University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Paweł Balawender
Student no. 429141

Practical programming languages capturing complexity classes

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr hab. Paweł Parys, prof. UW
Institute of Informatics, University of Warsaw

Warsaw, July 2025

Abstract

The aim of this thesis is to certify complexity bounds for computer programs by construction: to design programming languages whose syntax guarantees membership in a target complexity class. In particular, verifying a program's complexity reduces to checking that it is syntactically correct. We examine several lines of work that appear to support this goal, including formalization of the semantics of standard programming languages, descriptive complexity, and programming languages arising from implicit computational complexity. Our conclusions are mostly negative: in their current form, these approaches do not scale to certifying the complexity of realistic algorithms.

On the positive side, we argue that bounded arithmetic, whose theories are known to capture a range of complexity classes, can serve, when combined with modern proof assistants, as a viable basis for programming with certified complexity. Concretely, we implement in Lean 4 a formalization of bounded-arithmetic theories and discuss how some proofs in this formalization can be transformed into computational procedures. We develop one of the first studies of arithmetic theories for small complexity classes in a modern proof assistant, showing how these ideas can underpin a methodology for certifying both correctness and resource bounds of programs.

Keywords

Programming languages, Implicit computational complexity, Descriptive complexity, Bounded arithmetic, Lean 4, Rocq, Coq

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

F. Theory of computation
F.3. Logics and meanings of programs
F.3.3. Studies of program constructs

Tytuł pracy w języku polskim

Praktyczne języki programowania wyrażające klasy złożoności

Contents

List of Listings

Chapter 1

Introduction

The aim of this thesis is to certify complexity bounds by construction: to design programming and specification formalisms whose syntax guarantees membership in a target complexity class, so that verifying a program's complexity reduces to checking that it is syntactically correct. The goal is not to make programming itself easier, but to make it easier for someone who receives a program to trust that it is correct and runs within the claimed resource bounds. Our conclusions are mostly *negative*: we explore several very promising fields of research and conclude that, in their current form, they do not allow us to certify the complexity of programs at scale. In Section 8.4 we present the main technical contribution of this work: we explain how bounded arithmetic, combined with modern proof assistants, offers a viable way to approach this goal. Theories from bounded arithmetic generalize well to account for algorithms from higher and lower complexity classes, in contrast to many systems from Implicit Computational Complexity studied in Chapter 7, which are typically tailored to a single class.

Structure of this work In Chapter 3 we explore how different programming paradigms can help us reach this goal. We also consider the notion of complexity in other models of computation and whether it would make sense to capture them instead of the complexity of Turing machine-like (imperative) models.

In Chapter 4 we examine the most obvious way of certifying complexity: directly proving on a computer that a given C++ program operates in, say, time $\mathcal{O}(n^2)$. Using an existing development in the Rocq proof assistant as a case study, we argue that this is not a viable solution for certifying the complexity of realistic programs.

In Chapter 5 we study logics that characterize computational complexity classes by the complexity of formulas needed to describe their problems. These characterizations are very well suited to decision problems, but less so to computing general functions with output, as we usually do in programming.

In Chapter 6 we explore this idea further, examining how classes of decision and function problems relate and whether this distinction matters for certifying complexity. The answer is positive: it is not always trivial to reduce a general function problem to a decision problem. In the same chapter, we examine the feasibility of capturing complexity classes by first fixing a complete problem and then capturing the (weaker) programming language for reductions to that problem. We can obtain the function classes FL and FP, consisting of functions computable in logarithmic space and polynomial time, respectively, and several circuit complexity classes this way, but at the price of relying on characterizations of low circuit complexity classes that are not convenient to use either. We defer a detailed treatment of circuit reductions and

uniformity to an appendix and proceed to explore direct characterizations of FL and FP in the next chapter.

In Chapter 7 we study actual programming languages (more or less purely academic) that capture complexity classes. We conclude this chapter with a critique of these approaches: they are fragmented (every complexity class would need a separate programming language), and even for a fixed class, the theories are often able to *solve every problem* in the class but not *express every algorithm* we naturally use to solve problems in that class. Despite their relevance, we were not able to certify the complexity of existing algorithms in these systems, and therefore we cannot use them as foundation for the kind of programming language we are interested in.

In Chapter 8 we study how to extract computational content from proofs conducted in bounded arithmetic, a field that studies mathematical theories that also capture computational complexity classes.[1] This chapter collects our main technical results and explains why bounded arithmetic, formalized in a proof assistant, appears to be a promising foundation for complexity-certified programming. Part of this work was presented at AITP 2025 conference; an abstract of our early developments is available at: `https://aitp-conference.org/2025/abstract/AITP_2025_paper_7.pdf`.[2] A significant part of the developments described in this chapter was conducted during the main author's visit to INRIA Paris, made possible by a scholarship from ZSM IDUB program.

[1] Our work is available at `https://github.com/ruplet/formalization-of-bounded-arithmetic`.

[2] It was the basis for the acceptance of our short talk; for the reviews, please see `https://github.com/ruplet/formalization-of-bounded-arithmetic/blob/main/presentation-AITP2025/aitp-reviews.md`.

Chapter 2

Preliminaries

In this chapter we introduce the standard definitions for logic and complexity theory, necessary to fix the language we will use in the rest of the work. We restrict attention to the logical frameworks needed later (single-sorted first-order logic and its two-sorted counterpart) and omit the analogous presentations for propositional calculus or full second-order logic. The semantics is entirely classical (as in: not intuitionistic).

In Section 2.5 we first introduce the so-called *non-uniform* circuit families (e.g. $\mathsf{FAC}^i_{/poly}$), then define a weak notion of uniformity and the appropriate *uniform* circuit families (e.g. FAC^i). The notion of uniformity used in this thesis (cf. Definition 2.5.3) and the results using it are usually not interesting. However, in this work we will not focus too much on that problem. We leave out the complicated divagations about the appropriate (i.e. much weaker) notions of uniformity to Appendix A.

Remark 2.1 (Bibliography). The definitions in Section 2.1 are in the style of [CN08, Section 2B; CN10, Section II.2]. The definitions in Section 2.2 are in the style of [CN08, Section 4B; CN10, Section IV.2]. We will mostly need them for Chapter 8, but also for Chapter 5 — the results discussed in the latter are mostly from [Imm99], where different style of definitions is used. However, [Imm99] doesn't introduce two-sorted logic. The single-sorted definitions differences are mostly negligible and the consistent treatment of single- and two-sorted logic, is important for Chapter 8.

Much effort has been put into considering different versions of the definitions in Section 2.4 to keep them consistent with the definitions of functional complexity classes studied later in Section 6.2, e.g. the class FNP defined in Definition 6.2.3. These are much less standard, and often have no clear consensus in the literature.

The definitions of decisional circuit classes are from [Vol99].

