP(PV)}$ programs and the correctness of tuples (see Equation (2.53) to (2.55)), we know that PV proves

$$\begin{aligned}&[\textbf{let } x_i := e]_{\mathsf{PV}}(\omega, \mathsf{MakeTuple}(x_1, \dots, x_k))\\ &= \mathsf{MakeTuple}(x_1, \dots, x_{i-1}, \mathsf{Suf}(e, \omega), x_{i+1}, \dots, x_k).\end{aligned}$$

Moreover, notice that PV also proves

$$\varphi^{\mathsf{MakeTuple}(x_1,\dots,x_{i-1},\mathsf{Suf}([e],\omega),x_{i+1},\dots,x_k)} = \varphi[x_i/\mathsf{Suf}(e, \omega)].$$

Therefore the axiom scheme is provable by rewriting the consequence to $\varphi[x_i/\mathsf{Suf}(e, \omega)]$ (using Lemma 3.8.1) and then applying the assumption axiom of PV-PL. □

Before proving the admissibility of the for-loop rule, we need the following proposition showing that an ω-length restricted program will terminate with variables bounded by ω. Formally:

Proposition 4.2.5. *Let c be an* $\mathsf{IMP(PV)}$ *program.* PV-PL *proves for every $i \in [k]$ that $\Omega \vdash z_i = \mathsf{Suf}(z_i, \omega)$, where $z_i := ([c]_{\mathsf{PV}}(\omega, \mathsf{MakeTuple}(x_1, \dots, x_k)))_i$.*

The proof is a straightforward structural induction on the program c (in the meta-theory) and is left as an exercise.

Lemma 4.2.6. *Let c be an* $\mathsf{IMP(PV)}$ *program, P be the program **for** $x_i := e$; $x_i \neq \varepsilon$; $x_i := \mathsf{TR}(x_i)$ **do** c. The rule of for-loop, formalized as follows, is admissible in* PV-PL*:*

$$\frac{\Omega, \varphi[x_i/s_j(x_i)] \vdash \varphi^{[c]_{\mathsf{PV}}(\omega,\mathsf{MakeTuple}(x_1,\dots,x_k))} \quad (\forall j \in \{0,1\})}{\Omega, \varphi[x_i/\mathsf{Suf}(e, \omega)] \vdash \varphi[x_i/\varepsilon]^{[P]_{\mathsf{PV}}(\omega,\mathsf{MakeTuple}(x_1,\dots,x_k))}}$$

Proof Sketch. Let f_{iter}, f_{loop}, and f be the functions in the translation of for-loop (see Section 4.2.3) for P; in particular, $f(\omega, \pi) = [P]_{\mathsf{PV}}(\omega, \pi)$ and it satisfies that

$$f(\omega, \pi) = f_{\mathsf{loop}}(\omega, \mathsf{MakeTuple}(\pi_1, \dots, \pi_{i-1}, [e]^{\pi}_{\mathsf{PV}}, \pi_{i+1}, \dots, \pi_k), [e]^{\pi}_{\mathsf{PV}}),$$

where f_{loop} is recursively defined and satisfies that

$$\begin{aligned}f_{\mathsf{loop}}(\omega, \pi, \varepsilon) &= \pi\\ f_{\mathsf{loop}}(\omega, \pi, s_i(y)) &= \mathsf{MakeTuple}(\mathsf{Suf}(f_{\mathsf{iter}}(\omega, z)_1, \omega), \dots, \mathsf{Suf}(f_{\mathsf{iter}}(\omega, z)_k, \omega))\\ &\quad (z := f_{\mathsf{loop}}(\omega, \pi, y))\\ f_{\mathsf{iter}}(\omega, \pi) &= \mathsf{MakeTuple}(\pi'_1, \dots, \pi'_{i-1}, \mathsf{TR}(\pi'_i), \pi'_{i+1}, \dots, \pi'_k)\\ &\quad (\pi' := [c]_{\mathsf{PV}}(\omega, \pi))\end{aligned}$$

We first prove, by induction on y, that after $|y|$ iteration of the for-loop, x_i will be $\mathsf{ITR}(x_i, y)$, $x_j = \mathsf{Suf}(x_j, \omega)$, and the invariant φ still holds. Formally, let $\pi := \mathsf{MakeTuple}(x_1, \dots, x_k)$:

$$\Omega, \varphi \vdash \mathsf{ITR}(y, x_i) = \varepsilon \to (f_{\mathsf{loop}}(\omega, \pi, y))_i = \mathsf{ITR}(x_i, y); \tag{4.21}$$

$$\Omega, \varphi \vdash \mathsf{ITR}(y, x_i) = \varepsilon \to (f_{\mathsf{loop}}(\omega, \pi, y))_j = \mathsf{ITR}((f_{\mathsf{loop}}(\omega, \pi, y))_j, \omega); \tag{4.22}$$

$$\Omega, \varphi \vdash \mathsf{ITR}(y, x_i) = \varepsilon \to \varphi^{f_{\mathsf{loop}}(\omega,\pi,y)}. \tag{4.23}$$

for $j \in [k]$.

Note that Equation (4.21) formalizes that x_i will be $\mathsf{ITR}(x_i, y)$ after $|y|$ iterations; the base case is straightforward, while the induction case requires that $([c]_{\mathsf{PV}}(\omega, \pi))_i = \pi_i$, which is true as x_i is not changed in the for-loop by the *read-only restriction*, and can be formally proved by structural induction on the definition of c (in the meta-theory). Similarly, Equation (4.22) formalizes that all variables are bounded by ω, which can be proved by induction on y and apply Proposition 4.2.5.

Equation (4.23) formalizes that the loop invariant φ holds throughout the loop; the base case is also straightforward, so we will only explain the proof of the induction case of Equation (4.23). That is, suppose that

$$\Omega, \varphi, \mathsf{ITR}(y, x_i) = \varepsilon \to \varphi^{f_{\mathsf{loop}}(\omega,\pi,y)}, \tag{4.24}$$

we need to prove for $\sigma \in \{0, 1\}$ that

$$\mathsf{ITR}(s_\sigma(y), x_i) = \varepsilon \to \varphi^{f_{\mathsf{loop}}(\omega,\pi,s_\sigma(y))},$$

which is equivalent to say (by the definition axiom of f_{loop}) that:

$$\mathsf{ITR}(s_\sigma(y), x_i) = \varepsilon \to \varphi^{\mathsf{MakeTuple}(\mathsf{Suf}(f_{\mathsf{iter}}(\omega,z)_1,\omega),\ldots,\mathsf{Suf}(f_{\mathsf{iter}}(\omega,z)_k,\omega))}$$

where $z := f_{\mathsf{loop}}(\omega, \pi, y)$. By Equation (4.22) and Proposition 4.2.5, we know that all coordinates of $f_{\mathsf{iter}}(\omega, z)$ are bounded by ω, and thus it suffices to prove that $\mathsf{ITR}(s_\sigma(y), x_i) = \varepsilon \to \varphi^{f_{\mathsf{iter}}(\omega,z)}$, i.e., for $\pi' := [c]_{\mathsf{PV}}(\omega, z)$:

$$\mathsf{ITR}(s_\sigma(y), x_i) = \varepsilon \to \varphi^{\mathsf{MakeTuple}(\pi'_1,\ldots,\pi'_{i-1},\mathsf{TR}(\pi'_i),\pi'_{i+1},\ldots,\pi'_n)}. \tag{4.25}$$

Suppose that $\mathsf{ITR}(s_\sigma(y), x_i) = \varepsilon$, we also know that $\mathsf{ITR}(y, x_i) = \varepsilon$.[4] Recall that we have $\pi'_i = z_i = f_{\mathsf{loop}}(\omega, \pi, y)$ as the variable x_i is read-only in the program c. By Equation (4.21) and $\mathsf{ITR}(y, x_i) = \varepsilon$, we further know that $\pi'_i = \mathsf{ITR}(x_i, y)$. Moreover, we know by $\mathsf{ITR}(s_\sigma(y), x_i) = \varepsilon$ that $\pi'_i \neq \varepsilon$, and thus $\pi'_i = s_{\sigma'}(\mathsf{TR}(\pi'_i))$ for some $\sigma' \in \{0, 1\}$. This allows us to apply the premise

$$\Omega, \varphi[x_i/s_{\sigma'}(x_i)] \vdash \varphi^{[c]_{\mathsf{PV}}(\omega,\mathsf{MakeTuple}(x_1,\ldots,x_k))}$$

by substituting $x_i/\mathsf{TR}(\pi'_i)$ and x_j/z_j for any $j \in [k] \setminus \{i\}$.

Let δ be the aforementioned substitution, we can see that $\Omega[\delta]$ is provable by Equation (4.22), $\varphi[x_i/s_{\sigma'}(x_i)][\delta]$ is given by the antecedent Equation (4.24) of the assertion we are proving, and $\varphi^{[c]_{\mathsf{PV}}(\omega,\mathsf{MakeTuple}(x_1,\ldots,x_k))}[\delta]$ is exactly what we need to prove (see Equation (4.25)). This completes the induction case. □

4.3 Simulation of Turing Machines

Now we are ready to show that there is a PV-function computing a universal Turing machine in the standard model $\mathbb{M}$, which completes the proof of Theorem 2.3.2 that every function in FP can be defined in PV.

[4] This can be proved by structural induction on the program in meta-theory.

Moreover, we can see through the construction that the PV-function for a universal Turing machine is defined straightforwardly; in particular, properties of Turing machines that are intuitively feasibly provable are likely to be provable in PV, which justifies the Feasible Mathematics Thesis.

Encoding of Turing machines. We will simulate ordinary single-tape Turing machines with a two-way infinite tape. Nevertheless, the same construction generalizes directly to machine models such as multi-tape Turing machines and RAMs.

- (*Alphabet*): For some parameter k (encoded in unary), the alphabet is $\Sigma := \{0,1\}^{\le k}$, and the empty string ε is used to encode "blank".
- (*States*): For some parameter s (encoded in binary), the Turing machine has state $[s]$, where 1 is the initial state and s is the only halting state.
- (*Transition*): The transition function is encoded by a dictionary $\Gamma : [s] \times \Sigma \to [s] \times \{0,1\}$. That is, for every state $u \in [s]$ and character $x \in \Sigma$ read by the head, $\mathsf{Lookup}(\Gamma, \mathsf{MakePair}(u,x))$ outputs a pair $\mathsf{MakePair}(u', m)$, where $u' \in [s]$ is the new state, and $m \in \{0,1\}$ encodes the movement of the head.

Formally, a Turing machine is defined as a tuple (K, s, Γ) such that the following assertions are provable in $\mathsf{PV\text{-}PL}$:

$$[u \in [s]] = 1, \mathsf{ITR}(x, K) = \varepsilon \vdash [\mathsf{Left}(\mathsf{Lookup}(\Gamma, \mathsf{MakePair}(u,x))) \in [s]] = 1;$$
$$[u \in [s]] = 1, \mathsf{ITR}(x, K) = \varepsilon \vdash \mathsf{Right}(\mathsf{Lookup}(\Gamma, \mathsf{MakePair}(u,x))) \neq \varepsilon;$$
$$[u \in [s]] = 1, \mathsf{ITR}(x, K) = \varepsilon \vdash \mathsf{TR}(\mathsf{Right}(\mathsf{Lookup}(\Gamma, \mathsf{MakePair}(u,x)))) = \varepsilon;$$

where $[u \in [s]]$ is the PV-function checking whether u encodes a natural number in $[s]$, which can be easily defined by simultaneous induction on u and s (see Theorem 4.1.3), or alternatively by a straightforward length-restricted $\mathsf{IMP(PV)}$ program.

Simulation of tapes. We will implement the two-way infinite tape by two lists ℓ_{left} and ℓ_{right}. The tape is supposed to be $\ell_{\mathsf{left}} \mathbin{+\!+} \mathsf{RevList}(\ell_{\mathsf{right}})$, where $\mathsf{RevList}(\ell)$ denotes the reversion of the list ℓ, and can be defined by recursion on lists (see Theorem 4.1.3), or by a straightforward length-restricted $\mathsf{IMP(PV)}$ program. The head position is at the rightmost (i.e. outermost) entry of ℓ_{right}, i.e., the head is reading $\mathsf{Head}(\ell_{\mathsf{right}})$.

Suppose that we are simulating a Turing machine for $|T|$ steps, we initialize both lists with $|T|$ "blanks" (i.e. ε) so that there is no need to check whether the lists are empty. This can be implemented by the $\mathsf{IMP(PV)}$ commands:

$$(\textit{initialize tape})\text{: } \textbf{let } \ell_{\mathsf{left}} := \mathsf{GenList}(\varepsilon, T);\ \textbf{let } \ell_{\mathsf{right}} := \mathsf{GenList}(\varepsilon, T),$$

where $\mathsf{GenList}$ is defined by Equation (4.5) and (4.6). The movement of the head can thus be simulated as:

- (*move right*): **let** $\ell_{\mathsf{left}} := \ell_{\mathsf{left}} :: \mathsf{Head}(\ell_{\mathsf{right}})$; **let** $\ell_{\mathsf{right}} := \mathsf{Tail}(\ell_{\mathsf{right}})$;
- (*move left*): **let** $\ell_{\mathsf{right}} := \ell_{\mathsf{right}} :: \mathsf{Head}(\ell_{\mathsf{left}})$; **let** $\ell_{\mathsf{left}} := \mathsf{Tail}(\ell_{\mathsf{right}})$.

Generating the output. At the end of the simulation, we need to generate the output of the Turing machine by reading the tape encoded by ℓ_{left} and ℓ_{right}. We make the convention that, in case the Turing machine reaches the halting state s, the output of the Turing machine is the longest continuous string from the current head position to the right direction until reaching an ε.

We can define a function $\mathsf{Output}(\ell)$ using recursion on lists (see Theorem 4.1.3):

$$\mathsf{Output}(\varepsilon) = \varepsilon, \tag{4.26}$$

$$\mathsf{Output}(\ell :: u) = \mathsf{ITE}(\mathsf{IsEps}(u), \varepsilon, u \circ \mathsf{Output}(\ell)). \tag{4.27}$$

It is clear to see that $\mathsf{Output}(\ell_{\mathsf{right}})$ is the output of the Turing machine.

Universal Turing machine. We now describe the PV function for a universal Turing machine by a length-restricted IMP(PV) program. The program has the following variables: K, s, Γ intended to describe a Turing machine, T encoding in unary the number of steps to simulate, $\ell_{\mathsf{left}}, \ell_{\mathsf{right}}$ intended to simulate the tape, u intended to simulate the current state, r to record the output of the Turing machine, and auxiliary variables i, x_1, x_2. At the beginning of the program, we call (*initialize tape*) and **let** $u := 1$ to initialize the context.

To simulate one step of a Turing machine $M = (K, s, \Gamma)$, we proceed as follows:

```
(one step): let $x_1 := \mathsf{Tail}(\ell_{\mathsf{right}})$;
            let $x_2 := \mathsf{Lookup}(\Gamma, \mathsf{MakePair}(u, x))$;
            let $u := \mathsf{Left}(x_2)$;
            if $\mathsf{Right}(x_2)$ then (move left) else (move right)
```

We will obtain the IMP(PV) program for a universal Turing machine:

```
(UTM): (initialize tape); let $u := 1$;
       for $i := T$; $i \neq \varepsilon$; $i := \mathsf{TR}(i)$ do
          if $\mathsf{EQ}(u, s)$ then skip else (one step);
       if $\mathsf{EQ}(u, s)$ then let $r := \mathsf{Output}(\ell_{\mathsf{right}})$ else let $r := \varepsilon$
```

Compiling to PV. Finally, notice that the length of all variables is at most

$$\max\{|\mathsf{GenList}(K, T)|, |s|\}$$

provided that the Turing machine is valid, we can set $\omega := \mathsf{GenList}(K, T) \circ s$ and define a PV function $U(K, s, \Gamma, T)$ as

$$U(K, s, \Gamma, T) := \mathsf{GetVar}_r([(\mathrm{UTM})]_{\mathsf{PV}}(\mathsf{GenList}(K, T) \circ s, \pi_{K,s,\Gamma,T})),$$

where $\mathsf{GetVar}_r(\pi)$ denotes the PV function that unwind the tuple π and outputs the entry corresponding to the variable r, $\pi_{K,s,\Gamma,T}$ is the tuple that assigns variables

$$\ell_{\mathsf{left}}, \ell_{\mathsf{right}}, r, i, x_1, x_2$$

to be ε and K, s, Γ, T to be the corresponding variables in the input of U.

It is clear that this PV function $U(K, s, \Gamma, T)$, in the standard model, simulates the Turing machine that is encoded by (K, s, Γ) for at most $|T|$ steps and returns the output of the Turing machine in case that it terminates.

4.4 Feasible Set Theory

In this section, we will develop an extension of PV-PL that allows set-theoretic operations including union, intersection, and membership query, as well as universal and existential quantifiers over a finite set encoded by a list.

4.4.1 Membership, Quantification, and Subset Relation

We first define membership relation, existence and universal quantifiers over sets, and the subset relation. Following set-theoretic notation, we will denote the empty set ε by $\varnothing$.

Membership relation. Recall that in the previous chapter, we have already defined a function $\mathsf{Find}_x(\ell)$ that outputs 1 if and only if x is in the list ℓ. With the abbreviation $[x \in \ell] \equiv$ "$\mathsf{Find}_x(\ell) = 1$", we can prove in PV-PL that

$$\vdash \neg[x \in \varnothing], \tag{4.28}$$

$$\vdash [x \in \ell :: x], \tag{4.29}$$

$$[x \in \ell] \vdash [x \in \ell :: u]. \tag{4.30}$$

The following proposition can be a good exercise for the readers to get familiar with the definition of the membership relation:

Hint: Prove that the output length of $\mathsf{Find}_x(\ell)$ is smaller than the length of ℓ.

Proposition 4.4.1. PV-PL *proves that* $\mathsf{IsList}(x) = 1 \vdash x \notin x$.

Quantification. Similar to the membership relation, we will define universal and existential quantifiers over a list ℓ. Concretely, let $\varphi(\vec{x}, y)$ be a PV-PL formula, $\exists y \in \ell : \varphi$ is the abbreviation of a PV-PL formula with variables $\vec{x}$ defined as $\mathsf{Find}_{[\varphi(\vec{x},y)]_{\mathsf{PV}}}(\ell) \neq \varepsilon$. Subsequently, we can define $\forall y \in \ell : \varphi \equiv \neg\exists y \in \ell : \neg\varphi$. Note that the bounded variable y is *not* a variable of the PV-PL formula $\exists y \in \ell : \varphi$.

We will show that the PV-PL rules usually considered as the logical rules of quantifiers are admissible in PV-PL. This justifies the definition of quantifiers.

Theorem 4.4.2. *The following rules are admissible in* PV-PL*. Suppose that* $\Gamma = \Gamma', \mathsf{IsList}(\ell) = 1$*, then:*

$$(\exists_i) : \quad \frac{\Gamma \vdash \varphi[y/t] \quad \Gamma \vdash [t \in \ell]}{\Gamma \vdash \exists y \in \ell : \varphi} \tag{4.31}$$

$$(\exists_e) : \quad \frac{\Gamma \vdash \exists y \in \ell : \varphi \quad \Gamma, [z \in \ell], \varphi[y/z] \vdash \psi}{\Gamma \vdash \psi} \tag{4.32}$$

where in $(\exists_i)$ t is an arbitrary term, and in $(\exists_e)$ z must be a fresh variable that has no occurrence in Γ, φ, ψ, y.

Proof. We first consider the introduction rule $(\exists_i)$. Indeed, we will show that the assertion

$$\mathsf{IsList}(\ell'), \varphi[y/t] \vdash [t \in \ell'] \to \exists y \in \ell' : \varphi$$

is provable in PV-PL, where t' is a fresh variable that has no occurrence in φ, t. The admissibility of $(\exists_i)$ follows from the provability of the assertion.

We will prove the assertion by induction on ℓ' using Corollary 4.1.6. In the base case, i.e., $\ell'/\varnothing$, it is easy to see by the definition of the membership relation (and subsequently the definition axioms of Find_x) that $[t \in \varnothing]$ is false. In the induction case, suppose that we have $\mathsf{IsList}(\ell'), \varphi[y/t]$ and $[t \in \ell'] \to \exists y \in \ell' : \varphi$ in the antecedent, we need to prove that

$$[t \in \ell' :: u] \to \exists y \in \ell' :: u : \varphi.$$

We introduce $[t \in \ell' :: u]$ as an assumption and aim to prove $\exists y \in \ell' :: u : \varphi$. We perform a case study on whether $t = u$ (recall that excluded middle is available by the cut rule and proof by contradiction, see Remark 3.2.2).

- If $t = u$, we can see $\varphi[y/u]$ (by rewriting), and thus $\mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell' :: u) = s_0(u) \neq \varepsilon$, which means that $\exists y \in \ell' :: u : \varphi$ by the definition as well as the comprehension of PV translation in PV-PL (see Lemma 3.8.2).
- If $t \neq u$, we can see that $\mathsf{Find}_t(\ell' :: u) = \mathsf{Find}_t(\ell')$, so that $[t \in \ell' :: u]$ implies that $[t \in \ell']$. By the induction hypothesis (and the elimination rule of implication), we can show that $\exists y \in \ell' : \varphi$. This implies the goal $\exists y \in \ell' :: u : \varphi$ by a case study on whether $\varphi[y/u]$ holds.

Next, we prove that the elimination rule $(\exists_e)$ is admissible. From the premise that $\Gamma \vdash \exists y \in \ell : \varphi$ and the definition of the existential quantifier, we know that $\Gamma \vdash \mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell) \neq \varnothing$, which, by the correctness of Find_f, deduces that

$$\Gamma \vdash [\varphi]_{\mathsf{PV}}(y/\mathsf{TR}(\mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell))) = 1.$$

By the comprehension rule of the PV translation, we further know that

$$\Gamma \vdash \varphi[y/\mathsf{TR}(\mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell))].$$

Similarly, we have that $\Gamma \vdash [\mathsf{TR}(\mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell)) \in \ell]$.

By the substitution $z/\mathsf{TR}(\mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell))$ to the second premise, we can prove the assertion

$$\Gamma, [\mathsf{TR}(\mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell)) \in \ell], \varphi[y/\mathsf{TR}(\mathsf{Find}_{[\varphi]_{\mathsf{PV}}}(\ell))] \vdash \psi.$$

Notice that all formulas in the antecedent are provable from Γ as we demonstrated above. Therefore, by the cut rule, we can conclude that $\Gamma \vdash \psi$. □

Using the admissibility of $(\exists_i)$ and $(\exists_e)$ and rules in PV-PL, we can deduce the admissibility of the introduction and elimination rules for the universal quantifier:

Corollary 4.4.3. *The following rules are admissible in* $\mathsf{PV}\text{-}\mathsf{PL}$.

$$(\forall_i) : \quad \frac{\Gamma, z \in \ell \vdash \varphi[y/z]}{\Gamma \vdash \forall y \in \ell : \varphi} \qquad (\forall_e) : \quad \frac{\Gamma \vdash \forall y \in \ell : \varphi}{\Gamma, t \in \ell \vdash \varphi[y/t]} \tag{4.33}$$

where in $(\forall_i)$ z *must be a fresh variable that has no occurrence in* Γ, φ, y, *and in* $(\forall_e)$ t *is an arbitrary term.*

The proof is left as an exercise.

Subset relation. Using the universal quantifier over sets, we can easily define the subset relation $[\ell_1 \subseteq \ell_2] := \forall x \in \ell_1 : [x \in \ell_2]$. For simplicity, we define a notation $\mathsf{IsList}[\ell_1, \dots, \ell_k]$ as the sequence of formulas

$$\mathsf{IsList}(\ell_1) = 1, \dots, \mathsf{IsList}(\ell_k) = 1.$$

The following lemma shows that the basic properties of the subset relation are provable.

Lemma 4.4.4. *The following assertions are provable in* $\mathsf{PV}\text{-}\mathsf{PL}$*:*

- *(Reflexivity):* $\mathsf{IsList}[\ell] \vdash \ell \subseteq \ell$.
- *(Transitivity):* $\mathsf{IsList}[\ell_1, \ell_2, \ell_3], \ell_1 \subseteq \ell_2, \ell_2 \subseteq \ell_3 \vdash \ell_1 \subseteq \ell_3$.
- *(Empty Set):* $\mathsf{IsList}[\ell] \vdash \varnothing \subseteq \ell$.

All three properties follow easily from the basic properties of membership relation and the introduction and elimination rules of quantifiers.

With the subset relation, we can define $\ell_1 \equiv \ell_2$ meaning that ℓ_1 and ℓ_2 denotes the same set as $\ell_1 \subseteq \ell_2 \wedge \ell_2 \subseteq \ell_1$. It can be easily proved that "$\equiv$" is an equivalence relation, i.e., reflexive, transitive, and symmetric.

Remark 4.4.1. One difference between our current definition and the standard set theory formulation is that our sets do not satisfy that *axiom of extensionality*, namely two sets ℓ_1, ℓ_2 may not be "identical" (i.e. $\ell_1 = \ell_2$) if they contain the same elements. One may define a PV function sorting and deduplicating lists to ensure that the property is satisfied, which can be implemented using (for instance) the insertion sort algorithm that can be easily implemented by an $\mathsf{IMP}(\mathsf{PV})$ program. The details are omitted and left as an exercise for interested readers.

4.4.2 Specification Axiom Scheme

The specification axiom scheme in set theory allows us to construct a subset $T \subseteq S$ that exactly contains the elements in S that satisfy a property φ. If the property φ is feasibly verifiable (e.g. it is a $\mathsf{PV}\text{-}\mathsf{PL}$ formula), we can indeed efficiently construct the subset T given S. This is formalized as the specification axiom scheme:

Lemma 4.4.5. *For every* $\mathsf{PV}\text{-}\mathsf{PL}$ *formula* $\varphi(x, \vec{w})$, *there is a* PV *function* Select_φ *such that the following assertions are provable in* $\mathsf{PV}\text{-}\mathsf{PL}$*:*

$$\mathsf{IsList}[\ell] \vdash \mathsf{IsList}[\mathsf{Select}_\varphi(\ell, \vec{w})]; \tag{4.34}$$

$$\mathsf{IsList}[\ell] \vdash \forall x \in \ell : \varphi(x, \vec{w}) \to [x \in \mathsf{Select}_\varphi(\ell, \vec{w})]; \tag{4.35}$$

$$\mathsf{IsList}[\ell] \vdash \forall x \in \mathsf{Select}_\varphi(\ell, \vec{w}) : \varphi(x, \vec{w}). \tag{4.36}$$

Following standard notation, $\{x \in \ell \mid \varphi\}$ is an abbreviation of $\mathsf{Select}_\varphi(\ell, \vec{w})$.

The proof of the lemma is straightforward, so we will only sketch the proof. The function $\mathsf{Select}_\varphi(\ell, \vec{w})$ is defined using recursion on lists (see Theorem 4.1.3):

$$\mathsf{Select}_\varphi(\varepsilon, \vec{w}) = \varepsilon \tag{4.37}$$

$$\mathsf{Select}_\varphi(\ell :: u, \vec{w}) = \mathsf{ITE}([\varphi]_{\mathsf{PV}}(u, \vec{w}), \mathsf{Select}_\varphi(\ell, \vec{w}) :: u, \mathsf{Select}_\varphi(\ell)) \tag{4.38}$$

All these three properties can be proved by induction on ℓ using Corollary 4.1.6, and we will only demonstrate Equation (4.36).

In base case (i.e. ℓ/ε), we know by the definition axiom of Select_φ that it suffices to prove $\forall x \in \varnothing : \varphi(x, \vec{w})$. By the introduction rule of the universal quantifier, it suffices to prove that $\mathsf{IsList}[\ell], [x \in \varnothing] \vdash \varphi(x, \vec{w})$. This is provable as $\vdash \neg[x \in \varnothing]$ is provable (see Equation (4.28)).

In the induction case, we need to prove from $\Gamma = (\mathsf{IsList}[\ell], \forall x \in \mathsf{Select}_\varphi(\ell, \vec{w}) : \varphi(x, \vec{w}))$ that $\forall x \in \mathsf{Select}_\varphi(\ell :: u, \vec{w}) : \varphi(x, \vec{w})$. By the definition axiom of Select_φ, it suffices to prove that

$$\psi := \forall x \in \mathsf{ITE}([\varphi]_{\mathsf{PV}}(u, \vec{w}), \mathsf{Select}_\varphi(\ell, \vec{w}) :: u, \mathsf{Select}_\varphi(\ell)) : \varphi(x, \vec{w}).$$

We prove by a case study on $\varphi(u, \vec{w})$; the case study is done by the provability of the law of excluded middle (see Remark 3.2.2). The case study generates two subgoals:

- (*Subgoal 1*): $\Gamma, \neg\varphi(u, \vec{w}) \vdash \forall x \in \psi$. In this case, we can see that $[\varphi(u, \vec{w})]_{\mathsf{PV}} = 0$ by the comprehension of PV translation (see Lemma 3.8.2), and we can then rewrite ψ as
$$\forall x \in \mathsf{Select}_\varphi(\ell) : \varphi(x, \vec{w}),$$
which has already been available in Γ.

- (*Subgoal 2*): $\Gamma, \varphi(u, \vec{w}) \vdash \forall x \in \psi$. In this case, we can see that $[\varphi(u, \vec{w})]_{\mathsf{PV}} = 1$ by the comprehension of PV translation (see Lemma 3.8.2), and we can then rewrite ψ as
$$\forall x \in \mathsf{Select}_\varphi(\ell) :: u : \varphi(x, \vec{w}).$$
By the introduction rule, we can add $[x \in \mathsf{Select}_\varphi(\ell :: u)]$ into the antecedent towards proving $\varphi(x, \vec{w})$. Finally, we perform a case study on whether $x = u$, where both cases can be resolved by $\varphi(u, \vec{w})$ and the induction hypothesis $\forall x \in \mathsf{Select}_\varphi(\ell, \vec{w}) : \varphi(x, \vec{w})$.

This completes the proof of Equation (4.36).

4.4.3 Union, Intersection, and Cartesian Product

The union of two sets $\ell_1 \cup \ell_2$ can be simply defined by the concatenation of the lists, i.e, $\ell_1 \cup \ell_2 := \ell_1 \mathbin{++} \ell_2$. The intersection requires the specification axiom scheme: $\ell_1 \cap \ell_2 := \{x \in \ell_2 \mid [x \in \ell_1]\}$.

Lemma 4.4.6. PV-PL *proves the following assertions:*

- $\mathsf{IsList}[\ell_1, \ell_2] \vdash [x \in \ell_1 \cup \ell_2] \leftrightarrow [x \in \ell_1] \vee [x \in \ell_2]$
- $\mathsf{IsList}[\ell_1, \ell_2] \vdash [x \in \ell_1 \cap \ell_2] \leftrightarrow [x \in \ell_1] \wedge [x \in \ell_2]$
- $\mathsf{IsList}[\ell_1, \ell_2] \vdash \ell_1 \cap \ell_2 \equiv \ell_2 \cap \ell_1$
- $\mathsf{IsList}[\ell_1, \ell_2] \vdash \ell_1 \cup \ell_2 \equiv \ell_2 \cup \ell_1$
- $\mathsf{IsList}[\ell_1, \ell_2, \ell_3] \vdash \ell_1 \cap (\ell_2 \cup \ell_3) \equiv (\ell_1 \cap \ell_2) \cup (\ell_1 \cap \ell_3)$.
- $\mathsf{IsList}[\ell_1, \ell_2, \ell_3] \vdash \ell_1 \cup (\ell_2 \cap \ell_3) \equiv (\ell_1 \cup \ell_2) \cap (\ell_1 \cup \ell_3)$.

The proofs are omitted and left as an exercise. Also, this is of course not a complete list of properties we can prove in PV-PL; most straightforward properties of intersection and union should be provable.

We can also define the Cartesian product[5] of two sets ℓ_1 and ℓ_2. For instance, we can first define a function $\mathsf{Cartesian}'(x, \ell)$ that constructs the set $\{x\} \times \ell$ by recursion on lists using Theorem 4.1.3:

$$\mathsf{Cartesian}'(x, \varepsilon) = \varepsilon, \tag{4.39}$$

$$\mathsf{Cartesian}'(x, \ell :: u) = \mathsf{Cartesian}'(x, \ell) :: \mathsf{MakePair}(x, u). \tag{4.40}$$

Next, we define a function $\mathsf{Cartesian}(\ell_1, \ell_2)$ by recursion on ℓ_1:

$$\mathsf{Cartesian}(\varepsilon, \ell_2) = \varepsilon, \tag{4.41}$$

$$\mathsf{Cartesian}(\ell_1 :: u, \ell_2) = \mathsf{Cartesian}(\ell_1, \ell_2) \cup \mathsf{Cartesian}'(u, \ell_2). \tag{4.42}$$

Following standard notation, we may use $\ell_1 \times \ell_2$ as an abbreviation of $\mathsf{Cartesian}(\ell_1, \ell_2)$.