2.1. Single-sorted first-order logic

Definition 2.1.1 (First-order vocabulary and syntax)**.** A *first-order vocabulary* (or *language*) $\mathcal{L}$ consists of:

(i) for each $n \geq 0$, a (possibly empty) set of n-ary *function symbols*;

(ii) for each $n \geq 0$, a set of n-ary *predicate symbols*, which is nonempty for at least one n.

We use $f, g, h, \ldots$ as meta-variables for function symbols and $P, Q, R, \ldots$ for predicate symbols. A 0-ary function symbol is called a *constant symbol*. In addition, the following logical symbols are available to build first-order terms and formulas:

(i) an infinite set of *variables*. We use $x, y, z, \ldots$ and sometimes $a, b, c, \ldots$ as meta-variables for variables;
(ii) the connectives $\neg$, $\wedge$, $\vee$ (not, and, or) and logical constants $\bot$, $\top$ (false, true);
(iii) the quantifiers $\forall$, $\exists$ (for all, there exists);
(iv) parentheses (,).

Remark 2.2. Note that we don't introduce the equality sign $=$ as a special symbol in the vocabulary. We will treat equality as a standard binary relation and only require it to be treated specially when reasoning about semantics.

Definition 2.1.2 ($\mathcal{L}$-terms)**.** Let $\mathcal{L}$ be a first-order vocabulary. The set of *$\mathcal{L}$-terms* is defined inductively as follows:
(i) every variable is an $\mathcal{L}$-term;
(ii) if f is an n-ary function symbol of $\mathcal{L}$ and $t_1, \ldots, t_n$ are $\mathcal{L}$-terms, then

$$f(t_1, \ldots, t_n)$$

is an $\mathcal{L}$-term.

Definition 2.1.3 ($\mathcal{L}$-formulas)**.** Let $\mathcal{L}$ be a first-order vocabulary. The set of *first-order formulas in $\mathcal{L}$* (or *$\mathcal{L}$-formulas*) is defined inductively as follows:
(i) the logical constants $\bot$ and $\top$ are atomic formulas;
(ii) if P is an n-ary predicate symbol in $\mathcal{L}$ and $t_1, \ldots, t_n$ are $\mathcal{L}$-terms, then

$$P(t_1, \ldots, t_n)$$

is an *atomic* $\mathcal{L}$-formula;
(iii) if φ and ψ are $\mathcal{L}$-formulas, then $\neg\varphi$, $(\varphi \wedge \psi)$, and $(\varphi \vee \psi)$ are $\mathcal{L}$-formulas;
(iv) if φ is an $\mathcal{L}$-formula and x is a variable, then $\forall x.\, \varphi$ and $\exists x.\, \varphi$ are $\mathcal{L}$-formulas.
For example,

$$(\neg\forall x.\, P(x) \vee \exists x.\, \neg P(x)) \quad \text{and} \quad (\forall x.\, \neg Q(x, f(y)) \wedge \neg\forall z.\, Q(f(y), z))$$

are $\mathcal{L}$-formulas (for suitable choices of P, Q, and f in $\mathcal{L}$).

Definition 2.1.4 (The language of arithmetic)**.** The *language of arithmetic* is

$$L_A = [\, 0,\, 1,\, +,\, \cdot\, ;\, =,\, \leq\,],$$

where 0 and 1 are constant symbols, $+$ and $\cdot$ are binary function symbols, and $=$ and $\leq$ are binary predicate symbols. We will write these symbols in infix form.

Definition 2.1.5 (Free and bound variables)**.** Let φ be a formula and x a variable. An occurrence of x in φ is *bound* if it lies within a subformula of φ of the form $\forall x.\, \psi$ or $\exists x.\, \psi$. Any other occurrence of x in φ is called *free*.

Definition 2.1.6 (Closed terms, closed formulas, sentences)**.** A formula is *closed* if it contains no free occurrence of any variable. A term is *closed* if it contains no variables at all. A closed formula is also called a *sentence*.

Definition 2.1.7 ($\mathcal{L}$-structure)**.** Let $\mathcal{L}$ be a first-order vocabulary. An *$\mathcal{L}$-structure $\mathcal{M}$* consists of:

(i) a nonempty set M, called the *universe* (variables are intended to range over M);
(ii) for each n-ary function symbol f in $\mathcal{L}$, an associated function $f^{\mathcal{M}} : M^n \to M$;
(iii) for each n-ary predicate symbol P in $\mathcal{L}$, an associated relation $P^{\mathcal{M}} \subseteq M^n$.

Remark 2.3. Note that to "syntactical" relations, we assign "real" relations defined on the underlying elements of the structure. We will want to treat some of these relations specially, e.g. to make sure that the "=" relation is interpreted as the actual equality, or that a designated "SUCC(x, y)" relation holds only if the underlying objects are actual natural numbers, for which we have $x + 1 = y$; see e.g. Definition 5.0.4.

Definition 2.1.8 (Object Assignment)**.** Let $\mathcal{M}$ be a structure with universe M. An *object assignment* σ for $\mathcal{M}$ is a mapping from variables to the universe M.

Notation 1. Let x be a variable and $m \in M$. We write $\sigma(m/x)$ for the assignment that is the same as σ except that it maps x to m.

Definition 2.1.9 (Basic Semantic Definition)**.** Let $\mathcal{L}$ be a first-order vocabulary, let $\mathcal{M}$ be an $\mathcal{L}$-structure with universe M, and let σ be an object assignment for $\mathcal{M}$.

Interpretation of terms. Each $\mathcal{L}$-term t is assigned an element $t^{\mathcal{M}}[\sigma] \in M$, defined by structural induction on t:
(i) for each variable x, $x^{\mathcal{M}}[\sigma] = \sigma(x)$;
(ii) $(ft_1 \dots t_n)^{\mathcal{M}}[\sigma] = f^{\mathcal{M}}(t_1^{\mathcal{M}}[\sigma], \dots, t_n^{\mathcal{M}}[\sigma])$.