By induction on lists (see Corollary 4.1.6), we can prove the following meta-theorem showing the correctness of the Cartesian product:

Lemma 4.4.7. PV-PL *proves the following assertions:*

- $\mathsf{IsList}[\ell_1, \ell_2] \vdash \mathsf{IsList}[\ell_1 \times \ell_2]$
- $\mathsf{IsList}[\ell_1, \ell_2] \vdash \forall x \in \ell_1 \times \ell_2 : x = \mathsf{MakePair}(\mathsf{Left}(x), \mathsf{Right}(x))$
- $\mathsf{IsList}[\ell_1, \ell_2] \vdash \forall x \in \ell_1 \times \ell_2 : \mathsf{Left}(x) \in \ell_1 \wedge \mathsf{Right}(x) \in \ell_2$
- $\mathsf{IsList}[\ell_1, \ell_2] \vdash \forall x \in \ell_1 : \forall y \in \ell_2 : \mathsf{MakePair}(x_1, x_2) \in \ell_1 \times \ell_2$

The detail is omitted and left as an exercise.

Hint: Induction on ℓ_1 first and then on ℓ_2, which follows from the order when we recursively define the Cartesian product.

4.4.4 Counting

Except for ordinary set-theoretic operations, finite sets are widely used as the mathematical framework for combinatorics.

Indeed, for each PV-definable equivalence relation R (e.g. EQ or $\equiv$), one can define a PV function $\mathsf{Card}_R(\ell)$ that counts the number of different equivalence classes in ℓ with respect to R. For instance, we can use recursion on lists (see Theorem 4.1.3), or the

[5] The standard construction of Cartesian product in ZF set theory depends on the *power set axiom*, which is not feasible.

following $\mathsf{IMP(PV)}$ program:

```
$(\mathsf{Card}_R)$ : let $S := \varnothing$; let $\ell' := \ell$;
    for $i := \mathsf{Len}(\ell')$; $i \neq \varepsilon$; $i := \mathsf{TR}(i)$ do
        if $\mathsf{Find}_{[R]_{\mathsf{PV}}(\mathsf{Head}(\ell),\cdot)}(S)$ then let $S := S :: \mathsf{Head}(\ell)$ else skip;
        let $\ell := \mathsf{Tail}(\ell)$;
    let $r := \mathsf{Len}(S)$
```

As none of the intermediate variables will be longer than ℓ, we can fix the length bound $\omega = \ell$ and translate the $\mathsf{IMP(PV)}$ program to a PV function.

We introduce the notation $\mathsf{Equiv}[R]$ as the sequence of sentences $R(x,x)$, $R(x,y) \to R(y,x)$, $R(x,y) \to R(y,z) \to R(x,z)$; namely, R is an equivalence relation. Similarly, we may use $\mathsf{Equiv}[R_1,\dots,R_k]$ as a shorthand of $\mathsf{Equiv}[R_1],\dots,\mathsf{Equiv}[R_k]$.

We can prove basic counting principles, such as the addition principle and multiplication principle, are provable in $\mathsf{PV\text{-}PL}$:

Lemma 4.4.8. $\mathsf{PV\text{-}PL}$ *proves the following assertions:*

- *(Addition Principle):* $\mathsf{Equiv}[R], \mathsf{IsList}[\ell_1,\ell_2], \forall x \in \ell_1\ \forall y \in \ell_2\ \neg R(x,y) \vdash \mathsf{Card}_R(\ell_1 \cup \ell_2) = \mathsf{Card}_R(\ell_1) \circ \mathsf{Card}_R(\ell_2)$*.*
- *(Multiplication Principle):* $\mathsf{Equiv}[R], \mathsf{IsList}[\ell_1,\ell_2], R(\mathsf{MakePair}(x,y), \mathsf{MakePair}(x',y')) \leftrightarrow R(x,x') \wedge R(y,y') \vdash \mathsf{Card}_R(\ell_1 \times \ell_2) = \mathsf{Card}_R(\ell_1) \# \mathsf{Card}_R(\ell_2)$*.*

Moreover, $\mathsf{PV\text{-}PL}$ also proves the *inclusion-exclusion principle*:

Lemma 4.4.9. $\mathsf{PV\text{-}PL}$ *proves the following assertion:*

$$\mathsf{Equiv}[R], \mathsf{IsList}[\ell_1,\ell_2] \vdash \mathsf{Card}_R(\ell_1 \cup \ell_2) = \mathsf{ITR}(\mathsf{Card}_R(\ell_1) \circ \mathsf{Card}_R(\ell_2), \mathsf{Card}_R(\ell_1 \cap \ell_2)).$$

The proofs of these lemma can be down by induction on ℓ_1 and ℓ_2 in appropriate order, which is left as an exercise to the readers.

4.5 Proof of the Recursion on Lists Meta-theorem

Recall the statement of Theorem 4.1.3:

Theorem 4.1.3 (Recursion on Lists)**.** *Let* $g(\vec{x})$*,* $h(\vec{x},u,\ell,z)$*,* $k(\vec{x},u,\ell)$ *be* PV *functions satisfying that*

$$\mathsf{PV} \vdash \mathsf{ITR}(h(\vec{x},u,\ell,z), z \circ k(\vec{x},u,\ell)) = \varepsilon.$$

Then there is a PV *function* $f(\vec{x},\ell)$ *such that* PV *proves:*

$$f(\vec{x},\varepsilon) = g(\vec{x}) \tag{4.7}$$

$$f(\vec{x},\ell :: u) = h(\vec{x},u,\ell,f(\vec{x},\ell)) \tag{4.8}$$

The intuition of our proof is to define a function $f'(\vec{x},\ell,y)$ that outputs $f(\vec{x},\hat{\ell}_y)$, where $\hat{\ell}_y$ denotes the list obtained by iteratively removing the last (i.e. rightmost) element in ℓ for $y' := \mathsf{ITR}(\ell,y)$ times. Note that the function $\mathsf{ITRL}(\ell,y)$ that outputs the list obtained by iteratively removing the last element in ℓ for y times can be defined naturally by recursion on y, and we can prove that $\mathsf{ITRL}(\ell,\ell)=\varepsilon$.

Subsequently, $f'(\vec{x},\ell,y)$ can be defined by recursion on y. Note that when $y=\varepsilon$, we have that $\ell_y=\varepsilon$ and thus $f'(\vec{x},\ell,\varepsilon)=f(\vec{x},\varepsilon)$. Moreover, observe that

$$f'(\vec{x},\ell,s_i(y)) = \begin{cases} f(\vec{x},\varepsilon) & \mathsf{IsEps}(\hat{\ell}) \\ h(\vec{x},\mathsf{Head}(\hat{\ell}),\mathsf{Tail}(\hat{\ell}),f'(\vec{x},\ell,y)) & \text{otherwise} \end{cases}$$

where $\hat{\ell} := \mathsf{ITRL}(\ell,\mathsf{ITR}(\ell,s_i(y)))$. Finally, we will define $f(\vec{x},\ell) := f'(\vec{x},\ell,\ell)$ and prove that Equation (4.7) and (4.8) hold.

Step 1: Defining the function ITRL. Let $\mathsf{ITRL}(\ell,y)$ be the function defined as

$$\mathsf{ITRL}(\ell,\varepsilon) := \ell, \quad \mathsf{ITRL}(\ell,s_i(y)) := \mathsf{Tail}(\mathsf{ITRL}(\ell,y)) \quad (i\in\{0,1\}).$$

We need to prove that $\mathsf{ITRL}(\ell,\ell)=\varepsilon$. Indeed, we will prove a stronger result in PV-PL:

$$\vdash \mathsf{ITR}(\mathsf{ITRL}(\ell,x),\mathsf{ITR}(\ell,x))=\varepsilon,$$

which suffices as we can substitute x/ℓ.

The assertion above can be proved by induction on x using the (Ind_1) rule. The base case is straightforward. In the induction case, we need to prove that

$$\mathsf{ITR}(\mathsf{ITRL}(\ell,x),\mathsf{ITR}(\ell,x))=\varepsilon \vdash \mathsf{ITR}(\mathsf{ITRL}(\ell,s_i(x)),\mathsf{ITR}(\ell,s_i(x)))=\varepsilon,$$

Note that the consequence is PV-provably equivalent to

$$\mathsf{ITR}(\mathsf{Tail}(\mathsf{ITRL}(\ell,x)),\mathsf{TR}(\mathsf{ITR}(\ell,x)))=\varepsilon.$$

Therefore, it suffices (by the substitution/generalization rule) to prove that $\mathsf{ITR}(x,y)=\varepsilon \vdash \mathsf{ITR}(\mathsf{Tail}(x),\mathsf{TR}(y))=\varepsilon$. This can be derived from basic properties of ITR and $\vdash \mathsf{ITR}(\mathsf{Tail}(x),\mathsf{TR}(x))=\varepsilon$, which subsequently follows from a tedious but straightforward induction on x.

Step 2: Defining the function f'. We now define the function $f'(\vec{x},\ell,y)$ that (intuitively) outputs $f(\vec{x},\mathsf{ITRL}(\ell,\mathsf{ITR}(\ell,y)))$.

Let $g'(\vec{x},\ell)=g(\vec{x})$. For $i\in\{0,1\}$, we define

$$h'_i(\vec{x},\ell,y,z) := \begin{cases} g(\vec{x}) & \mathsf{IsEps}(\hat{\ell}) \\ h(\vec{x},\mathsf{Head}(\hat{\ell}),\mathsf{Tail}(\hat{\ell}),z) & \text{otherwise} \end{cases}$$

where $\hat{\ell} := \mathsf{ITRL}(\ell,\mathsf{ITR}(\ell,s_i(y)))$, and

$$k'_i(\vec{x},\ell,y) := \begin{cases} g(\vec{x}) & \mathsf{IsEps}(\hat{\ell}) \\ k(\vec{x},\mathsf{Head}(\hat{\ell}),\mathsf{Tail}(\hat{\ell}),z) & \text{otherwise} \end{cases}$$

To recursively define a function from g', h'_i, k'_i, we need to show that it is provable in PV that $\mathsf{ITR}(h'_i(\vec{x}, \ell, y, z), z \circ k'_i(\vec{x}, \ell, y)) = \varepsilon$ for $i \in \{0, 1\}$. Notice that both h'_i and k'_i are defined with a case study on the condition $\mathsf{IsEps}(\hat{\ell})$. Therefore, by case study on ITE (see Theorem 2.4.6 and Remark 2.4.2), it suffices to prove that

- $\mathsf{ITR}(g(\vec{x}), g(\vec{x})) = \varepsilon$
- $\mathsf{ITR}(h(\vec{x}, \mathsf{Head}(\hat{\ell}), \mathsf{Tail}(\hat{\ell}), z), z \circ k(\vec{x}, \mathsf{Head}(\hat{\ell}), \mathsf{Tail}(\hat{\ell}), z)) = \varepsilon$

The first equation follows from $\mathsf{ITR}(x, x) = \varepsilon$, and the second follows from the assumption.

We can then define a function $f'(\vec{x}, \ell, y)$ recursively from g', h'_i, k'_i in PV, and we define $f(\vec{x}, \ell) := f'(\vec{x}, \ell, \ell)$.

Step 3: Verifying the properties. Now it suffices to verify the properties in Equation (4.7) and (4.8) for the function f we defined above. To see that Equation (4.7), notice that PV proves

$$f(\vec{x}, \varepsilon) = f'(\vec{x}, \varepsilon, \varepsilon) = g(\vec{x})$$

by unfolding the definitions.

The proof of Equation (4.8), however, is much more complicated. We will prove that

$$f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(y))) = f'(\vec{x}, \ell, \mathsf{ITR}(\ell, y)). \tag{4.43}$$

Recall that $f'(\vec{x}, \ell, y)$ is intended to be $f(\vec{x}, \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, y)))$, and when $|y| \le |\ell|$, Equation (4.43) is intended to be

$$f(\vec{x}, \mathsf{ITRL}(\ell :: u, s_0(y))) = f(\vec{x}, \mathsf{ITRL}(\ell, y)),$$

which should be true since $\mathsf{ITRL}(\ell :: u, s_0(y)) = \mathsf{ITRL}(\ell, y)$.

Proposition 4.5.1. $\mathsf{PV} \vdash \mathsf{ITRL}(\ell :: u, s_0(y)) = \mathsf{ITRL}(\ell, y)$.

Proof. This can be proved by induction on y. The equation trivially holds when $y = \varepsilon$ by unfolding the LHS. Suppose that the equation holds for y, notice that

$$\mathsf{ITRL}(\ell :: u, s_0(s_0(y))) = \mathsf{Tail}(\mathsf{ITRL}(\ell :: u, s_0(y)))$$

and

$$\mathsf{ITRL}(\ell, s_0(y)) = \mathsf{Tail}(\mathsf{ITRL}(\ell, y)).$$

Since $\mathsf{ITRL}(\ell :: u, s_0(y)) = \mathsf{ITRL}(\ell, y)$ by the assumption hypothesis, we can prove the induction case by applying the PV-PL axioms of equality. □

Proof of Equation (4.8). We first show that Equation (4.43) suffices to prove Equation (4.8). Suppose that Equation (4.43) is true, by substituting y/ε and unfolding ITR, we can obtain that

$$f'(\vec{x}, \ell :: u, \mathsf{TR}(\ell :: u)) = f'(\vec{x}, \ell, \ell)$$

where the RHS is $f(\vec{x}, \ell)$. We will then show that

$$f(\vec{x}, \ell :: u) = h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(\ell :: u))),$$

which, together with the fact that $f'(\vec{x}, \ell :: u, \mathsf{TR}(\ell :: u)) = f(\vec{x}, \ell)$, concludes the proof of Equation (4.8).

Proposition 4.5.2. PV *proves that*

$$f'(\vec{x}, \ell :: u, y) = \begin{cases} g(\vec{x}) & \mathsf{Or}(\mathsf{IsEps}(\hat{\ell}), \mathsf{IsEps}(y)) \\ h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(y))) & \textit{otherwise} \end{cases}$$

where $\hat{\ell} := \mathsf{ITRL}(\ell :: u, \mathsf{ITR}(\ell :: u, y))$.

Proof Sketch. We perform a case study on y using the induction rule of PV-PL. This equation holds for all three cases y/ε, $y/s_0(y)$, and $y/s_1(y)$ by simple unfolding. □

Proposition 4.5.3. $\mathsf{PV} \vdash f(\vec{x}, \ell :: u) = h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(\ell :: u)))$.

Proof Sketch. Note that it is easy to prove that $\mathsf{PV} \vdash \mathsf{IsNotEps}(\ell :: u) = 1$ by unfolding the definition of Append. Moreover, let $\hat{\ell} = \mathsf{ITRL}(\ell :: u, \mathsf{ITR}(\ell :: u, \ell :: u))$, we know that $\hat{\ell} = \ell :: u$ and thus we can prove that $\mathsf{IsEps}(\hat{\ell}) = 0$. By Proposition 4.5.2 with $y/\ell :: u$ and unfolding ITE's (used to implement the case analysis), we can conclude that

$$f'(\vec{x}, \ell :: u, \ell :: u) = h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(\ell :: u))),$$

where the LHS is exactly $f(\vec{x}, \ell :: u)$ by the definition axiom of f. □

Proof of Equation (4.43). We will prove Equation (4.43) by induction on $\mathsf{ITR}(\ell, y)$. Concretely, let y' be a fresh variable, we will prove the equation where y is substituted by $\mathsf{ITR}(\ell, y')$ by induction on y'.

Note that this will lead to a proof of

$$f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, y')))) = f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, y'))) \tag{4.44}$$

that is not known to imply Equation (4.43). Nevertheless, since what we need to do is to prove the original property Equation (4.8) of the function f by substituting y/ε, the equation above also suffices with the substitution y'/ℓ instead.

Recall that $\mathsf{EQL}(x, y)$ outputs 1 (resp. 0) if and only if $|x| = |y|$ (resp. $|x| \neq |y|$). It turns out:

Proposition 4.5.4. PV-PL *proves the following assertions:*

- $\mathsf{IsNotEps}(\mathsf{ITR}(x, y)) \vdash \mathsf{EQL}(\mathsf{ITR}(x, \mathsf{ITR}(x, y)), y) = 1$.
- $\mathsf{EQL}(x, y) \vdash \mathsf{ITRL}(\ell, x) = \mathsf{ITRL}(\ell, y)$.
- $\mathsf{EQL}(x, y) \vdash f'(\vec{z}, \ell, x) = f'(\vec{z}, \ell, y)$.

Proof Sketch. All the equations can be proved by applying induction on x and y using the induction rule of PV-PL. □

We perform induction on y' to prove Equation (4.44), in which we will need to resolve the following three subgoals:

(*Subgoal* 1). $\vdash$ Equation (4.44) with substitution y'/ε.
(*Subgoal* 2). Equation (4.44) $\vdash$ Equation (4.44) with substitution $y'/s_0(y')$
(*Subgoal* 3). Equation (4.44) $\vdash$ Equation (4.44) with substitution $y'/s_1(y')$

We will deal with the three cases in the following lemmas.

Lemma 4.5.5 (*Subgoal* 1)**.** $\mathsf{PV} \vdash f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\ell))) = f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \ell))$.

Proof Sketch. Since $\mathsf{PV} \vdash \mathsf{ITR}(\ell, \ell) = \varepsilon$, the RHS of the equation is PV-provably equal to $f'(\vec{x}, \ell, \varepsilon)$, and by the definition axiom of f' it is further PV-provably equal to $g(\vec{x})$.

As for the LHS, notice that $\mathsf{PV} \vdash \mathsf{IsEps}(\mathsf{ITR}(\ell :: u, s_0(\ell))) = 0$ by unfolding $\ell :: u$ and proving basic properties about ITR. By Proposition 4.5.2, we know that

$$\begin{aligned} & f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\ell))) \\ = & \begin{cases} g(\vec{x}) & \mathsf{Or}(\mathsf{IsEps}(\hat{\ell}), \mathsf{IsEps}(\mathsf{ITR}(\ell :: u, s_0(\ell)))) \\ h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(\mathsf{ITR}(\ell :: u, s_0(\ell))))) & \text{otherwise} \end{cases} \end{aligned}$$

where $\hat{\ell} := \mathsf{ITRL}(\ell :: u, \mathsf{ITR}(\ell :: u, \mathsf{TR}(\mathsf{ITR}(\ell :: u, s_0(\ell)))))$. Note that

$$\mathsf{EQL}(\mathsf{ITR}(\ell :: u, \mathsf{ITR}(\ell :: u, s_0(\ell))), s_0(\ell)) = 1,$$

Hint: Using Proposition 4.5.4.

and the proof is left as an exercise. By $\mathsf{EQL}(x, y) = 1 \Rightarrow \mathsf{ITRL}(\ell, x) = \mathsf{ITRL}(\ell, y)$ we can further prove that

$$\hat{\ell} = \mathsf{ITRL}(\ell :: u, s_0(\ell)),$$

which is further equal to $\mathsf{ITRL}(\ell, \ell) = \varepsilon$. Therefore by unfolding the equation from Proposition 4.5.2 we have $f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\ell))) = g(\vec{x})$. □

Next, we prove (*Subgoal* 2) and (*Subgoal* 3). [6]Fix any $i \in \{0, 1\}$. Let $\hat{\ell}$ be a fresh variable, so that we can introduce $\hat{\ell} = \mathsf{ITRL}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))$ as an assumption (aka. in the antecedent). Note that by unfolding ITRL we can prove in PV that $\hat{\ell} = \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, s_i(y')))$.

We will perform a case analysis on whether $\mathsf{IsEps}(\hat{\ell}) = 1$. Concretely, let Γ be the set of antecedents including:

- $\hat{\ell} = \mathsf{ITRL}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))$
- $\hat{\ell} = \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, s_i(y')))$
- $f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, y')))) = f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, y')))$ (i.e. Equation (4.44)).

We will prove the following two lemmas:

Lemma 4.5.6. PV-PL *proves* $\Gamma, \mathsf{IsEps}(\hat{\ell}) = 1 \vdash$

$$f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))) = f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y')))).$$

[6] Introduction of a fresh variable can be done by (V), (A), and (Cut).

Lemma 4.5.7. $\mathsf{PV\text{-}PL}$ *proves* $\Gamma, \mathsf{IsEps}(\hat{\ell}) = 0 \vdash$

$$f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))) = f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y')))).$$

These two lemma suffices to prove that $\Gamma \vdash$ Equation (4.44) (with the substitution $y'/s_i(y')$ by a case study on $\hat{\ell}$, which subsequently completes the proof of all subgoals.

Proof Sketch of Lemma 4.5.6. Indeed, we will prove that the LHS and the RHS of the equation are both identical to $g(\vec{x})$.

(LHS). Note that by Proposition 4.5.2, we know that

$$\begin{aligned} & f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))) \\ = & \begin{cases} g(\vec{x}) & \mathsf{Or}(\mathsf{IsEps}(\hat{\ell}'), \mathsf{IsEps}(y'')) \\ h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(y''))) & \text{otherwise} \end{cases} \end{aligned}$$

where

$$\begin{aligned} y'' &:= \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y')))), \\ \hat{\ell}' &:= \mathsf{ITRL}(\ell :: u, \mathsf{ITR}(\ell :: u, y'')). \end{aligned}$$

Note that we can prove $\mathsf{EQL}(\mathsf{ITR}(\ell :: u, y''), s_0(\mathsf{ITR}(\ell, s_i(y')))) = 1$ (details omitted), so that by Proposition 4.5.4:

$$\hat{\ell}' = \mathsf{ITRL}(\ell :: u, \mathsf{ITR}(\ell :: u, y'')) = \mathsf{ITRL}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y')))) = \hat{\ell}.$$

Since $\mathsf{IsEps}(\hat{\ell}) = 1$ is available in the antecedent, we can apply the rewrite rule $(\triangleq_/)$, unfold Or and ITE, and conclude that

$$f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))) = g(\vec{x}).$$

(RHS). Again, by Proposition 4.5.2, we know that

$$\begin{aligned} & f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y')))) \\ = & \begin{cases} g(x) & \mathsf{Or}(\mathsf{IsEps}(\hat{\ell}'), \mathsf{IsEps}(y'')) \\ h(\vec{x}, u, \ell, f'(\vec{x}, \ell, \mathsf{TR}(y''))) & \text{otherwise} \end{cases} \end{aligned}$$

where

$$\begin{aligned} y'' &:= \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y'))), \\ \hat{\ell}' &:= \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, y'')). \end{aligned}$$

Note that we can prove $\mathsf{EQL}(\mathsf{ITR}(\ell, y''), \mathsf{ITR}(\ell, s_i(y'))) = 1$ (details omitted), so that by Proposition 4.5.4:

$$\hat{\ell}' = \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, y'')) = \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, s_i(y'))) = \hat{\ell}.$$

Since $\mathsf{IsEps}(\hat{\ell}) = 1$ is available in the antecedent, we can apply the rewrite rule $(\triangleq_/)$, unfold Or and ITE, and conclude that

$$f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y')))) = g(\vec{x}).$$

This completes the proof. □

Proof of Lemma 4.5.7. Similar to the proof of Lemma 4.5.6, we can apply Proposition 4.5.2 to show that:

$$\begin{aligned}&f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y')))))\\ &= \begin{cases} g(\vec{x}) & \mathsf{Or}(\mathsf{IsEps}(\hat{\ell}'), \mathsf{IsEps}(y'')) \\ h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(y''))) & \text{otherwise} \end{cases}\end{aligned}$$

where it is provable that $\hat{\ell}' = \hat{\ell}$, and $y'' := \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))$. It can be proved that $\mathsf{IsEps}(y'') = 0$ as $\mathsf{IsEps}(\mathsf{ITR}(\ell :: u, s_0(\ell))) = 0$. Therefore, by the assumption that $\mathsf{IsEps}(\hat{\ell}) = 0$, we can conclude by applying the rewrite rule and unfolding that:

$$\begin{aligned}&f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y')))))\\ &= h(\vec{x}, u, \ell, f'(\vec{x}, \ell :: u, \mathsf{TR}(\mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))))).\end{aligned} \tag{4.45}$$

Similarly, we can also prove that

$$\begin{aligned}&f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y'))))\\ &= h(\vec{x}, u, \ell, f'(\vec{x}, \ell, \mathsf{TR}(\mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y')))))).\end{aligned} \tag{4.46}$$

Notice that the LHS of Equation (4.45) and (4.46) are the LHS and RHS of the equation in the lemma, respectively. Therefore, it suffices to prove that

$$\begin{aligned}&f'(\vec{x}, \ell :: u, \mathsf{TR}(\mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))))\\ &= f'(\vec{x}, \ell, \mathsf{TR}(\mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y'))))).\end{aligned} \tag{4.47}$$

Finally, notice that PV proves:

$$\begin{aligned}&\mathsf{EQL}(\mathsf{TR}(\mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, s_i(y'))))), \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, y')))) = 1,\\ &\mathsf{EQL}(\mathsf{TR}(\mathsf{ITR}(\ell, \mathsf{ITR}(\ell, s_i(y')))), \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, y'))) = 1.\end{aligned}$$

(The detail is omitted.) We can therefore modify our goal Equation (4.47) using Proposition 4.5.4 and the rewrite rule to:

$$f'(\vec{x}, \ell :: u, \mathsf{ITR}(\ell :: u, s_0(\mathsf{ITR}(\ell, y')))) = f'(\vec{x}, \ell, \mathsf{ITR}(\ell, \mathsf{ITR}(\ell, y'))),$$

which is available as the third bullet of the antecedent (see the definition of Γ above). □

4.6 Proof of the Admissibility of Induction on Lists

We first recall the formal statement of the induction rule on lists:

Theorem 4.1.5 (Induciton on Lists)**.** *Let u be a variable with no occurrences in Γ, t_1, t_2, and ℓ be a variable with no occurrences in Γ. The following rule is admissible in* PV-PL*:*

$$\frac{\Gamma \vdash t_1[\ell/\varepsilon] = t_2[\ell/\varepsilon] \quad \Gamma, \mathsf{IsList}(\ell) = 1, t_1 = t_2 \vdash t_1[\ell/\ell :: u] = t_2[\ell/\ell :: u]}{\Gamma, \mathsf{IsList}(\ell) = 1 \vdash t_1 = t_2}$$

Intuitively, we will prove the theorem by induction on the length of the list ℓ using the (Ind_1) rule of $\mathsf{PV\text{-}PL}$, where the base case and the induction case will be resolved by the premises of the rule. The idea is quite similar to the proof of Theorem 4.1.3; we will prove by induction on a fresh variable y that $t_1 = t_2$ after the substitution

$$y/\mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, y)).$$

That is, $t_1 = t_2$ holds for every prefix of the list ℓ, and we prove the fact by induction on the list of the prefix.

We will prove Theorem 4.1.5 in two steps: We first define the partial list function $\mathsf{PrfList}(\ell, x) = \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, x))$ and proves its properties, and then perform the induction proof.

Step 1: Properties of partial lists. Recall that $\mathsf{ITRL}(\ell, x)$ iteratively remove the outermost element in the list ℓ for $|x|$ times, and $\mathsf{Len}(\ell)$ computes the length of the list ℓ. We can define a function $\mathsf{PrfList}(\ell, x) = \mathsf{ITRL}(\ell, \mathsf{ITR}(\ell, x))$.

Proposition 4.6.1. *The following assertions are provable in* $\mathsf{PV\text{-}PL}$*:*[7]

- $\mathsf{IsList}[\ell] \vdash \mathsf{PrfList}(\ell, \varepsilon) = \varepsilon$*;*
- $\mathsf{IsList}[\ell] \vdash \mathsf{IsList}[\mathsf{PrfList}(\ell, x)]$
- $\mathsf{IsList}[\ell] \vdash \mathsf{PrfList}(\ell, x) = \mathsf{PrfList}(\ell, s_i(x)) \lor \mathsf{PrfList}(\ell, x) = \mathsf{Tail}(\mathsf{PrfList}(\ell, s_i(x)))$*.*

Proof Sketch. The first bullet is equivalent to $\mathsf{ITRL}(\ell, \ell) = \varepsilon$, which has been proved in Step 1 of the proof of Theorem 4.1.3 (see Section 4.5).

The second bullet can be proved by induction on y on the stronger statement $\mathsf{IsList}[\ell] \vdash \mathsf{IsList}[\mathsf{ITRL}(\ell, y)]$. The base case is trivial as $\mathsf{ITRL}(\ell, \varepsilon) = \varepsilon$, while the induction case follows from the definition equation of ITRL and IsList.

For the last bullet, we first prove a case study on whether $\mathsf{ITR}(x, \ell) = \varepsilon$. If this is not the case, we have that $\mathsf{ITR}(\ell, x) = \mathsf{ITR}(\ell, s_i(x)) = \varepsilon$ and the left side of the disjunction holds by the reflexivity of equality. Otherwise, we know that

$$\mathsf{PV} \vdash \mathsf{EQL}(\mathsf{ITR}(\ell, x), s_0(\mathsf{ITR}(\ell, s_i(x)))) = 1.$$

and it suffices to prove for a fresh variable z that

$$\mathsf{ITRL}(\ell, z) = \mathsf{ITRL}(\ell, s_0(z)) \lor \mathsf{ITRL}(\ell, s_0(z)) = \mathsf{Tail}(\mathsf{ITRL}(\ell, z))$$

by the second bullet of Proposition 4.5.4. The right side of the disjunction is implied by the definition equation of ITRL. □

Step 2: Induction. Let y be a fresh variable. We will prove a stronger statement that

$$\Gamma, \mathsf{IsList}[\ell] \vdash t_1[\ell/\mathsf{PrfList}(\ell, y)] = t_2[\ell/\mathsf{PrfList}(\ell, y)]. \tag{4.48}$$

[7] The disjunction is defined in terms of → and ⊥. Note that standard inference rules are admissible, see Remark 3.2.1.

It is easy to verify that this suffices as we can make the substitution $y/\mathsf{Len}(\ell)$ so that the consequence is PV-PL provably equivalent to $t_1 = t_2$.

We prove Equation (4.48) by applying (Ind_1) on the variable y. It generates three subgoals:

- (*Subgoal 1*): $\Gamma, \mathsf{IsList}[\ell] \vdash t_1[\ell/\mathsf{PrfList}(\ell,\varepsilon)] = t_2[\ell/\mathsf{PrfList}(\ell,\varepsilon)]$.
- (*Subgoal 2*): From $\Gamma, \mathsf{IsList}[\ell], t_1[\ell/\mathsf{PrfList}(\ell,y)] = t_2[\ell/\mathsf{PrfList}(\ell,y)]$ deduce that
$$t_1[\ell/\mathsf{PrfList}(\ell,s_0(y))] = t_2[\ell/\mathsf{PrfList}(\ell,s_0(y))].$$
- (*Subgoal 3*): From $\Gamma, \mathsf{IsList}[\ell], t_1[\ell/\mathsf{PrfList}(\ell,y)] = t_2[\ell/\mathsf{PrfList}(\ell,y)]$ deduce that
$$t_1[\ell/\mathsf{PrfList}(\ell,s_1(y))] = t_2[\ell/\mathsf{PrfList}(\ell,s_1(y))].$$

We first prove (*Subgoal 1*). By the first bullet of Proposition 4.6.1, we know that $\mathsf{PrfList}(\ell,\varepsilon) = \varepsilon$, and thus it suffices to prove $t_1[\ell/\varepsilon] = t_2[\ell/\varepsilon]$, which follows from the first premise of the rule.

It remains to prove the (*Subgoal 2*) and (*Subgoal 3*). Let $i \in \{0,1\}$, we need to prove that

$$t_1[\ell/\mathsf{PrfList}(\ell,s_i(y))] = t_2[\ell/\mathsf{PrfList}(\ell,s_i(y))] \tag{4.49}$$

from $\Gamma, \mathsf{IsList}[\ell], t_1[\ell/\mathsf{PrfList}(\ell,y)] = t_2[\ell/\mathsf{PrfList}(\ell,y)]$. By the last bullet of Proposition 4.6.1, we can perform a case study on whether

$$\mathsf{PrfList}(\ell,x) = \mathsf{PrfList}(\ell,s_i(x)) \text{ or } \mathsf{PrfList}(\ell,x) = \mathsf{Tail}(\mathsf{PrfList}(\ell,s_i(x))).$$

In the former case, we can prove Equation (4.49) as the equation is equivalent to $t_1[\ell/\mathsf{PrfList}(\ell,y)] = t_2[\ell/\mathsf{PrfList}(\ell,y)]$, which is available in the antecedent.

In the latter case, we first prove that $\mathsf{IsList}[\mathsf{PrfList}(\ell,y), \mathsf{PrfList}(\ell,s_i(y))]$ by the second bullet of Proposition 4.6.1. Then it follows that

$$\begin{aligned} & \mathsf{PrfList}(\ell,s_i(x)) \\ = {} & \mathsf{Tail}(\mathsf{PrfList}(\ell,s_i(x))) :: \mathsf{Head}(\mathsf{PrfList}(\ell,s_i(x))) \\ = {} & \mathsf{PrfList}(\ell,x) :: \mathsf{Head}(\mathsf{PrfList}(\ell,s_i(x))), \end{aligned}$$

where the first equality follows from the definition equation of $\mathsf{IsList}[\mathsf{PrfList}(\ell,s_i(x))]$, and the second equality follows from that $\mathsf{PrfList}(\ell,x) = \mathsf{Tail}(\mathsf{PrfList}(\ell,s_i(x)))$.