Satisfaction of formulas. For an $\mathcal{L}$-formula φ, the relation

$$\mathcal{M} \models \varphi[\sigma]$$

(read: "$\mathcal{M}$ satisfies φ under σ") is defined by structural induction on φ:
(i) the structure $\mathcal{M}$ satisfies $\top[\sigma]$ and $\mathcal{M} \not\models \bot[\sigma]$;
(ii) for an atomic formula $Pt_1 \dots t_n$ (with P an n-ary predicate symbol),

$$\mathcal{M} \models (Pt_1 \dots t_n)[\sigma] \text{ iff } \langle t_1^{\mathcal{M}}[\sigma], \dots, t_n^{\mathcal{M}}[\sigma] \rangle \in P^{\mathcal{M}};$$

(iii) if $\mathcal{L}$ contains $=$, then for terms s, t,

$$\mathcal{M} \models (s = t)[\sigma] \text{ iff } s^{\mathcal{M}}[\sigma] = t^{\mathcal{M}}[\sigma];$$

(iv) the structure satisfies $\neg\varphi[\sigma]$ iff $\mathcal{M} \not\models \varphi[\sigma]$;
(v) the structure satisfies $(\varphi \vee \psi)[\sigma]$ iff $\mathcal{M} \models \varphi[\sigma]$ or $\mathcal{M} \models \psi[\sigma]$;
(vi) the structure satisfies $(\varphi \wedge \psi)[\sigma]$ iff $\mathcal{M} \models \varphi[\sigma]$ and $\mathcal{M} \models \psi[\sigma]$;
(vii) the structure satisfies $(\forall x.\, \varphi)[\sigma]$ iff $\mathcal{M} \models \varphi[\sigma(m/x)]$ for all $m \in M$;
(viii) the structure satisfies $(\exists x.\, \varphi)[\sigma]$ iff $\mathcal{M} \models \varphi[\sigma(m/x)]$ for some $m \in M$.

If t is a closed term, then $t^{\mathcal{M}}[\sigma]$ is independent of σ, and we simply write $t^{\mathcal{M}}$. Similarly, if φ is a sentence, we often write $\mathcal{M} \models \varphi$ instead of $\mathcal{M} \models \varphi[\sigma]$, since the choice of σ does not matter.

2.2. Two-sorted first-order logic

Two-sorted first-order logic extends the single-sorted setting in a routine way. We will only record the differences and skip the analogous definitions. A systematic presentation can be found in [CN08, Section 4B; CN10, Section IV.2]. In principle one could work with arbitrary pairs of sorts, but in this thesis we instantiate the framework to the familiar number sort (ranging over $\mathbb{N}$) and string sort (ranging over finite binary strings). The goal of this section is therefore to emphasise what changes when we move from the definitions of Section 2.1 to this concrete two-sorted setting.

Definition 2.2.1 (Two-sorted first-order vocabularies)**.** A *two-sorted first-order vocabulary* (often abbreviated simply as a two-sorted language) $\mathcal{L}$ consists of collections of function and predicate symbols, much like an ordinary single-sorted vocabulary, but now the symbols may accept arguments of either of the two sorts. Moreover, the function symbols come in two varieties:

(i) *number-valued* function symbols, whose outputs lie in the number sort; and
(ii) *string-valued* function symbols, whose outputs lie in the string sort.

For any pair $n, m \in \mathbb{N}$, the vocabulary contains:

(i) a set of (n, m)-ary number-function symbols;
(ii) a set of (n, m)-ary string-function symbols; and
(iii) a set of (n, m)-ary predicate symbols.

A $(0, 0)$-ary function symbol is simply a constant symbol, which may be either a constant of the number sort or a constant of the string sort.

We use $f, g, h, \ldots$ as metavariables for number-valued function symbols, $F, G, H, \ldots$ for string-function symbols, and $P, Q, R, \ldots$ for predicate symbols.

Definition 2.2.2 (The language $\mathcal{L}_A^2$)**.** As an example, consider the following two-sorted extension of the arithmetical language $\mathcal{L}_A$ (Definition 2.1.4):

$$\mathcal{L}_A^2 \;=\; [\,0,\ 1,\ +,\ \cdot,\ |\cdot|\ ;\ =_1,\ =_2,\ \leq,\ \in\,].$$

Here the symbols $0, 1, +, \cdot, =_1, \leq$ are symbols of $\mathcal{L}_A$ (with $=_1$ corresponding to the usual equality of numbers). The symbol $\vec{X}$ is a number-valued function symbol giving the length of a string X. The binary predicate $\in$ relates a number and a string and is used to express membership: intuitively, $i \in X$ means that the i-th bit of the string X is 1. The symbol $=_2$ denotes equality between objects of the second sort.

For convenience, when t is a number term, we abbreviate

$$X(t) \;\coloneqq\; t \in X.$$

Thus $X(i)$ plays the role of the i-th bit of the binary string X.

Remark 2.4. Note that $X(i)$ and $|X|$ intuitively should be related in some way. We don't set any implicit assumptions on the models we will consider. The necessary conditions will be fixed by the axiomatic system from Definition 8.2.1.

In $\mathcal{L}_A^2$, the symbols $+$ and $\cdot$ each have arity $(2, 0)$; the length function $|\cdot|$ has arity $(0, 1)$; and the predicate $\in$ has arity $(1, 1)$.

Notation 2 (Bounded formulas)**.** Let $\mathcal{L}$ be a two-sorted vocabulary. If x is a number variable and X a string variable that do not occur in the $\mathcal{L}$-number term t, we use the following abbreviations:

$$\begin{aligned}
\exists x \leq t.\, \varphi \text{ stands for } \exists x.\, (x \leq t \ \wedge \ \varphi),\\
\forall x \leq t.\, \varphi \text{ stands for } \forall x.\, (x \leq t \ \rightarrow \ \varphi),\\
\exists X \leq t.\, \varphi \text{ stands for } \exists X.\, (\,|X| \leq t \ \wedge \ \varphi),\\
\forall X \leq t.\, \varphi \text{ stands for } \forall X.\, (\,|X| \leq t \ \rightarrow \ \varphi).
\end{aligned}$$

A quantifier appearing in one of these forms is called *bounded*, and a *bounded formula* is a formula in which every quantifier is bounded.

Notation. The expression $\exists \vec{x} \leq \vec{t}.\, \varphi$ abbreviates a block of bounded number quantifiers $\exists x_1 \leq t_1. \cdots \exists x_k \leq t_k.\, \varphi$ for some k, where no variable x_i occurs in any term t_j (even when $i < j$). The same convention applies to $\forall \vec{x} \leq \vec{t}$, $\exists \vec{X} \leq \vec{t}$, and $\forall \vec{X} \leq \vec{t}$.

Definition 2.2.3 (The $\Sigma^1_1(\mathcal{L})$, $\Sigma^B_i(\mathcal{L})$, and $\Pi^B_i(\mathcal{L})$ formulas)**.** Let $\mathcal{L} \supseteq \mathcal{L}^2_A$ be a two-sorted vocabulary.

(i) the class $\Sigma^B_0(\mathcal{L}) = \Pi^B_0(\mathcal{L})$ consists of all $\mathcal{L}$-formulas whose only quantifiers are *bounded number quantifiers* (string variables may occur free);

(ii) for $i \geq 0$, the class $\Sigma^B_{i+1}(\mathcal{L})$ (resp. $\Pi^B_{i+1}(\mathcal{L})$) consists of formulas of the form

$$\exists \vec{X} \leq \vec{t}.\, \varphi(\vec{X}) \quad (\text{resp. } \forall \vec{X} \leq \vec{t}.\, \varphi(\vec{X})),$$

where:

(i) $\vec{X}$ is a vector of string variables;

(ii) $\vec{t}$ is a vector of $\mathcal{L}^2_A$-terms not involving variables from $\vec{X}$;

(iii) φ is a $\Pi^B_i(\mathcal{L})$ formula (resp. a $\Sigma^B_i(\mathcal{L})$ formula).