By the cut rule, we can add this equation to the antecedent, and thus it suffices to prove from

$$\Gamma, \mathsf{IsList}[\mathsf{PrfList}(\ell,y)], t_1[\ell/\mathsf{PrfList}(\ell,y)] = t_2[\ell/\mathsf{PrfList}(\ell,y)]$$

that

$$\begin{aligned} & t_1[\ell/\mathsf{PrfList}(\ell,x) :: \mathsf{Head}(\mathsf{PrfList}(\ell,s_i(x)))] \\ = {} & t_2[\ell/\mathsf{PrfList}(\ell,x) :: \mathsf{Head}(\mathsf{PrfList}(\ell,s_i(x))]). \end{aligned}$$

Pick fresh variable $\hat{\ell}$ and u. By the substitution/generalization rule of PV-PL, it then suffices to prove that:

$$\Gamma, \mathsf{IsList}[\hat{\ell}], t_1[\ell/\hat{\ell}] = t_2[\ell/\hat{\ell}] \vdash t_1[\ell/\hat{\ell} :: u] = t_2[\ell/\hat{\ell} :: u].$$

This can be proved from the second premise with the substitution $\ell/\hat{\ell}$, which completes the proof.

4.7 Bibliographical and Other Remarks

Feasible Mathematics Thesis. In the previous four chapters, we have worked hard to justify the *Feasible Mathematics Thesis*, namely PV captures the informal notion of feasible mathematics we defined with the three postulates. We provide evidence that PV is strong enough to simulate known feasible algorithms and carries known feasibly constructive proofs over feasible algorithms:

- In Chapter 2, we developed if-then-else, basic Boolean logic, the pairing function, conditional equations, and extensions on recursion and induction. The pairing function allows us to work with structured data rather than raw strings, and the extensions on recursion allow us to define, for instance, arithmetic operations and the function EQ.

As computer scientists, we always love abstraction :)

- In Chapter 3, we developed a natural deduction style proof system called PV-PL that allows us to perform conditional reasoning, propositional reasoning, as well as a general form of induction (similar to Postulate 3) that is more general and easier to use. Note that the function EQ, the method of conditional equations, as well as induction on multiple variables, are the key ingredients empowering the translation theorem (see Theorem 3.3.4) to PV proofs.
- In this chapter, we developed lists and dictionaries, which further extend our ability to define new functions by recursion and performing induction on the recursively defined function. The meta-theorems (Theorem 4.1.3 and 4.1.6) effectively allow us to deal with lists just as strings, for which the recursion and induction rules are built-in functionalities of PV. We highlight that in the proof of these two meta-theorems, we heavily rely on the PV-PL system and conditional reasoning; the proof will be much harder without the system PV-PL as an abstract layer.

Trust me, as in the first draft I tried to prove Theorem 4.1.3 without PV-PL, and it's not a good experience :(

- We further provide simulations of other computation models, namely an imperative programming language IMP(PV) built upon PV and (single-tape) Turing machines in PV. Moreover, we show that the Hoare logic over IMP(PV) programs, which serves as a natural and intuitive tool to reason about IMP(PV) programs, can be translated back to PV-PL (and thus PV) proofs. We also show that the elementary first-order theory of finite sets, when sets are explicitly given as lists, can be formalized in PV-PL using the meta-theorems for lists developed in Section 4.1.

Given the robustness of PV as demonstrated in previous chapters, we will put more trust in the Feasible Mathematics Thesis in writing PV functions and proofs. We will write proofs in informal feasible mathematics like computer scientists and programmers pseudo-codes (rather than Turing machines or real programs), and as mathematicians writing informal mathematical formulations and proofs (rather than writing proofs in axiomatized set theory).

Lists and maps. The constructions of lists and maps are implicit in, e.g., [Bus86, Kra95a]. To our knowledge, the recursion meta-theorem (see Theorem 4.1.3) and induction meta-theorem (see Theorem 4.1.5) have not been explored in the literature; nevertheless, both theorems are widely believed to be true.

It is worth noting that these meta-theorems can be viewed as time-bounded versions of the recursion and induction functionalities for *inductively defined types* in modern proof assistants, see, e.g., [DMKA+15, Section 3].

Imperative programming language IMP. The imperative programming language IMP(PV) is an abstraction of modern structured programming languages. Similar languages are popular in compiler and formal verification textbooks, see, e.g., [PCG+10, ImpCEvalFun]. One major difference between our and the standard formulation is that we require an explicit length bound for all variables, which ensures that the program runs in polynomial time.

A similar imperative programming language was introduced by Cook [Coo90] to define high-order polynomial-time functions. Cook [Coo90] did not develop proof systems for the functionality of programs.

Feasible set theory. The universal and existential quantification over feasible sets can be viewed as a "typed" version of Buss's sharply bounded quantifiers [Bus86], as the numbers of elements in sets encoded by lists are bounded by their encoding lengths. All results in Section 4.4 are widely believed to be true, and we view them as a formalization of the folklore intuition that "almost all standard mathematics on small sets can be formalized in PV".

A recent work of Beckmann, Buss, Friedman, Müller, and Thapen [BBF+19] developed a feasible set theory that combines the axiomatic set theory and complexity theory. Their results extend to infinite sets such as ω.

Chapter 5

Connection to Propositional Proofs

We will now answer a question posted in Chapter 1. Recall that feasible mathematics is defined as an interpretation of constructive mathematics (i.e. the BHK interpretation), where the "effective procedures" are interpreted by feasible functions, i.e., polynomial-time computable functions. However, the meaning of "proofs" remains unspecified.

We will introduce the interpretation of "proofs" in [Coo75], which is known as Cook's translation, that provides a tight connection between propositional proof systems and feasible mathematics.

5.1 Cook's Translation and Interpretation of Proofs

We first explain our goal — the interpretation of proofs in BHK interpretation — in more detail. Suppose that $\varphi(\vec{x})$ is a PV-PL formula, a proof of $\varphi(\vec{x})$ must correspond to an efficient procedure that given any $\vec{x}$, outputs a "proof" of $\varphi(\vec{x})$.

Equivalently, one may consider equations in PV, which is essentially identical by the $[\cdot]_{\mathsf{PV}}$ translation.

Intuition of the Cook's interpretation. To further clarify the question, we consider the meaning of "$\varphi(\vec{x})$". Let $\vec{x} = (x_1, \dots, x_k)$. Cook [Coo75] interprets $\varphi(\vec{x})$ as a family of *propositional formulas*, where for $n_1, \dots, n_k \in \mathbb{N}$,

$$[\varphi(\vec{x})]_{\mathsf{Cook}}^{n_1,\dots,n_k}$$

is a propositional formula consisting of $n_1 + n_2 + \cdots + n_k$ Boolean variables that simulates the formula $\varphi(\vec{x})$ when $x_1, \dots, x_k$ are of length $n_1, \dots, n_k$, respectively.

Such propositional formulas can be constructed by the Cook-Levin theorem; this ensures that the size of the formula $[\varphi(\vec{x})]_{\mathsf{Cook}}^{n_1,\dots,n_k}$ is at most $\mathsf{poly}(n_1, \dots, n_k)$. Indeed, this translation is essentially the same as the translation of uniform polynomial-time Turing machines to polynomial-sized circuits.

Therefore, we can naturally interpret proofs of $\varphi(\vec{x})$ as a family of *propositional proofs* of the formula $[\varphi(\vec{x})]_{\mathsf{Cook}}^{n_1,\dots,n_k}$ in some propositional proof systems P. Following the BHK interpretation, we would like to prove a meta-theorem:

Theorem 5.1.1 (Cook's interpretation of proofs, informal)**.** *The following holds for a (sound and complete) propositional proof system P. Suppose that $\varphi(\vec{x})$ is provable in*

$\mathsf{PV\text{-}PL}$*, there is* PV*-function* Gen_φ *such that* $\mathsf{PV\text{-}PL}$ *proves that* $\mathsf{Gen}_\varphi(z_1, \dots, z_k)$ *outputs a correct propositional proof of* $[\varphi(x)]_{\mathsf{Cook}}^{|z_1|,\dots,|z_k|}$ *in* P*.*

As an immediate consequence, we know that if $\varphi(\vec{x})$ is provable in $\mathsf{PV\text{-}PL}$, the propositional formula $[\varphi(x)]_{\mathsf{Cook}}^{n_1,\dots,n_k}$ admits P-proof of size $\mathsf{poly}(n_1, \dots, n_k)$, as Gen_φ is a feasible function.

We stress that the statement "$\mathsf{PV\text{-}PL}$ proves that $\mathsf{Gen}_\varphi(z_1, \dots, z_k)$ outputs ..." needs to be formalized as a $\mathsf{PV\text{-}PL}$ sentence. Nevertheless, as long as P is a standard propositional proof system, there will be a straightforward feasible algorithm verifying $\mathsf{Verifier}_P(\psi, \pi)$ whether a proof π is a correct proof of a formula ψ or not. This statement can therefore be formalized as the PV equation

$$\mathsf{Verifier}_P([\varphi(x)]_{\mathsf{Cook}}^{|z_1|,\dots,|z_k|}, \mathsf{Gen}_\varphi(z_1, \dots, z_k)) = 1,$$

where the function $(z_1, \dots, z_k) \mapsto [\varphi(x)]_{\mathsf{Cook}}^{|z_1|,\dots,|z_k|}$ can also be implemented by a straightforward PV function.

Theorem 5.1.1 shows that $\mathsf{PV\text{-}PL}$ can be simulated by a uniform family of (short) propositional proofs in P. This could be a trivial statement if P is too strong; for instance, we can even define a proof π of ψ in P as a tuple $\pi := (\varphi(x), \pi_{\mathsf{PV}})$, and the proof is correct if $\psi = [\varphi(x)]_{\mathsf{Cook}}^n$ and π_{PV} is a PV-proof of ψ. Therefore, a complement of Theorem 5.1.1 must be proved to show that P is not too strong:

Theorem 5.1.2 (Cook's interpretation of proofs, strengthened)**.** *The followings are equivalent for a (sound and complete) propositional proof system* P*.*

- $\varphi(\vec{x})$ *is provable in* $\mathsf{PV\text{-}PL}$*;*
- $\mathsf{PV\text{-}PL}$ *proves that for any* $z_1, \dots, z_k$*,* $\mathsf{Gen}_\varphi(z_1, \dots, z_k)$ *outputs a correct propositional proof of* $[\varphi(\vec{x})]_{\mathsf{Cook}}^{|z_1|,\dots,|z_k|}$*.*

In some sense, this is similar to the standard result in complexity theory that $\mathsf{P} = \mathsf{P\text{-}uniform}\ \mathsf{P}_{/\mathsf{poly}}$ that connects feasible algorithms with feasibly generated families of circuits.

Extended Frege system. The propositional proof system P in Theorem 5.1.2 is indeed the Extended Frege system, denoted by EF. Extended Frege EF consists of a finite set of axiom schemes as well as two rules:

- (*Modes Ponens*). From $\varphi, \varphi \to \psi$, deduce ψ.
- (*Extension*). Let z be a fresh variable that does not appear in previous lines of the proof. Deduce $z \leftrightarrow \varphi$.

Note that the primary reason for introducing the extension rule is to allow us formalizing and reasoning about *circuits* rather than only *formulas* without significant blowup on the length of the sentences and proofs.

As a concrete example, we can define EF using the following set of axiom schemes that is known to be sound and complete for propositional logic:

- $\mathsf{K} : \varphi \to \psi \to \varphi$;
- $\mathsf{S} : (\varphi \to \psi \to \gamma) \to (\varphi \to \psi) \to \varphi \to \gamma$;

- $\mathsf{D} : \neg\neg\varphi \to \varphi$, where $\neg\varphi$ is short for $\varphi \to \bot$.

Similar to the definition of PV-PL, we can consider propositional formulas using only $\{\to, \bot\}$ as connectives, while other connectives such as $\wedge$ and $\vee$ can be introduced naturally. Alternatively, we can define formulas to have more connectives (e.g. $\to, \bot, \wedge, \vee$) by introducing corresponding axioms (see, e.g., Remark 3.2.1).

Remark 5.1.1. Why do we need EF while translating PV proofs? Specifically, why is the *extension* rule crucial? It turns out that the induction rule of PV (more generally, the induction postulate of informal feasible mathematics) requires intermediate variables to record the intermediate computation during the induction procedure. In particular, if we remove the extension rules, we can only argue about formulas, which corresponds to the complexity class NC_1 that is unlikely to be equal to P.

This should be more clear in the proof sketch of Cook's translation theorem; see Section 5.3 for more details.

5.2 Formal Definitions

Now we formally define Cook's translation, the encoding of Extended Frege proofs in PV, and Theorem 5.1.2.

5.2.1 Translation from PV Equations to Propositional Formulas

We first formally define Cook's translation from PV equations to propositional formulas; that is, for any PV equation $s = t$ with k variables, we will define a PV function $z_1, \dots, z_k \mapsto [s = t]_{\mathsf{Cook}}^{|z_1|,\dots,|z_k|}$. The translation of PV-PL formulas can be obtained by the $[\cdot]_{\mathsf{PV}}$ translation. For simplicity, we will describe the translation without formalizing it in PV; nevertheless, the formalization should be clear given the programming functionalities we have developed by now.

Alternatively, one can define the translation of PV-PL formulas directly.

Indeed, formulas obtained from the translation $[\cdot]_{\mathsf{Cook}}$ will be of form $\wedge\Gamma \to \wedge\Delta$, where Γ and Δ are sets of formulas that contain at most three variables.

Specification of the translation. We first define the translation of PV functions and terms. Formally, the translation of a PV-function $f(x_1, \dots, x_k)$ on input lengths $n_1, \dots, n_k$ is a formula $[f]_{\mathsf{Cook}}^{n_1,\dots,n_k}$ consisting of $\mathsf{poly}(n_1, \dots, n_k)$ variables, where

- There are $n_1 + \cdots + n_k$ variables $x_1 \in \{0,1\}^{n_1}, \dots, x_k \in \{0,1\}^{n_k}$ that takes the input to the function f.
- There are m variables $y_1, \dots, y_m$ defined to carry out the internal computation of the function f. Note that m depends on the function f, while for every fixed PV function f, $m = m(\vec{n})$ is a polynomial in $n_1, \dots, n_k$.
- There are ℓ variables $z \in \{0,1\}^\ell$ denoting the output of f. Similar to the previous case, $\ell = \ell(\vec{n})$ is a polynomial in $n_1, \dots, n_k$ depending on f.
- A set Γ of formulas containing at most three variables, where $[f]_{\mathsf{Cook}}^{n_1,\dots,n_k} = \wedge\Gamma$.

- The translation satisfies that for any assignment of the variables, if all formulas in Γ are satisfied, z must be the output of $f(x_1, \dots, x_k)$.

The translation of terms can be defined accordingly.

One technical issue that is worth mentioning is that the output length of a PV function is not necessarily the same even if the input length is fixed. Therefore, the translation must deal with the issue of encoding strings of different lengths by a fixed number of variables. We will use the following simple encoding: Strings of length at most m are encoded by $m+1$ variables, where the encoding of $x \in \{0,1\}^{\leq m}$ is $0^{m+1-|x|-1} \circ 1 \circ x$.

This also proves $\sum_{i \leq m} 2^i \leq 2^{m+1}$:)

Remark 5.2.1. An advantage of the encoding is that to convert the n-bit encoding of a string to a $(n+1)$-bit encoding of a string, we only need to add a leading zero to the encoding, which can be implemented by introducing an additional variable z and a constraint $\neg z$.

Description of the translation. Now we went through the formulation rules of functions and terms to define the translation. Let $\vec{n} = (n_1, \dots, n_k)$ be the input length of variables $x_1, \dots, x_k$. The translation of functions and terms is defined by structural induction on the formulation of formulas and terms (in the meta-theory).

That is, the output is the same as the input.

- (*Variable*). The translation of a variable $[x_i]^{\vec{n}}_{\mathsf{Cook}}$ is defined as $m := 0$, $\ell := n_i$, $\Gamma := \{z_j \to x_{ij} \mid j \in [n_i]\} \cup \{x_{ij} \to z_j \mid j \in [n_i]\}$.
- (*Constants*). The translation of the constant ε is defined as $m := 0$, $\ell := 1$, $\Gamma := \{z_1\}$. That is, ε is encoded as the string 1.
- (*Composition*). Suppose that f is a d-variant function and $t_1, t_2, \dots, t_d$ be terms such that we have already obtained the translation of f, $t_1, \dots, t_d$. The translation of the term $f(t_1, \dots, t_k)$ is obtained as follows:

  Well, it's just composition... The only job is to define new variables for the output of $t_1, \dots, t_d$.