(iii) a $\Sigma^1_1(\mathcal{L})$ formula is a formula of the form $\exists \vec{X}.\, \varphi$, where $\vec{X}$ is a vector of zero or more string variables and φ is a $\Sigma^B_0(\mathcal{L})$ formula.

We usually write Σ^B_i for $\Sigma^B_i(\mathcal{L}^2_A)$ and Π^B_i for $\Pi^B_i(\mathcal{L}^2_A)$.

Remark 2.5. The above definition might seem excessively limiting, as it doesn't allow us to have a string quantifier under a number quantifier. An example of a formula which is in none of Σ^B_i is $\forall x. \forall X. \exists Z.\, \varphi(x, Y, Z)$. We don't claim that every formula can be transformed into an equivalent formula in some Σ^B_i. In what follows, we will freely reason about arbitrary two-sorted formulas — we will just not say that they are in Σ^B_i. Details of one such reasoning are e.g. in [CN10, Lemma V.4.10; CN08, Lemma 5.35].

Remark 2.6. The above definition also doesn't allow using X in the term t used to "guard" a bounded quantifier $\exists X \leq t$. Note that if you used $t := |X|$, such a bounded quantifier would expand to $\exists X \leq |X|$, which is obviously equivalent to an unbounded quantifier $\exists X$.

Remark 2.7. The formalism described above is similar to the usual weak monadic second-order logic (WMSO) on words. We phrase it as a two-sorted first-order system to match the presentation in [CN08; CN10] and don't discuss the differences between the two formalisms here.

2.3. Functions computable by Turing machines

We introduce the notion of computing a general $\{0,1\}^* \to \{0,1\}^*$ function early, as this is the primary interest of this thesis. Most of the literature in computational complexity focuses solely on computing Boolean functions which we introduce in Section 2.4. To properly discuss this imbalance, we postpone introducing the *functional* complexity classes until Chapter 6.

Definition 2.3.1 ([AB09, Definition 1.3; AB07, Definition 1.4])**.** Let $f : \{0,1\}^* \to \{0,1\}^*$ and $T : \mathbb{N} \to \mathbb{N}$ be functions, and let M be a Turing machine. We say that M *computes* f if, for every input $x \in \{0,1\}^*$, when M is started in its initial configuration on input x, it eventually halts with the string $f(x)$ written on its output tape.

2.4. Decisional Turing machine complexity classes

In this section we introduce standard complexity classes such as L, P and NP. It is important to note that these classes only contain *decision* problems, i.e. only require computing a function $f : \{0,1\}^* \to \{0,1\}$. Complexity classes for general functions will appear in.

Notation 3. We say that a machine *decides* a language $L \subseteq \{0,1\}^*$ iff it computes the function $f_L : \{0,1\}^* \to \{0,1\}$, where $f_L(x) = 1 \iff x \in L$.

Definition 2.4.1 (Time complexity [AB07, Definition 1.19; AB09, Definition 1.12])**.**
Let $T : \mathbb{N} \to \mathbb{N}$ be a function. We define $\mathsf{DTIME}(T(n))$ to be the class of all Boolean functions that are computable by some deterministic Turing machine running in at most $c_1 \cdot T(n) + c_2$ steps on every input of length n, for some constants $c_1, c_2 > 0$.

Definition 2.4.2 (Space complexity [AB07, Definition 4.1; AB09, Definition 4.1])**.**
Let $S : \mathbb{N} \to \mathbb{N}$ be a function and let $L \subseteq \{0,1\}^*$ be a language. We say that $L \in \mathsf{SPACE}(S(n))$ iff there exist constants $c_1, c_2 > 0$ and a deterministic Turing machine M deciding L such that, on every input of length n, the machine M visits at most $c_1 \cdot S(n) + c_2$ distinct cells on its read-write work tapes (the input tape is read-only and does not count toward the space bound).

Definition 2.4.3 (Logarithmic space [AB09, Definition 4.5; AB07, Definition 4.5])**.**

$$\mathsf{L} = \mathsf{SPACE}(\log n).$$

Definition 2.4.4 (Polynomial time [AB09, Definition 1.13; AB07, Definition 1.20])**.**

$$\mathsf{P} = \bigcup_{c \geq 1} \mathsf{DTIME}(n^c).$$

Definition 2.4.5 (The class NP [AB09, Definition 2.1; AB07, Definition 2.1])**.**
A language $L \subseteq \{0,1\}^*$ belongs to NP if there exist

(i) a polynomial $p : \mathbb{N} \to \mathbb{N}$ (bounding the length of a certificate); and

(ii) a deterministic polynomial-time Turing machine M (called a *verifier* for L),

such that for every input string $x \in \{0,1\}^*$,

$$x \in L \iff \exists\, u \in \{0,1\}^{p(|x|)} \text{ with } M(x,u) = 1.$$

Whenever $x \in L$ and a string $u \in \{0,1\}^{p(|x|)}$ satisfies $M(x,u) = 1$, the string u is called a *certificate* (or *witness*) for x with respect to the language L and the verifier M.

2.5. Classes of binary circuits

We omit the standard definition of a Boolean circuit (a kind of finite directed acyclic graph of n inputs and m outputs). We also omit the definitions of circuit size, depth, and value; these are discussed in detail e.g. in [Vol99].

Definition 2.5.1 ($\mathsf{NC}^i_{/poly}$, $\mathsf{AC}^i_{/poly}$, $\mathsf{TC}^i_{/poly}$)**.** Fix $i \geq 0$. A language $L \subseteq \{0,1\}^*$ belongs to one of the following circuit classes if there exists a family of circuits $\{C_n\}_{n\in\mathbb{N}}$ such that $C_{|x|}(x) = 1$ iff $x \in L$ and:

(i) every circuit C_n has polynomially many gates w.r.t. n;

(ii) every circuit C_n has depth $\mathcal{O}((\log n)^i)$;

(iii) $L \in \mathsf{NC}^i_{/poly}$ (Nick's class) if each C_n uses only fan-in 2 $\wedge$-, fan-in 2 $\vee$-gates and fan-in 1 $\neg$-gates;

(iv) $L \in \mathsf{AC}^i_{/poly}$ (Alternating circuits) if each C_n uses unbounded fan-in $\wedge$-, unbounded fan-in $\vee$-gates and fan-in 1 $\neg$-gates;

(v) $L \in \mathsf{TC}^i_{/poly}$ (Threshold circuits) if each C_n uses unbounded fan-in $\wedge$-, unbounded fan-in $\vee$-gates, fan-in 1 $\neg$-gates and unbounded fan-in *majority* gates (i.e. a gate that outputs 1 iff at least half of its inputs are one).

Note that the condition $C_n(x) = 1$ iff $x \in L$ requires the circuits to have precisely one output node. We lift this requirement in Definition 2.5.2.

Definition 2.5.2 ($\mathsf{FNC}^i_{/poly}$, $\mathsf{FAC}^i_{/poly}$, $\mathsf{FTC}^i_{/poly}$)**.** A function $F : \{0,1\}^* \to \{0,1\}^*$ belongs to $\mathsf{FNC}^i_{/poly}$(resp. $\mathsf{FAC}^i_{/poly}$, $\mathsf{FTC}^i_{/poly}$) iff there exists a family of circuits with multiple output nodes $\langle C_n\rangle_{n\in\mathbb{N}}$ such that whenever the input bits of C_n encode $X \in \{0,1\}^n$, the output bits encode $F(X)$; C_n satisfies the size and depth conditions (i), (ii) of Definition 2.5.1; and additionally C_n satisfies the class condition (iii) (resp. (iv), (v)) of 2.5.1.

Definition 2.5.3 (L-uniform circuit families)**.** We say that a family of circuits $\langle C_n\rangle_{n\in\mathbb{N}}$ is L-uniform if there exists a Turing machine operating in space $\mathcal{O}(\log n)$, computing the function $1^n \to C_n$ for some representation of C_n.

Definition 2.5.4 (NC^i, AC^i, TC^i, FNC^i, FAC^i, FTC^i)**.** A function $F : \{0,1\}^* \to \{0,1\}$ belongs to NC^i(resp. AC^i, TC^i) iff there exists a L-uniform family of circuits satisfying the conditions from Definition 2.5.1. A function $F : \{0,1\}^* \to \{0,1\}^*$ belongs to FNC^i(resp. FAC^i, FTC^i) iff there exists a L-uniform family of circuits satisfying the conditions from Definition 2.5.2.

Chapter 3

Models of computation, programming paradigms and complexity measures

One of the first decisions a programming language designer has to make is choosing the programming paradigm convenient for writing the programs of interest. In the practice of programming, imperative languages have no competition when a user needs to reason about the computational complexity of programs. The structure of imperative programs closely mirrors how the computation is executed on modern CPUs and GPUs. In turn, it is inherently unintuitive to reason about the complexity of programs written e.g. in Haskell or Prolog.

Yet if we want to understand what classes of functions can be characterized syntactically, we have to temporarily step away from the imperative mindset. As argued in [Gur12], the very notion of an "algorithm" is still evolving, so we shouldn't limit our considerations to a single paradigm. This chapter tests whether alternative models of computation could be better suited to form the basis of languages that capture popular complexity classes. This is studied in much more detail in [AB09, Section 1.6.3; AB07, Section 1.5.2], where also complexity of randomized computation, quantum computation and computation on real numbers (as opposed to bits) is considered.

We can foreshadow that the answer is (perhaps surprisingly) positive: even though the complexity classes are defined on Turing machines, the characterizations studied in literature are rarely imperative.

Remark 3.1. In most complexity discussions we speak about decision problems, but real programs usually return structured data rather than a single bit. To keep that practical viewpoint, we favour computational models that natively handle producing structured output such as a binary string. We will still mention purely decisional models whenever that is the natural formulation.

3.1. Finite automata and transducers

The existing research on finite-state automata has not been directly useful for our work. The expressiveness of this model is inherently limited to Boolean-valued functions, so for general functions it is strictly better for us to focus on transducers. Yet, we consider some characterization of REG in Section 5.4. Please also see [Coo71, Corollary 1 and Definition in section 2] for the characterization of P by two-way multi-head pushdown automata and [Sud77, Lemma 2] for proof that L languages are characterized by two-way multi-head finite automata.

Transducers extend automata to binary string outputs and have elegant characterizations. In [Boj18], four characterizations are given for the class of so-called *polyregular* functions. The definitions described there readily constitute the basis of a programming language. Another programming language for transducers is studied in [Sch05]. A particular class of string-to-string functions defined using logic, *MSO transductions*, is characterized to be precisely the class of functions computable by two-way deterministic finite transducers (2DFT) in [EH99]. All of these results provide basis for new programming languages.

Because expressiveness of transducers remain poorly understood, it is usually unclear whether an arbitrary problem belongs to the class recognized by a given flavor of transducer. Writing programs as transducers would rarely be useful for certifying an arbitrary program to be in the desired complexity class, hence for now we focus on other styles of characterization. A good overview of existing research on transducers can be found in [MP19].

3.2. Turing machines

This model of computation underpins the imperative programming style. There is a variety of flavors of Turing machines, and the details of a specific definition will most of the time not affect our considerations in this work. When not explicitly stating otherwise, when describing a computational process we will implicitly assume the realization of it on some kind of a Turing machine. As we will see in Chapter 6, the most popular complexity classes were specifically defined in terms of computation on Turing machines. The obvious approach to our task of obtaining certificates of complexity bounds would be to start from the bare definition of a Turing machine, and meticulously formally proving properties of more and more complex Turing machines. Yet, this approach fails miserably, as we will have a chance to explore in Chapter 4.

Surprisingly few elegant, high-level imperative languages are known to characterize well-studied complexity classes beyond the trivial ones. This scarcity is precisely why much of the work surveyed in later chapters relies on non-imperative paradigms. For a thorough overview of the computational complexity of imperative programs, see [KN04].

3.2.1. Random-access Turing machines

Random-access Turing machines are a version of Turing machines that allow to read from and write to a specific memory cell in one step, if only the address of the cell has been properly encoded on a special address tape. Reasoning about computation in complexity classes such as logarithmic space or polynomial time is the same for traditional Turing machines and random-access Turing machines. The choice of model starts to matter for *fine-grained* complexity classes such as the class of problems solvable in linear or quadratic time — since a standard Turing machine can only move its head one cell at a time, obviously there are problems that will take it more than linear time, while being solvable in linear time by a random-access machine. Due to insufficient existing research on implicit characterizations of fine-grained complexity classes, we will not consider them besides a brief discussion in Remark 6.7; also, the notion of DLOGTIME-uniformity explored in Section A.3 is only defined for random-access Turing machines.

3.3. Circuits

Circuits as a model of computation are the theoretical foundation of parallel programming. The theoretical implications turn out to not be widely applicable in practice, however. In this work, we will mostly use circuits to reason about very weak complexity.

A very rich overview of different characterizations of circuit complexity classes is in [ADK25]. A particular logical characterization of AC^0 (Definition 2.5.4) will be important for us in Chapter 8.

3.4. Discrete differential equations

An original point of view on computation is to describe functions as solutions to discrete differential equations. For example, in [BD19], FP (Definition 6.2.2) and FNP (Definition 6.2.3) are characterized. Characterizations based on differential equations of various circuit complexity classes from FAC^0 to FAC^1 (Definition 2.5.4) are described in [ADK25].