  - Let $\ell_f, \ell_1, \dots, \ell_d$ be the output length of $f, t_1, \dots, t_d$, respectively. The output length of the composition is $\ell(\vec{n}) = \ell_f(\ell_1(\vec{n}), \dots, \ell_d(\vec{n}))$.
  - Let $m_f, m_1, \dots, m_d$ be the number of internal variables of $f, t_1, \dots, t_d$, respectively. The number of internal variables of the composition is

    $$m(\vec{n}) := m_f(\ell_1(\vec{n}), \dots, \ell_d(\vec{n})) + \sum_{i \in [d]} \ell_i(\vec{n}) + \sum_{i \in [d]} m_i(\vec{n}).$$

    Note that $\ell_1 + \cdots + \ell_d$ variables $p_1 \in \{0,1\}^{m_1}, \dots, p_d \in \{0,1\}^{m_d}$ are used to maintain the output of $t_1, \dots, t_d$ (see the second term).
  - Let $\Gamma_1, \dots, \Gamma_d$ are the constraints for $t_1, \dots, t_d$, respectively, on input length $\vec{n}$. Let Γ_f be the constraints of f on input length $\ell_1(\vec{n}), \dots, \ell_d(\vec{n})$. We identify the input variables of the translation of f on input length $\ell_1(\vec{n}), \dots, \ell_d(\vec{n})$ and the new internal variables $p_1, \dots, p_d$. The constraints for the composition are $\Gamma = \cup_{i \in [d]} \Gamma_i \cup \Gamma_f$.
- (*Non-Recursive Initial Functions*). Next, we consider the translation of initial functions $s_0(x), s_1(x)$ that are not recursively defined.

  - The translation $[s_0(x_i)]^{\vec{n}}_{\mathsf{Cook}}$ is defined as $m := 0$, $\ell := n_i + 1$, and $\Gamma := \{z_{j+1} \to x_{ij} \mid j \in [n_i]\} \cup \{x_{ij} \to z_{j+1} \mid j \in [n_i]\} \cup \{\neg z_1\}$.
  - The translation $[s_1(x_i)]^{\vec{n}}_{\mathsf{Cook}}$ is defined as $m := 0$, $\ell := n_i + 1$, and $\Gamma := \{z_{j+1} \to x_{ij} \mid j \in [n_i]\} \cup \{x_{ij} \to z_{j+1} \mid j \in [n_i]\} \cup \{z_1\}$.

- (*Functions via Composition*). For a function f_t defined by a term t, the translation of f_t is defined as the translation of the term t.

- (*Functions via Recursion*). Now we consider a function $f(\vec{x}, y)$ that is recursively defined from $g(\vec{x})$ and $h_i(\vec{x}, y, z)$ $(i \in \{0,1\})$.

  This includes the initial functions $\mathsf{TR}(x)$ (where $g = \varepsilon$ and $h_i(x,y,z) = y$), $\mathsf{ITR}(x,y)$ (where $g(x) = x$ and $h_i(x,y,z) = \mathsf{TR}(z)$), and $\circ(x,y)$ (where $g(x) = x$ and $h_i(x,y,z) = s_i(z)$), as well as functions introduced by the rule of limited recursion. (Note that $\#(x,y)$ is redundant, see Remark 2.4.1). In either case, we can assume that we have already obtained the translation of g, h_0, and h_1. Also, we assume that PV proves that $\mathsf{ITR}(h_i(\vec{x}, y, z), z \circ k_i(x,y)) = \varepsilon$, $i \in \{0,1\}$, for some PV function k_i whose translation is known.

This is essentially the only non-trivial step, but it's just as simple as unwinding a for-loop :)

  - Let (m_g, ℓ_g, Γ_g) be the translation of g, $(m_h^i, \ell_h^i, \Gamma_h^i)$ be the translation of h_i, and $(m_k^i, \ell_k^i, \Gamma_k^i)$ be the translation of k_i. For simplicity, we only consider the case that $\vec{x} = x$, i.e., there is only one additional variable.
  - Suppose that we are translating to the input length n_x for x and n_y for y of the function $f(x,y)$. Let $x_1, \dots, x_{n_x} \in \{0,1\}$ and $y_1, \dots, y_{n_y} \in \{0,1\}$ be the Boolean variables for the input (x_1, y_1 are the leftmost bits).
  - (*Parse Input Length*). We introduce variables $w_1, w_2, \dots, w_{n_y} \in \{0,1\}$ intended to be $w_i = 1$ if and only if y is of form $0^{i-1} \circ 1 \circ y'$; this is used to determine the actual input length of y. We need to introduce $O(n_y)$ additional variables and constraints for each $i \in [n_y]$ to implement w_i.
  - (*Recursion*). For each $i \in [n_y]$, we will introduce internal variables $z_0^i, z_1^i, \dots, z_{n_y}^i$ where

$$z_0^i \text{ maintains the output of } g(x) \tag{5.1}$$
$$z_1^i \text{ maintains the output of } h_{y_{i+1}}(x, \varepsilon, z_0^i) \tag{5.2}$$
$$z_2^i \text{ maintains the output of } h_{y_{i+2}}(x, y_{i+1}, z_1^i) \tag{5.3}$$
$$\vdots \tag{5.4}$$
$$z_{n_y-i}^i \text{ maintains the output of } h_{y_{n_y}}(x, y_{i+1} \circ \cdots \circ y_{n_y-1}, z_{n_y-i-1}^i). \tag{5.5}$$

  In addition, we introduce necessary internal variables to carry out the corresponding computation of $g(x)$, $h_{y_{i+1}}(x, \varepsilon, z_0)$, ... The total number of internal variables introduced is at most

$$\begin{aligned} m^i_f(n_x,n_y) := {}& m_g(n_x) + \zeta_0 && (g(x) \text{ and } z^i_0) \\ &+ \sum_{i\in\{0,1\}} m^i_h(n_x,0,\zeta_0) + \zeta_1 && (h_{y_{i+1}}(x,\varepsilon,z_0) \text{ and } z^i_1) \\ &+ \sum_{i\in\{0,1\}} m^i_h(n_x,1,\zeta_1) + \zeta_2 && (h_{y_{i+2}}(x,y_1,z_1) \text{ and } z^i_2) \\ &+ \cdots + \sum_{i\in\{0,1\}} m^i_h(n_x,n_y,\zeta_{n_y-1}) + \zeta_{n_y-i} \\ &&& (h_{y_{n_y}}(x,y_{1+1}\circ\cdots\circ y_{n_y-1},z^i_{n_y-i-1}) \text{ and } z^i_{n_y}) \end{aligned}$$

where ζ_j is an upper bound of the length of z^i_j, which is determined by $\zeta_0 := \ell_g(n_x)$ and $\zeta_{j+1} := \zeta_j + \ell^0_k(n_x,j) + \ell^1_k(n_x,j)$. Clearly, $m_f(n_x,n_y) \le \mathsf{poly}(n_x,n_y)$.

WLOG, we assume that $\zeta_{j+1} \ge \zeta_j$.

- (*Collect Output*). Let $\ell_f(n_x,n_y) = \zeta_{n_y}$ and $\hat{z} \in \{0,1\}^{\zeta_{n_y}}$ be the output variables. We add additional variables and constraints such that
$$\hat{z}_j = \bigvee_{i\in[n_y]} (w_i \wedge z^i_j).$$
This requires $O(n_y)$ variables and constraints for each $j \in [\zeta_{n_y}]$.
- (*Internal Variables*). The total number of internal variables introduced in the translation is at most
$$m_f(n_x,n_y) = O(n_y^2) + \sum_{i\in[n_y]} m^i_f(n_x,n_y) + O(n_y\cdot\zeta_{n_y}) = \mathsf{poly}(n_x,n_y).$$
The first term counts the variables for defining $w_1,\dots,w_{n_y}$, the second term counts the number of variables for $z^i_0,\dots,z^i_{n_y-i}$, and the last term counts the number of variables to collect the output (i.e. $\hat{z}_1,\dots,\hat{z}_{\zeta_{n_y}}$).
- (*Constraints*). The set of constraints Γ_f is defined as the constraints corresponding to Equation (5.1) to Equation (5.5), the constraints to define $w_1,\dots,w_{n_y}$, and the constraints to collect the output. The total number of constraints is bounded by $\mathsf{poly}(m_f(n_x,n_y)) = \mathsf{poly}(n_x,n_y)$.

Translation of equations. Finally, we can consider the propositional translation of equations. Let $s(\vec{x})$ and $t(\vec{x})$ be terms and $\vec{n}$ be the input lengths. We first obtain the translation $(m_s,\ell_s,\Gamma_s) := [s(\vec{x})]^{\vec{n}}_{\mathsf{Cook}}$ and $(m_t,\ell_t,\Gamma_t) := [t(\vec{x})]^{\vec{n}}_{\mathsf{Cook}}$. Assume, without loss of generality, that $\ell_s(\vec{n}) \le \ell_t(\vec{n})$. Let $z^{(s)} \in \{0,1\}^{\ell_s(\vec{n})}$ be the output variables for s and $z^{(t)} \in \{0,1\}^{\ell_t(\vec{n})}$ be the output variables for t.

Additional variables should also be introduced to deal with the $0^i\circ 1\circ x$ encoding of strings, which we ignore here.

Let $z^{(s)}_i$ and $z^{(t)}_j$ be the i-th bit of $z^{(s)}$ and $z^{(t)}$, where the rightmost bit of the first bit, respectively. The translation of the equation $s(\vec{x}) = t(\vec{x})$ is defined as the formula $\wedge\Gamma \to \wedge\Delta$, where

$$\begin{aligned} \Gamma &:= \Gamma_s \cup \Gamma_t; \\ \Delta &:= \{z^{(s)}_i \to z^{(t)}_i \mid i \in [m_s(\vec{n})]\} \cup \{z^{(t)}_i \to z^{(s)}_i \mid i \in [m_s(\vec{n})]\} \\ &\quad\ \cup \{\neg z^{(t)}_j \mid m_s(\vec{n}) < j \le m_t(\vec{n})\}. \end{aligned}$$

The set of constraints Γ encodes the computation of $s(\vec{x})$ and $t(\vec{x})$, while Δ encodes the fact that the output of $s(\vec{x})$ is the same as the output of $t(\vec{x})$.

5.2.2 Formalizing EF in PV

Propositional formulas and EF proofs can be naturally formalized in PV. The exact formalization is usually not important as natural formalizations are usually PV-provably equivalent; we provide the following encoding for concreteness.

- (*Formulas*). We formalize a formula φ as a list ℓ of natural numbers, where for each $u \in \ell$, it denotes "$\bot$" if $u = 0$, or "$\rightarrow$" if $u = 1$, or a variable of ID u otherwise. The list is the prefix expression of φ according to the encoding above, and it is easy to define (by recursion on lists using Theorem 4.1.3) functions that decide what is the outermost connective and (in case that it is $\rightarrow$) what are the left and right sides of it. For simplicity, we use $[\varphi]$ to denote the encoding of a formula φ.

- (*Proofs*). We formalize a EF proof as a list ℓ, where each element denotes a line of the proof encoded by a tuple $(\tau, [\phi_0], [\phi_1], [\phi_2])$ for $\tau \in [5]$ and ϕ_0, ϕ_1, ϕ_2 be encoding of formulas.

  - The MP rule (i.e. $\varphi, \varphi \rightarrow \psi \vdash \psi$) is encoded by $(1, [\varphi], [\psi], \bot)$.
  - The extension rule (i.e. $z \leftrightarrow \varphi$ for a fresh variable z) is encoded by $(2, [z], [\varphi], \bot)$.
  - The axiom K (i.e. $\varphi \rightarrow \psi \rightarrow \varphi$) is encoded $(3, [\varphi], [\psi], \bot)$.
  - The axioms S, D are encoded similar to the axiom K. The outermost element in the list ℓ is the last line of the proof.

- (*Conclusion*). We can define a function $\mathsf{Conc}(\pi)$ that takes a tuple π encoding a line of the proof and outputs the conclusion. For instance, $\mathsf{Conc}((3, [p], [q]))$ outputs $[p \rightarrow q \rightarrow p]$. Similarly, the conclusion of a proof ℓ, also denoted by $\mathsf{Conc}(\ell)$, is defined to be the conclusion of the last line of the proof.

- (*Pattern Matching*). For each axiom scheme $\sigma \in \{\mathsf{K}, \mathsf{S}, \mathsf{D}\}$, we can define a pattern matching function $\mathsf{Match}_\sigma(\varphi)$ that checks whether φ satisfies the pattern of the axiom scheme σ. It returns ε if it fails, and returns a tuple denoting the sub-formulas matched for each symbol of the axiom scheme σ; for instance, $\mathsf{Match}_\mathsf{K}([p \rightarrow (q \rightarrow p) \rightarrow p])$ will return a pair $([p], [q \rightarrow p])$.

- (*Proof Verification*). We can define a function $\mathsf{Verifier}(\ell)$ that checks whether the EF proof is valid by recursion on the list ℓ (see Theorem 4.1.3). Concretely, we have that:

  - An empty list ε is a valid proof.
  - If a proof ℓ is valid and u encodes a new line, we first verify that u is a correctly encoded line of proof, and then verify accordingly. For instance, if $u = (1, [\varphi], [\psi], \bot)$, the proof $\ell :: u$ is valid if both $[\varphi \rightarrow \psi]$ and $[\varphi]$ are conclusions of existing lines of proofs in ℓ.

5.3 Arithmetic Proofs to Propositional Proofs

As the first half of Cook's translation theorem (see Theorem 5.1.2), we will need to prove that an arithmetic proof of $\varphi(\vec{x})$ in PV-PL implies the existence of short propositional proofs of $[\varphi(\vec{x})]^{\vec{n}}_{\mathsf{Cook}}$, and such existence of short propositional proofs can be proved in PV-PL. Formally:

Lemma 5.3.1. *Let $s(\vec{x}) = t(\vec{x})$ be a PV equation, where $\vec{x} = (x_1, \dots, x_k)$ be the variables. Suppose that PV proves $s(\vec{x}) = t(\vec{x})$, then there exists a PV function $\mathsf{Gen}_{s=t}$ such that PV-PL proves*

$$\ell = \mathsf{Gen}_{s=t}(z_1, \dots, z_k) \vdash \mathsf{Verifier}(\ell) = 1 \land \mathsf{Conc}(\mathsf{Head}(\ell)) = [s = t]^{|z_1|,\dots,|z_k|}_{\mathsf{Cook}}.$$

That is, $\mathsf{Gen}_{s=t}(z_1, \dots, z_k)$ produces an EF proof of $[s = t]^{|z_1|,\dots,|z_k|}_{\mathsf{Cook}}$.

The proof of the lemma is tedious but straightforward, so we will only sketch the proof. Intuitively, we will perform induction on the PV proof of $s(\vec{x}) = t(\vec{x})$; that is, we translate the PV proof line-by-line into short propositional proofs. We need to consider the case that the next line to translate is a definition axiom, a logical rule, or the induction rule.

Indeed, what we will exactly prove is that suppose the translations of the first $i - 1$ lines on *any* input length $\vec{m}$ admit $\mathsf{poly}(\vec{m})$-size proofs, then the translation of the i-th line on *any* input length $\vec{m}'$ also admits a $\mathsf{poly}(\vec{m})$-size proof. This induction is *infeasible* as it is unclear how to verify whether the translation works on *any* input length. This is not a problem as the induction is done in meta-theory. Moreover, the construction of the function $\mathsf{Gen}_{s=t}$ can be easily extracted from the inductive proof, which we will omit here.

Sorry to hear that you cannot understand this, real feasible mathematicians :(

Definition axioms. The definition axiom of functions introduced via composition trivially admits short EF proofs, thus it suffices to consider functions $f(\vec{x}, y)$ introduced via recursion. This includes the initial functions $\mathsf{TR}(x)$, $\mathsf{ITR}(x, y)$, and $\circ(x, y)$, as well as other functions introduced by the rule of limited recursion. In either case, we may assume that:

- $f(\vec{x}, y)$ is introduced by $g(\vec{x})$ and $h_i(\vec{x}, y, z)$ for $i \in \{0, 1\}$.
- It is provable in PV that $\mathsf{ITR}(h_i(\vec{x}, y, z), z \circ k_i(\vec{x}, y)) = \varepsilon$.

We need to show that the translations of the definition axioms

$$f(\vec{x}, \varepsilon) = g(\vec{x}), \quad f(\vec{x}, s_i(y)) = h_i(\vec{x}, y, f(\vec{x}, y))$$

admit short EF proofs. For simplicity, we will only sketch the proof.

- (*Base Case*). The base case $f(\vec{x}, \varepsilon) = g(\vec{x})$ can be proved in EF as follows. Recall that the translation of $f(\vec{x}, \varepsilon)$ is the composition of the translation of $f(\vec{x}, y)$ and ε. In the translation of $f(\vec{x}, y)$, we introduce variables $w_1, \dots, w_{n_y}$ to determine the input length of y; here, we can prove in EF that $w_{n_y} = 1$ and $w_j = 1$ for $j \neq n_y$, i.e., the input length is 0.

In such case, we can further prove that the output of $f(\vec{x}, \varepsilon)$ is determined by the internal variable $z_0^{n_y}$, which is defined to maintain the output of $g(\vec{x})$. This implies a proof of the equation, as the RHS is also $g(\vec{x})$.

- (*Recursion Case*). Fix $i \in \{0,1\}$. The first step of the proof is to show that the length of $s_i(y)$ is the length of y plus 1. Formally, let $w_1, \ldots, w_{n_y}$ be the variables corresponding to the input length for the function f in the LHS and $w'_1, \ldots, w'_{n_y}$ be the variables for f in the RHS, we will prove that $w_i \leftrightarrow w'_{i+1}$. For each possible input length $i \in [n_y]$, we can see that the LHS and the RHS are following the same computation procedure, and thus we can prove that the output of them are identical.

I'm being a bit sloppy on the boundary; hopefully you would agree that it's fine :)

Remark 5.3.1. With careful inspection, we can see that for this case, the proof can be formalized in the Frege system (i.e. without using the extension rule).

It may be instructive to think about the following task as an abstraction of the proofs above. Suppose that $C_1 : \{0,1\}^n \to \{0,1\}$ and $C_2 : \{0,1\}^n \to \{0,1\}$ are two copies of the same circuit represented by a 3-CNF by introducing additional variables for each gate (i.e. in the fashion of Cook-Levin Theorem). Let $x_1, x_2 \in \{0,1\}^n$. We need to prove from

$$\Gamma := \{x_{1i} \leftrightarrow x_{2i} \mid i \in [n]\}$$

that $C_1(x_1) \leftrightarrow C_2(x_2)$. (For instance, $f(\vec{x}, s_i(\cdot))$ and $h_i(\vec{x}, \cdot, f(\vec{x}, \cdot))$ are essentially two copies of the same circuit by construction.)

Our proof goes as follows. We prove gate-by-gate that the output wires of C_1 and C_2 are identical. For each gate, we only need to deal with a 3-CNF of $O(1)$ size, which can be proved in Frege with $O(1)$ size. Therefore, the total size of the proof is bounded by the number of gates in C_1 and C_2.

Logical rules. Next, we need to consider the case that the next line to translate is a logical rule. Recall that we have the following logical rules:

- (L0): One may introduce $s = s$ for any term s.
- (L1): If $s = t$ has been introduced, one may introduce $t = s$.
- (L2): If $s = t$, $t = u$ have been introduced, one may introduce $s = u$.
- (L3): If $s_1 = t_1, \ldots, s_n = t_n$ has been introduced and $f(x_1, \ldots, x_n)$ is an order-i function symbol with n variables, one may introduce the equation $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$.
- (L4): If $s = t$ has been introduced, v is an order-i term, and x is an variable, one may introduce $s[x/v] = t[x/v]$, where $s[x/v]$ denote the term obtained from s by substituting all occurrences of x by v.

The first three rules (L0), (L1), and (L2) are simple, so we will only sketch the proof for (L3) and (L4).

For (L3), we know by the induction hypothesis that we have already obtained the proofs for the translation of $s_1 = t_1, \ldots, s_n = t_n$. That is, the output variables of s_i and t_i are identical to each other. By the definition of the translation of composition, we are identifying the input variables of f to the output variables of $s_1, \ldots, s_n$ (resp. $t_1, \ldots, t_n$)

to obtain the translation of $f(s_1, \ldots, s_n)$ (resp. $f(t_1, \ldots, t_n)$). We then reduce the task to the form of Remark 5.3.1, which can be proved in polynomial size without the extension rule.

For (L4), we know by the induction hypothesis that we have already obtained the proofs for the translation of $s = t$. Note that this means that we have the proofs for the translation of $s = t$ on *any* input length. In particular, let ℓ_v be the output length of the translation of the term v, we first obtain the proof of the translation of $s = t$ where the input length of x is ℓ_v, and the input lengths of other variables depend on the input lengths we need for the translation of $s[x/v] = t[x/v]$.

Here is why we need the translation to work on any input length!

Next, we prove by structural induction on s (resp. t), again in the meta-theory, that the translation of $s[x/v]$ (resp. $t[x/v]$) is obtained from the translation of s (resp. t) by identifying (each occurrence of) the variable x and (a copy of) the output variables of v. We can easily prove that all copies of the output variables of v are identical according to Remark 5.3.1 in polynomial size as there are only $O(1)$ copies. Therefore, it suffices to prove the translation of $s[x/v] = t[x/v]$ where, instead of creating copies of v for each of its occurrences, always refers to the same translation of v. This is because the following *substitution axiom scheme*

$$\varphi \wedge (p \leftrightarrow q) \to \varphi[p/q]$$

admits a polynomial size proof in Frege (i.e. without the extension rule) by structural induction on the formula φ, which is left as an exercise.

Subsequently, we can use the substitution (generalization) rule

$$\frac{(\mathsf{EF} \vdash)\varphi}{(\mathsf{EF} \vdash)\varphi[p/\psi]}$$

to derive a proof of the translation of $s[x/v] = t[x/v]$ (with all occurrences of v referring to the same set of variables) from a proof of the translation of $s = t$ where the input length of x is ℓ_v; this is because we can substitute each bit of x in the latter proof to the corresponding bits of the output bits of v. This rule can be proved to be *feasibly admissible* in EF by performing induction on the proof of φ; that is, the size of the EF proof of $\varphi[p/\psi]$ is bounded by a polynomial of the size of the EF proof of φ. This completes the case for the rule (L4).

Remark 5.3.2. The admissibility of the substitution (generalization) rule $\frac{\varphi}{\varphi[p/\psi]}$ is feasibly admissible in EF as well as Frege system (i.e. without the extension rule). This is because the axioms and rules in both systems are schematic, and we can simply replace all occurrences of the variable p to a formula ψ without incurring a significant size overhead.

Induction. Finally, it suffices to consider the next line to translate to be an application of the induction rule. Recall that the induction rule is as follows: Suppose that $g(\vec{x})$, $h_0(\vec{x}, y, z)$, $h_1(\vec{x}, y, z)$ functions, and $f_1(\vec{x}, y)$, $f_2(\vec{x}, y)$ are two functions satisfying that equations

This is the only place that we really need the extension rule.

$$f_1(\vec{x}, \varepsilon) = g(\vec{x}), \quad f_1(\vec{x}, s_i(y)) = h(\vec{x}, y, f_1(\vec{x}, y)),\ i \in \{0, 1\} \tag{5.6}$$

$$f_2(\vec{x}, \varepsilon) = g(\vec{x}), \quad f_2(\vec{x}, s_i(y)) = h(\vec{x}, y, f_2(\vec{x}, y)),\ i \in \{0, 1\} \tag{5.7}$$

have all been proved, then one may prove $f_1(\vec{x}, y) = f_2(\vec{x}, y)$. For simplicity, we consider the case where $\vec{x} = x$ (i.e. there is only one additional variable).

Let n_x be the input length for x and n_y be the input length for y in the translation. We assume for simplicity that y is the actual string rather than the $0^i \circ 1 \circ x$ encoding of a string, but the proof extends to the latter case. Suppose that the translation of f_i is $(m_f^i, \ell_f^i, \Gamma_f^i)$. Let ζ_j for $0 \le j \le n_y$ be defined as

$$\zeta_j := m_f^1(n_x, j) + m_f^2(n_x, j)$$

and $\zeta = \max\{\zeta_j\}_{0\le j\le n_y}$. We introduce fresh variables $z_0, z_1, \dots, z_{n_y} \in \{0,1\}^{\zeta}$ satisfying that $z_j = f_1(x, y_{\le j})$ ($y_{\le j}$ denotes the prefix of y of length j); concretely, this is done by the extension rules

$$z_{jk} \leftrightarrow \text{the } k\text{-th output bit of } f_1(x, y_{\le j})$$

for each $j \le n_y$ and $k \in [\zeta]$.

Next, we will show that for $j = 0, 1, \dots, n_y$ there is an EF proof that "z_j is equal to the output of $f_2(x, y_{\le j})$", which is formalized as

$$\bigwedge_{k\in[\zeta]} z_{n_y k} \leftrightarrow \text{the } k\text{-th output bit of } f_2(x, y_{\le j})$$

This suffices as for $j = n_y$, it implies that $z_{n_y} = f_2(\vec{x}, y) = f_1(\vec{x}, y)$. We will prove this by induction on n_y. Note that this induction works within PV rather than in the metatheory. Therefore, to formalize the induction argument in PV, we need to construct an explicit PV function that outputs the proof of "z_j is equal to the output of $f_2(x, y_{\le y})$ given x, y and j, which is left as an exercise.

Base case of the inner induction. For $j = 0$, we can first prove in EF that "z_0 is identical to the output of $g(x)$" by the induction hypothesis. Subsequently, since we have the EF proof of "$g(x)$ is identical to $f_2(\vec{x}, \varepsilon)$", we can obtain an EF proof of "z_0 is identical to the output of $f_2(x, \varepsilon)$". (More formally, we will use the transitivity of implication that is admissible in Frege.)

Induction case of the inner induction. We then consider the induction case. Suppose that we have already obtained the EF proof of "z_j is equal to the output of $f_2(x, y_{\le j})$", we need to search for the EF proof of "z_{j+1} is equal to the output of $f_2(x, y_{\le j+1})$". Indeed, we first find the EF proof of the translation of

$$f_1(x, y_{\le j+1}) = h_{y_{j+1}}(x, y_{\le j}, f_1(x, y_{\le j})), \tag{5.8}$$

which follows from the induction hypothesis (of the "outer" induction in the metatheory) and the definition of the translation. By the definition, we know that "z_j is equal to the output of $f_2(x, y_{\le j})$" and "z_{j+1} is equal to the output of $f_2(x, y_{\le j+1})$".

The "equality" is formalized as a bit-by-bit comparison using the equivalence connective $\leftrightarrow$. Therefore, we can use the substitution axiom scheme $\varphi \wedge (p \leftrightarrow q) \to \varphi[p/q]$ to obtain a proof of

$$\text{“}z_{j+1} \text{ is equal to the output of } h_{y_{j+1}}(x, y_{\le j}, z_j)\text{”;} \tag{5.9}$$

that is, we start from Equation (5.8) and substitute the output bits of $f_1(x, y_{\le j+1})$ to z_{j+1}, and the output bits of $f_1(x, y_{\le j})$ to z_j.

Subsequently, we find the EF proof of the translation of

$$f_2(x, y_{\le j+1}) = h_{y_{j+1}}(x, y, f_2(x, y_{\le j})), \tag{5.10}$$

similar to Equation (5.8). As "z_j is equal to the output of $f_2(x, y_{\le j})$" admits an EF proof, we can again use the substitution axiom scheme to prove the translation of

$$f_2(x, y_{\le j+1}) = h_{y_{j+1}}(x, y_{\le j}, z_j)$$

in EF. By Equation (5.9) and the substitution axiom scheme, we can then prove in EF that the output of $f_2(x, y_{\le j+1})$ is identical to z_{j+1}, which completes the proof.

Remark 5.3.3. We make a brief remark on why we need the extension rule here. Suppose that we do not have the extension rule, to formalize the same proof, we will need to explicitly write down a sequence of formulas $\vec{\psi}_0, \vec{\psi}_1, \dots, \vec{\psi}_{n_y}$ such that $\vec{\psi}_{j+1}$ *syntactically* encodes $h_{y_{j+1}}(x, y_{\le j}, \vec{\psi}_j)$ — the naïve encoding requires exponential size as we need to consider 2^{n_y} possibilities of the variable y.

5.4 Propositional Proofs to Arithmetic Proofs

Finally, we prove the other side of Theorem 5.1.1, i.e., the existence of short propositional proofs to Cook's translation of an arithmetic formula implies provability in PV. Formally:

Lemma 5.4.1. *Let $s(\vec{x}) = t(\vec{x})$ be a PV equation, where $\vec{x} = (x_1, \dots, x_k)$ be the variables. Suppose that there exists a PV function $\mathsf{Gen}_{s=t}$ such that $\mathsf{PV}\text{-}\mathsf{PL}$ proves*

$$\ell = \mathsf{Gen}_{s=t}(z_1, \dots, z_k) \vdash \mathsf{Verifier}(\ell) = 1 \wedge \mathsf{Conc}(\mathsf{Head}(\ell)) = [s = t]_{\mathsf{Cook}}^{|z_1|, \dots, |z_k|},$$

i.e., $\mathsf{Gen}_{s=t}(z_1, \dots, z_k)$ produces an EF proof of $[s = t]_{\mathsf{Cook}}^{|z_1|, \dots, |z_k|}$, then the equation $s(\vec{x}) = t(\vec{x})$ is provable in PV.

Intuition. The proof of Lemma 5.4.1 involves two steps. First, assume towards a contradiction that $s(\vec{x}) \neq t(\vec{x})$, we need to prove that

> "there is a falsifying assignment of the propositional formula $[s = t]_{\mathsf{Cook}}^{|x_1|, \dots, |x_k|}$."

(Here, we assume a straightforward formalization of the assignment and valuation of propositional formulas.) This is proved by structural induction on s and t *in the meta-theory.* Next, we prove the "reflection principle" of EF in $\mathsf{PV}\text{-}\mathsf{PL}$, namely,

> ($\mathsf{PV}\text{-}\mathsf{PL} \vdash$) "for any formula φ, assignment a, and EF proof π, if π is a valid proof of φ, then a is not a falsifying assignment of φ."

This suffices as we know by the assumption that $[s = t]_{\mathsf{Cook}}^{|x_1|, \dots, |x_k|}$ admits an explicit EF proof, which leads to a contradiction that $\vec{x}$ is a falsifying assignment.

To prove the reflection principle, we perform induction on the proof π *within* $\mathsf{PV}\text{-}\mathsf{PL}$ — from a computational perspective, we will design an explicit feasible algorithm such that given a formula φ, a falsifying assignment a, and an EF proof π, it outputs a line of the EF proof π that is syntactically incorrect.

Step 1: False arithmetic statement to falsifying assignment. In the first step, we need to show that if $s(\vec{x}) \neq t(\vec{x})$, then $\vec{x}$ induces a falsifying assignment of the propositional formula $[s = t]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}$. The formalization of the statement is straightforward: Fix encoding of assignments (e.g. as a dictionary from variable ID to $\{0, 1\}$); consider a function $\mathsf{Val}(\varphi, a)$ that evaluates the formula φ with the assignment a, we will design a function $\mathsf{Assign}_{s\neq t}(\vec{x})$ such that $\mathsf{PV\text{-}PL}$ proves

Hopefully you are convinced that such an encoding method can be implemented :)

$$s(\vec{x}) \neq t(\vec{x}) \vdash \mathsf{Val}([s = t]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}, \mathsf{Assign}_{s=t}(\vec{x})) = 0.$$

Note that $\mathsf{Assign}_{s\neq t}(\vec{x})$ does not simply output $\vec{x}$, as the translation $[s = t]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}$ involves additional variables that encode the intermediate computation and the output of $s(\vec{x})$, $t(\vec{x})$, and the judgment of whether $s(\vec{x}) = t(\vec{x})$.

Nevertheless, the assignment to all intermediate and output variables can be feasibly derived from the assignment $\vec{x}$ of the input variables to $[s = t]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}$.

Recall that the Cook translation of $s = t$ is defined in two steps. We first obtain the Cook translation of the terms s and t, each of which is a tuple $(\Gamma, m(\vec{n}), \ell(\vec{n}))$. Let $(\Gamma_s, m_s(\vec{n}), \ell_s(\vec{n}))$ be the translation of s and $(\Gamma_t, m_t(\vec{n}), \ell_t(\vec{n}))$ be the translation of t. The translation of $s = t$ is then defined by a formula formalizing

$$\begin{aligned} &\bigwedge \Gamma_s \land \bigwedge \Gamma_t \quad \text{(i.e., if variables encode correct computation of } s \text{ and } t\text{)} \\ &\to \text{“the output variables for } s \text{ is identical to the output variables for } t\text{”.} \end{aligned} \tag{5.11}$$

(The last line is formalized by a bit-by-bit comparison of the output variables of s and t.)