3.5. Logic programming and descriptive complexity

Logic provides very deep complexity-theoretic connections, primarily through descriptive complexity theory, which we explore in more detail in Chapter 5.

3.6. Untyped recursion

Another classical paradigm is that of general recursive functions, or equivalently the untyped lambda calculus. Because these systems are Turing complete, the interesting question is how to constrain recursion so that the resulting language captures a specific class. We treat these ideas in Section 7.2.

3.7. Typed lambda calculus

Typed lambda calculus underpins functional programming. In this section we focus on typed variants, unlike in Section 3.6.

Lambda calculus does not line up cleanly with traditional Turing machine complexity measures. For instance, for some representation of strings $\{0,1\}^*$, [HK96, Theorem 3.4] identifies the functions $\{0,1\}^* \to \{0,1\}$ definable in simply typed lambda calculus (STLC), with regular languages. But with a different encoding of inputs, [HKM96] relates STLC to the whole $\mathsf{ELEMENTARY}$ class. Moreover, [Zak07] states that with a different "standard" encoding, STLC instead characterizes extended polynomials, and further shows that if we slightly modify the encoding, the class is yet different. For more discussion, see also [Maz18]. Consequently, it is not obvious how to reason about complexity theory in the language of lambda calculus.

Nevertheless, typed lambda calculi have been utilized very successfully to syntactically characterize complexity classes. Recall that one of the reasons linear logic is studied is the potential to reason about resource creation and utilization. Concepts from linear logic have been implemented in the theory of type systems to transfer the resource interpretation. This will be discussed further in Section 7.3.

3.8. Set theory as inspiration for model of computation

An interesting connection appears when we think of traditional notions of "complexity" of sets in set theory from the point of view of computational complexity. If we treat taking complements, intersections, countable unions as operations in a programming language, perhaps we could design a programming language for constructing sets. By itself such a language would probably not be the most interesting one. However, thinking about mathematical reasoning in terms of a computational process is a very powerful technique. It has been particulary deeply explored for connecting logic and lambda calculus under the name of Curry-Howard or proofs-as-programs correspondence.[1] For the computational content of set theory in particular, an interesting brief discussion is presented in [Tao10].

For our purposes, an interesting connection appears when we look at descriptive set theory. The most basic classes of sets distinguished there are open and closed sets. Slightly higher, an F_σ set is a countable union of closed sets; a G_δ set is a countable intersection of open sets. A few levels higher up the *Boldface hierarchy*, which in a way quantify the complexity of sets, Borel sets are considered:

Definition (Borel sets)**.** Let X be a topological space. The class of Borel sets of X, $\mathcal{B}(X)$ is the smallest class of sets containing every open set of X and closed under (i) and (ii):

(i) if A is Borel, then its complement $X \setminus A$ is Borel;

(ii) if A_n is Borel for each $n \in \mathbb{N}$, then the countable union $\bigcup_{n\in\mathbb{N}} A_n$ is Borel.

As this is a standard least fixed point definition, it is strikingly similar to definitions of classes of recursive functions that we will consider later, e.g. Definition 7.2.1. We can also take a computational point of view on theorems about the determinacy of Gale-Stewart games (which we shall not introduce here). It is widely known that Gale-Stewart games are determined when the underlying set is open or closed. Allowing the underlying set to be more and more complicated, we quickly reach the limits of provability in Zermelo-Fraenkel set theory: determinacy for Borel sets is a difficult theorem, and determinacy for analytic and projective sets is independent of ZF, yet provable assuming as axiom the existence of an appropriately large cardinal.

A good question to ask is if by carefully curating the axioms, we could obtain a mathematical theory such that theorems corresponding to computation in our desired complexity class are provable, and the theorems that wouldn't be "implementable" are not.

We will circle back to this intuition while considering the `PIGEON` computational problem in Subsection 6.2.2 and the unprovability of the related pigeonhole principle in the weak theories studied in Theorem 8.11. Especially the last fact about unprovability is interesting for us in this section, as it resembles e.g. independence of continuum hypothesis from ZFC, but in a strictly computational setting of not being able to perform enough computation in a low complexity class. The forcing technique used originally by Cohen to prove the mentioned independence, turns out to also be useful for the theory of computation, as discussed in Section 6.2.6.

While this line of thought is very far from "practical programming languages", these considerations inspired[2] the approach that we study in Chapter 8.

[1] A good introduction to the immensely deep topic of proofs-as-programs is [SU06].

[2] This line of study grew out of presenting the topic at the JAiO master's seminar at the University of Warsaw — slides are available online at [Bal25].

Chapter 4

Formalized semantics

One approach to certify complexity class of a program would be to simply accept an implementation of the algorithm in C++ accompanied by a proof that the number of "steps" of the algorithm when executed on a standard computer is bounded by a polynomial p. For at least three reasons, such a language would not be satisfactory for us. We explore them in this chapter.

Informality of language semantics It is not actually defined how much time will an arbitrary C++ program take to execute. A famous shortcoming of C++'s standard is the notion of *undefined behaviour*. As it turns out, close to none programming languages have well-defined semantics. For the vast majority of languages, the only reliable semantics is the source code of the most popular compiler. There are, however, programming languages with formally defined semantics. Most notably, a large fragment of the C programming language has been formalized in the CompCert project [Ler09]. CakeML [Kum+14] has formal semantics and a proven-correct compiler. A fragment of OCaml has been formalized in [Sea+25].

Representation of a proof in computer-verifiable form Even if someone gives us a program with a well-defined semantics and the proof required, it is not obvious how to check that the proof is correct. A system in which it is convenient to write mathematical proofs, in a way that they can be checked by the computer, and the code of the checker is simple enough to be widely believable, is called a *proof assistant*. The successful projects such as Mizar, Rocq, Lean and Isabelle/HOL took decades of work to be developed. Only in the last decade, the most popular proof assistants got to the stage where research-level reasoning can be transferred to them *with reasonable overhead*.

In a separate but related field, a very interesting class of programming languages rely on automated theorem proving to verify user-provided pre- and post-conditions of functions. Dafny [Lei10] (see: Listing 4.1) does it by utilizing *SMT-solvers*, not allowing the user to conduct a mathematical proof manually. Why3 [Bob+11] provides a comprehensive framework, combining the power of SMT-solvers for automation where possible, and allowing the user to conduct proofs manually where they fail. Similarly, Liquid Haskell extends Haskell with functionality allowing the user to specify correctness properties using liquid types [RKJ08]. A recent work adapts the same liquid-type approach to Rust [Leh+23].