We first define $\mathsf{Assign}_s(\vec{x})$ that outputs an assignment to the $m_s(|x_1|, \dots, |x_k|)$ internal and $\ell_s(|x_1|, \dots, |x_k|)$ output variables of $[s]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}$. We require that $\mathsf{PV\text{-}PL}$ proves $\mathsf{Val}(\land\Gamma_s, \mathsf{Assign}_s(\vec{x})) = 1$, i.e., $\mathsf{Assign}_s(\vec{x})$ encodes a correct computation history of s, and that “the $\ell(|x_1|, \dots, |x_k|)$ output variables are assigned to encode $s(\vec{x})$ by $\mathsf{Assign}_s(\vec{x})$”. To see that this is possible, we prove by induction on the term s in the meta-theory, where in each case we need to properly assign values to the internal variables. Similarly, we define $\mathsf{Assign}_t(\vec{x})$ that outputs an assignment to $[t]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}$.

Next, we define $\mathsf{Assign}_{s=t}(\vec{x})$ as the following assignment to $[s = t]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}$: We assign $\mathsf{Assign}_s(\vec{x})$ to the internal and output variables of s and $\mathsf{Assign}_t(\vec{x})$ to the internal and output variables. Now we can see that under the assignment $\mathsf{Assign}_{s=t}(\vec{x})$ and given $s(\vec{x}) \neq t(\vec{x})$, $\mathsf{PV\text{-}PL}$ proves

- $\bigwedge \Gamma_s$ is satisfied by the correctness of $\mathsf{Assign}_s(\vec{x})$.
- $\bigwedge \Gamma_t$ is satisfied by the correctness of $\mathsf{Assign}_t(\vec{x})$.
- The output variables for s and t are *not* identical; this is because the output variables for s is identical to $s(\vec{x})$ and the output variables for t is identical to $t(\vec{x})$, but $s(\vec{x}) \neq t(\vec{x})$.

Namely, Equation (5.11) (i.e. the Cook translation of $s(\vec{x}) = t(\vec{x})$) evaluates to 0 given the assignment $\mathsf{Assign}_{s=t}(\vec{x})$.

Step 2: Reflection principle. In the second step, we prove the reflection principle of EF in PV-PL. This could be formalized as the PV-PL assertion

$$\mathsf{Verifier}(\pi), \mathsf{Conc}(\pi) = \varphi \vdash \mathsf{Val}(\varphi, a) = 1.$$

Nevertheless, under the Feasible Mathematics Thesis, it suffices to informally prove the statement

> "for any formula φ, assignment a, and EF proof π, if π is a valid proof of φ, then a is not a falsifying assignment of φ."

using informal feasible mathematics, i.e., the three informal postulates.

Recall that in Extended Frege, a valid proof π is a list of formulas where each line is either an instance of an axiom scheme (i.e. K, S or D), or an application of the extension rule, or an application of Modus Ponens rule. Let π be a (possibly invalid) EF proof and φ be the conclusion of the proof π. We will design a feasible algorithm that given π and a falsifying assignment a of φ, outputs a line of π that is invalid. Note that the correctness proof of the feasible algorithm implies the reflection principle of EF.

The algorithm could be formalized as an $\mathsf{IMP}(\mathsf{PV})$ program of form $c_1; c_2$, where c_1 denotes the program for the substitution step, and c_2 denotes the program for the iteration step.

The algorithm works as follows.

- (*Substitution*). Let π be the proof concluding φ. We substitute all variables to Boolean values $\{0, 1\}$ such that the proof remains valid if it was valid before the substitution. Concretely:
  - For each variable x in φ that is assigned to be $a_x \in \{0, 1\}$ by the assignment a, we substitute all occurrences of x in the proof π to a_x.
  - For each variable y in π that is neither in φ nor introduced by the extension rule, we substitute all occurrences of y in the proof π to 0.
  - For variables $z_1, z_2, \dots, z_k$ in π introduced by the extension rule, if they are introduced by

$$\begin{aligned} z_1 &\leftrightarrow \psi_1 \\ z_2 &\leftrightarrow \psi_2 \\ &\vdots \\ z_k &\leftrightarrow \psi_k \end{aligned}$$

    where ψ_i does not consist of $z_i, \dots, z_k$ for each $i \in [k]$, we can assign $z_1, \dots, z_k$ to $\{0, 1\}$ one by one such that all formulas are valid. Otherwise, it means that one of the applications of the extension rule is invalid as it does not introduce a "fresh" variable — the algorithm outputs the line and halts.

This invariant is feasibly checkable; therefore it could be formalized, for instance, using the (Hoare) loop rule of $\mathsf{IMP}(\mathsf{PV})$ programs.

- (*Iteration*). Now we obtain a proof π that does not contain any variable and the conclusion of π (i.e. the last line) is a formula that evaluates to 0 (as a is a falsifying assignment of φ). Suppose that there are ℓ lines in π. We initialize $i \leftarrow \ell$ and consider the following iterative procedure:
  - We maintain the iteration invariant that at the start of each iteration, the i-th line in π is a formula that evaluates to 0, which holds in the first iteration.

  The index i is strictly decreased at the end of the iteration if the algorithm does not halt (by finding an invalid line of proof in π). Therefore, the algorithm halts in at most ℓ iterations.

- In each iteration, we consider the axiom or rule applied in the i-th line of π — it is either an application of one of the axiom schemes $\mathsf{K}, \mathsf{S}, \mathsf{D}$, or an extension rule, or a Modus Ponens rule.
  - Suppose that the i-th line is an application of an axiom scheme, it must be invalid, and the algorithm can halt and output the index i. This is because axiom schemes are PV provably valid (i.e., it cannot evaluate to 0), but by the invariant that the i-th line evaluates to 0.
  - It could not be an application of the extension rule. This is because by the (*substitution*) step of the algorithm, we have assigned $\{0, 1\}$ to variables introduced by the extension rule such that all applications of the extension rule are valid formulas, i.e., evaluates to 1.
  - Suppose that the i-line is an application of the Modus Ponens rule concluding ψ, the i-th line also maintains a formula φ such that φ and $\varphi \to \psi$ should have appeared as the conclusions of the j_1-th and the j_2-th line of π, respectively, for some $j_1, j_2 < i$. If this is not the case, the i-th line is invalid and our algorithm can simply halt and output the index i. Otherwise, we PV provably know that either φ evaluates to 0, or $\varphi \to \psi$ evaluates to 0. We can update $i \leftarrow j_1$ in the former case and $i \leftarrow j_2$ in the latter case.

The correctness and feasibility of the iteration algorithm can be feasibly provable (using induction on feasible property) by the iteration invariant.

Wrapping up the proof. Recall that we want to show that if PV proves that $[s = t]_{\mathsf{Cook}}^{|z_1|,\dots,|z_k|}$ admits an EF proof (generated by an explicit PV formula $\mathsf{Gen}_{s=t}$), then $s(\vec{x}) = t(\vec{x})$ is provable in PV. We argue in informal feasible mathematics (which can be formalized in PV-PL). Suppose, towards a contradiction that $s(\vec{x}) \neq t(\vec{x})$ for some $\vec{x}$, we prove in the first step that we can obtain a falsifying assignment a of the formula

$$\varphi := [s = t]_{\mathsf{Cook}}^{|x_1|,\dots,|x_k|}$$

from $\vec{x}$. Moreover, we know by the assumption that $\mathsf{Gen}_{s=t}$ generates an EF proof π of φ. By the reflection principle proved in the second step, however, either the proof π is invalid or a is not a falsifying assignment of φ, which leads to a contradiction.

Sanity Check: Why ain't we assuming $s(\vec{x}) = t(\vec{x})$ is unprovable in PV?

5.5 Bibliographical and Other Remarks

The idea of translating arithmetic formulas and proofs to families of propositional formulas and proofs is due to Cook [Coo75] (see also [KP90] for an extension). Cook [Coo75] used Extended Resolution instead of Extended Frege, while these two systems polynomially simulate each other [CR79]. Indeed, one can verify that the simulation is formalizable in PV. The propositional translation was later generalized to other bounded theories, including Parikh's $I\Delta_0$ [PW85, Ajt83] and Buss's theories [KP90, KT90].

The idea of simulating propositional proof systems with the reflection principle is due to Cook [Coo75] and was extended to other systems [KP90, KT90, Kra95b].

Readers interested in the propositional translation of other theories are referred to standard textbooks, see [Kra95a, Chapter 9], [Kra19, Chapter 12], [CN10, Chapter 10].

The propositional translation is crucial to the *model-theoretic* approach in bounded arithmetic, which we will not cover. We refer interested readers to the textbook [Kra11] and [Kra19, Chapter 20] and the references therein.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach.* Cambridge University Press, 2009.

[Aho11] Alfred V. Aho. Ubiquity symposium: Computation and computational thinking. *Ubiquity*, 2011(January):1, 2011.

[Ajt83] Miklós Ajtai. Σ_1^1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24(1):1–48, 1983.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.

[AT24] Albert Atserias and Iddo Tzameret. Feasibly constructive proof of schwartz-zippel lemma and the complexity of finding hitting sets. *Electron. Colloquium Comput. Complex.*, TR24-174, 2024.

[BBF$^+$19] Arnold Beckmann, Sam Buss, Sy-David Friedman, Moritz Müller, and Neil Thapen. Feasible set functions have small circuits. *Comput.*, 8(1):67–98, 2019.

[Bis67] Errett Bishop. Foundations of constructive analysis. 1967.

[BPI22] Douglas Bridges, Erik Palmgren, and Hajime Ishihara. Constructive Mathematics. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Fall 2022 edition, 2022.

[Bus85] Samuel R Buss. *Bounded arithmetic.* Princeton University, 1985.

[Bus86] Samuel R. Buss. *Bounded Arithmetic.* Bibliopolis, 1986.

[Bus94] Samuel R Buss. On gödel's theorems on lengths of proofs i: Number of lines and speedup for arithmetics. *The Journal of Symbolic Logic*, 59(3):737–756, 1994.

[Bus95a] Samuel R Buss. On gödel's theorems on lengths of proofs ii: Lower bounds for recognizing k symbol provability. In *Feasible mathematics II*, pages 57–90. Springer, 1995.

[Bus95b] Samuel R. Buss. Relating the bounded arithmetic and polynomial time hierarchies. *Annals of Pure and Applied Logic*, 75(1-2):67–77, 1995.

[Bus99] Samuel R. Buss. Bounded arithmetic, proof complexity and two papers of parikh. *Ann. Pure Appl. Log.*, 96(1-3):43–55, 1999.

[Bus08] Samuel R. Buss. Bounded arithmetic, cryptography and complexity. *Theoria*, 63:147–167, 2008.

[CKKO21] Marco Carmosino, Valentine Kabanets, Antonina Kolokolova, and Igor C. Oliveira. Learn-uniform circuit lower bounds and provability in bounded arithmetic. In *Symposium on Foundations of Computer Science* (FOCS), 2021.

[CLO24] Lijie Chen, Jiatu Li, and Igor C. Oliveira. Reverse mathematics of complexity lower bounds. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 505–527. IEEE, 2024.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[CN10] Stephen Cook and Phuong Nguyen. *Logical Foundations of Proof Complexity.* Cambridge University Press, 2010.

[Cob65] Alan Cobham. The intrinsic computational difficulty of functions. 1965.

[Coo75] Stephen A. Cook. Feasibly constructive proofs and the propositional calculus (preliminary version). In William C. Rounds, Nancy Martin, Jack W. Carlyle, and Michael A. Harrison, editors, *Proceedings of the 7th Annual ACM Symposium on Theory of Computing, May 5-7, 1975, Albuquerque, New Mexico, USA*, pages 83–97. ACM, 1975.

[Coo90] Stephen A Cook. *Computational complexity of higher type functions.* American Mathematical Society, 1990.

[CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.

[Cra36] Harald Cramér. On the order of magnitude of the difference between consecutive prime numbers. *Acta arithmetica*, 2:23–46, 1936.

[CU93] Stephen Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103–200, 1993.

[Dij68] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[Dij72] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Din07] Irit Dinur. The PCP theorem by gap amplification. *J. ACM*, 54(3):12, 2007.

[DM22] Damir D. Dzhafarov and Carl Mummert. *Reverse mathematics: problems, reductions, and proofs.* Springer Nature, 2022.

[DMKA$^+$15] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

[Gen36] Gerhard Gentzen. Die widerspruchsfreiheit der reinen zahlentheorie. *Mathematische annalen*, 112(1):493–565, 1936.

[Göd36] Kurt Gödel. Über die länge von beweisen. *Ergebnisse eines mathematischen Kolloquiums*, 7:23–24, 1936.

[ILW23] Rahul Ilango, Jiatu Li, and R. Ryan Williams. Indistinguishability obfuscation, range avoidance, and bounded arithmetic. In *Symposium on Theory of Computing* (STOC), pages 1076–1089. ACM, 2023.

[Jeř06] Emil Jeřábek. The strength of sharply bounded induction. *Mathematical Logic Quarterly*, 52(6):613–624, 2006.

[Jer07] Emil Jerábek. Approximate counting in bounded arithmetic. *J. Symb. Log.*, 72(3):959–993, 2007.

[JJ22] Abhishek Jain and Zhengzhong Jin. Indistinguishability obfuscation via mathematical proofs of equivalence. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 1023–1034. IEEE, 2022.

[JKLM24] Zhengzhong Jin, Yael Tauman Kalai, Alex Lombardi, and Surya Mathialagan. Universal snargs for np from proofs of correctness. *Cryptology ePrint Archive*, 2024.

[JKLV24] Zhengzhong Jin, Yael Kalai, Alex Lombardi, and Vinod Vaikuntanathan. Snargs under LWE via propositional proofs. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1750–1757. ACM, 2024.

[KP90] Jan Krajícek and Pavel Pudlák. Quantified propositional calculi and fragments of bounded arithmetic. *Math. Log. Q.*, 36(1):29–46, 1990.

[KP98] Jan Krajíček and Pavel Pudlák. Some consequences of cryptographical conjectures for S_2^1 and EF. *Inf. Comput.*, 140(1):82–94, 1998.

[KPT91] Jan Krajíček, Pavel Pudlák, and Gaisi Takeuti. Bounded arithmetic and the polynomial hierarchy. *Annals of Pure and Applied Logic*, 52(1-2):143–153, 1991.

[Kra95a] Jan Krajíček. *Bounded Arithmetic, Propositional Logic, and Complexity Theory.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1995.

[Kra95b] Jan Krajíček. On frege and extended frege proof systems. In *Feasible Mathematics II*, pages 284–319. Springer, 1995.

[Kra01] Jan Krajíček. On the weak pigeonhole principle. *Fundamenta Mathematicae*, 1(170):123–140, 2001.

[Kra11] Jan Krajícek. *Forcing with Random Variables and Proof Complexity*, volume 382 of *London Mathematical Society lecture note series.* Cambridge University Press, 2011.

[Kra19] Jan Krajíček. *Proof complexity*, volume 170. Cambridge University Press, 2019.

[KT90] Jan Krajíček and Gaisi Takeuti. On bounded Σ_1^1 polynomial induction. In *Feasible Mathematics: A Mathematical Sciences Institute Workshop, Ithaca, New York, June 1989*, pages 259–280. Springer, 1990.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[LLR24] Jiawei Li, Yuhao Li, and Hanlin Ren. Metamathematics of resolution lower bounds: A TFNP perspective. *Electron. Colloquium Comput. Complex.*, TR24-190, 2024.

[Ngu07] Phuong Nguyen. Separating dag-like and tree-like proof systems. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 235–244. IEEE Computer Society, 2007.

[Oja04] Kerry Ojakian. *Combinatorics in bounded arithmetic.* PhD thesis, Carnegie Mellon University, 2004.

[Oli24] Igor Carboni Oliveira. Meta-mathematics of computational complexity theory. 2024.

[Par71] Rohit Parikh. Existence and feasibility in arithmetic. *Journal of Symbolic Logic*, 36(3):494–508, 1971.

[PCG+10] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, page 16, 2010.

[Pic15] Ján Pich. Logical strength of complexity theory and a formalization of the PCP theorem in bounded arithmetic. *Log. Methods Comput. Sci.*, 11(2), 2015.

[PS21] Ján Pich and Rahul Santhanam. Strong co-nondeterministic lower bounds for NP cannot be proved feasibly. In *Symposium on Theory of Computing* (STOC), pages 223–233, 2021.

[Pud87] Pavel Pudlák. Improved bounds to the lengths of proofs of finitistic consistency statements. *Contemporary Mathematics*, 65, 1987.

[PW85] Jeff Paris and Alex Wilkie. Counting problems in bounded arithmetic. In *Methods in Mathematical Logic: Proceedings of the 6th Latin American Symposium on Mathematical Logic held in Caracas, Venezuela August 1–6, 1983*, pages 317–340. Springer, 1985.

[Sim09] Stephen George Simpson. *Subsystems of second order arithmetic*, volume 1. Cambridge University Press, 2009.

[Sko23] Thoralf Skolem. *Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Ausdehnungsbereich.* Dybusach, 1923.

[Sti20] John Stillwell. *Reverse mathematics.* Springer, 2020.

[TS00] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2 edition, 2000.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il