Infeasibility of formal proofs about Turing machines Even while operating in the right programming language, and assuming trust in the modern proof assistants, we will still not be able to proceed this way. The field of computational complexity and computability theory

```
function fib(n: nat): nat
{
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)
}
method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
  var i := 1;
  var a := 0;
  b := 1;
  while i < n
    invariant 0 < i <= n
    invariant a == fib(i - 1)
    invariant b == fib(i)
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```

Listing 4.1: Dafny example

has a reputation of being particularly hand-wavy. This naturally generates difficulties when attempting to formalize the proofs from these areas. Voluminous discussion on that problem emerged with the rise of popularity of proof assistants. The below quote is due to Yannick Forster, author of the most successful works on formalizing results about computability:

> making these arguments [i.e. about Turing machines] formal is several orders of magnitude more involved than formalising other areas of mathematics, due to the amount of invisible mathematics (a term coined by Andrej Bauer) involved. [For25]

Formally defining Turing machines has mostly been considered in [AR12] and, building on top of it, [FKW20][1]. Besides that, few examples of concrete Turing machine definitions have been described in the literature, as working with them directly is greatly time-consuming and not incentivized by academia. Interesting concrete examples were provided in[Kud96], [Rog96] and in the work we discuss in Section 4.1. Alternative approaches are studied as Synthetic Computability [Bau06].

In Remark 7.12, we briefly discuss one formalization of *a characterization* of polynomial-time functions.

4.1. Ciaffaglione's formalization of undecidability of HALT

In [Cia16], an elegant formalization of the undecidability of the halting problem is presented. The proof goes by defining simple Turing machines needed and directly proving results about

[1]Part of an ongoing formalization project: `https://github.com/uds-psl/coq-library-undecidability`.

their semantics.[2] As part of our work, we refined the code to a newer Coq version, fixing the proofs where necessary. Full code (also presented on the JAiO seminar) is available online with installation and verification instructions.[3]

TODO: ensure code didn't move

[2] As of November 2025, the code referred to in the paper is not accessible anymore.

[3] Code and slides are available at: `https://github.com/ruplet/prezentacja-seminarium-1`.

Chapter 5

Descriptive Complexity

In the rest of this thesis, we usually measure complexity of algorithms: how much time and memory will a given algorithm need to run to solve a problem of size n? In this chapter we will instead focus on the complexity of defining the problem itself.

The central problem of Descriptive Complexity is to characterize a complexity class by the *power of logic required to define its problems*. This is in contrast to *Implicit Complexity Theory*, discussed later in Chapter 7, which seeks the weakest system sufficient to *implement algorithms* of the class; and in contrast to *Bounded Arithmetic*, which studies the weakest *theory* required to define a function *and prove its correctness*. This perspective has led to elegant logical characterizations of many traditional complexity classes, as we will discuss in Section 5.1.

Before going into details, we will discuss an example to recall what it means for a structure to model a formula (Definition 2.1.7):

Remark 5.1. In Descriptive Complexity, there is no notion of a proof. The problem of deciding if a given sentence is provable is independent of this chapter's considerations, and will be studied by us only in Chapter 8.

Example 5.2. Consider a vocabulary $\mathcal{L}$ consisting of unary relations $\mathrm{Zero}(x), \mathrm{One}(x)$ and binary relations $=, \leqslant$. If we think of positions in a binary string as elements of the universe, we can define e.g. for a string 01011 an $\mathcal{L}$-structure $\mathcal{M}$ with:

(i) universe $\{1, 2, 3, 4, 5\}$;

(ii) $\mathrm{Zero} := \{1, 3\}; \mathrm{One} := \{2, 4, 5\}; = := \{\langle 1, 1\rangle, \langle 2, 2\rangle, \ldots, \langle 5, 5\rangle\}; \leqslant := \{\ldots\}$.

Now, we can reason about the original binary string using logic. The formula $\exists x.\, \mathrm{One}(x)$ is true in $\mathcal{M}$. However, the formula $\forall x.\, \mathrm{Zero}(x)$ is not. It is insightful to analyze in general what kind of logical formulas are true in $\mathcal{M}$.

We can use the language of logic to describe computational queries about the underlying structure. This is the intuition behind the languages designed for querying databases. To formally connect logic and computation, consider the following

Definition 5.0.1 (Language of binary strings)**.** Consider a vocabulary τ_{string} to contain only the unary relations $\mathrm{Zero}(x), \mathrm{One}(x)$ and the binary relations $=, \leqslant$. Intuitively, $x \leqslant y$ means that memory cell x comes before memory cell y. As we can represent most of the real-world structures of interest as some binary string on a computer, this simple vocabulary already allows us to ask interesting questions.

Definition 5.0.2 (Language generated by a formula)**.** For a binary string $w \in \{0,1\}^n$ define its corresponding structure $\mathcal{M}_w$ over vocabulary τ_{string} to have the universe $M := \{1, \ldots, n\}$ and the standard semantics of relations from τ_{string}.

We define the language L_φ to be *generated* by a first-order sentence φ over τ_{string} iff

$$\mathcal{M}_w \models \varphi \iff w \in L_\varphi.$$

Now let's shift our focus to a more standard definition.

Definition 5.0.3 (First order logic on ordered structures)**.** Define the vocabulary $\tau_\leqslant$ to contain:

(i) the constants $0, 1, \max$;
(ii) the unary relations $\text{Zero}(x), \text{One}(x)$;
(iii) the binary relations $=, \leqslant$.

$\mathsf{FO}_\leqslant$ is the class of sentences of first-order logic in the language $\tau_\leqslant$.

From now on, whenever talking about the semantics of any such sentence, we will require the elements of the (finite) universe to be interpreted as actual natural numbers $0, 1, \ldots$; the equality and order to be interpreted as the actual equality and order on the elements of the model; $0, 1, \max$ to be interpreted as the minimum, second, and maximum elements under $\leqslant$. Additionally, we will use the unary relations $\text{Zero}(x), \text{One}(x)$ to interpret the elements of the domain as positions of some input binary word to talk about deciding if a formula holds in a given word.

Remark 5.3 (Bibliography). In literature, $\mathsf{FO}_\leqslant$ is called $\mathsf{FO}(\mathsf{wo\ BIT})$; see [Imm99, Ordering Proviso 1.14].

Definition 5.0.4 (First order logic with arithmetical predicates)**.** FO_{BIT} is the class of sentences of first-order logic over $\tau_\leqslant$ extended with the binary relation BIT. Semantically, we require $\text{BIT}(x, y)$ to hold iff bit y in the binary representation of x is 1.

Remark 5.4. By default in the literature, the arithmetic predicates and order are included. Usually, arithmetic predicates $\text{PLUS}(x, y, z)$, $\text{TIMES}(x, y, z)$, $\text{SUCC}(x, y)$, denoting that $x + y = z, x * y = z, x + 1 = y$ respectively, are also added. We use the fact that PLUS, TIMES are first-order definable from BIT [Imm99, Theorem 1.17] and that SUCC is first-order definable from $\leqslant$ [Imm99, Section 1.2].

5.1. Results

The field of descriptive complexity has a long history and we should not introduce all the results in this work, as our primary purpose is to examine if, in the first place, they are relevant to our problem. We will only describe the logical characterizations of the classes that are the most interesting for us: P and uniform AC^0 which will underpin our Chapter 8. A concise overview of the classical results is presented in [Imm99, Section 15.1], where characterizations of $\mathsf{SPACE}(n^k)$, L, NL, P, NP and PSPACE are described. Please also see the insightful [Imm87] that introduces model-theoretical characterizations of numerous decisional complexity classes.

One intensely studied logic is $\mathsf{FO}_{\mathrm{BIT}}[\mathsf{LFP}]$, defined inductively.

Definition 5.1.1 ([Imm99, Definition 4.5] Least fixed-point logic)**.** $\mathsf{FO}_{\mathrm{BIT}}[\mathsf{LFP}]$ is a class of logical sentences such that:

(i) if $\varphi \in \mathsf{FO}_{\mathrm{BIT}}$ then $\varphi \in \mathsf{FO}_{\mathrm{BIT}}[\mathsf{LFP}]$;

(ii) if $\varphi_R(x_1, \ldots, x_k) \in \mathsf{FO}_{\mathrm{BIT}}[\mathsf{LFP}]$, where R is a k-ary relation and R only occurs positively in $\varphi_R(x_1, \ldots, x_k)$ (i.e. every occurrence of R is preceded by an even number of negations), then $\mathsf{LFP}_{R(x_1,\ldots,x_k)}\varphi$ may be used as a new k-ary relation symbol denoting the least fixed-point of φ.

Theorem 5.5 ($\mathsf{FO}_{\mathrm{BIT}}[\mathsf{LFP}] = \mathsf{P}$)**.** *The class of languages generated by sentences from* $\mathsf{FO}_{\mathrm{BIT}}[\mathsf{LFP}]$ *is precisely* P*.*

Remark 5.6. We can think of the LFP operator as allowing us to write recursive formulas.

Remark 5.7. This result has been proved independently by Immerman [Imm86] and Vardi [Var82]. For a more uniform treatment, see [Imm99, Theorem 4.10].

Theorem 5.8 ([Imm99, Corollary 5.32] $\mathsf{FO}_{\mathrm{BIT}} = \mathsf{FO}$-uniform AC^0)**.** *Languages* $L \subseteq \{0,1\}^*$ *decidable by uniform* AC^0 *circuits are precisely the languages generated by* $\mathsf{FO}_{\mathrm{BIT}}$ *formulas, in the sense of Definition 5.0.2.*

Remark 5.9 (Bibliography). The theorem is originally stated in terms of $\mathsf{FO}_{\mathrm{BIT}}[\mathsf{t}(\mathsf{n})]$ defined in [Imm99, Definition 4.24]. We limit to $t(n) = \mathcal{O}(1)$, i.e. in our case $\mathsf{FO}_{\mathrm{BIT}}[\mathsf{t}(\mathsf{n})] = \mathsf{FO}_{\mathrm{BIT}}$. The uniformity condition is also different to the one we fixed in Definition 2.5.3: their circuits are so-called FO-uniform, which is a much stronger condition. For details about the different notions of uniformity, refer to Appendix A.

5.2. The Quest for a Logic Capturing PTIME

Many of the problems in P are graph problems, i.e. they ask to decide a property $\varphi(G)$ for some abstract graph G. It is natural to represent a graph as a logical structure: the nodes of the graph correspond to the elements of the universe and we add to the vocabulary a special relation $E(x, y)$ denoting there is an edge from x to y. This is reflected by the following

Definition 5.2.1 (Logic on graphs)**.** Define the vocabulary τ_{graph} to contain only the binary relations $x = y, E(x, y)$. The class $\mathsf{FO}_{\mathrm{graph}}$ contains precisely the sentences of first-order logic over the vocabulary τ_{graph}.

From now on, when talking about the semantics of sentences from $\mathsf{FO}_{\mathrm{graph}}$, we will assume the interpretation of elements of the universe as nodes of the graph G, and the interpretations of $x = y$, $E(x, y)$ to agree with the underlying node equality and edge relation of G. For example, we will assume that $E(x, y)$ holds if and only if there is an edge between nodes of G corresponding to the universe elements x, y.

Remark 5.10 (Bibliography). Typically, the vocabulary of logic on graphs also contains unary symbols $a(x), b(x), \ldots$ denoting that the color of node x is a or b etc. [Imm99, Definition 12.2; Imm99, Theorem 1.36]. As we will not consider coloring of nodes in this thesis, we don't need to introduce that. In the literature, $\mathsf{FO}_{\mathrm{graph}}$ is typically called $\mathsf{FO}(\mathsf{wo} \leqslant)$, modulo the addition of the edge relation. For more details, see [Imm99, Ordering Proviso 1.14] and also discussion under [Imm99, Question 12.1].

Defining graph problems rarely requires us to impose any numbering on the nodes of the graph. However, to talk about deciding a property of a graph on a Turing machine, we need

to encode the graph as a binary string. This imposes some artificial ordering on the vertices of the graph. Perhaps a more suitable definition of logic of graphs would thus be

Definition 5.2.2 (Logic on graphs with order)**.** We denote by $\mathsf{FO}_{\text{graph}\leqslant}$ the class of first-order sentences over τ_{graph} extended with the constants $0, 1, \max$, the binary relations $\text{BIT}, \leqslant$ and the operator LFP, interpreted as earlier.

Now, let's consider adding the least fixed-point operator to logic on graphs. We will denote by $\mathsf{FO}_{\text{graph}}[\mathsf{LFP}]$ the logic obtained by extending $\mathsf{FO}_{\text{graph}}$ with the LFP operator. For the $\mathsf{FO}_{\text{graph}\leqslant}$ with LFP, we need to notice two things. First, we refer to [Imm99, Proposition 9.16] stating that the relation BIT is first-order definable with ordering and LFP. Second, we notice that the addition of the binary edge relation doesn't change the expressive power here. Indeed, with ordering we can encode the edge relation as part of input. Thus, we will treat this logic as equally strong to FO_{BIT} introduced earlier. In particular, we will assume without transferring the proof of Theorem 5.5 that $\mathsf{FO}_{\text{graph}\leqslant} = \mathsf{P}$. Thus, we obtain two similarly defined theories: $\mathsf{FO}_{\text{graph}}[\mathsf{LFP}]$, not having access to the order relation, and $\mathsf{FO}_{\text{graph}\leqslant}[\mathsf{LFP}]$, only operating on ordered structures.

For graph problems we typically want the Turing machine M to return the same answer regardless of the order we pass the vertices in. However, when looking at the code of a particular Turing machine, it's usually difficult to tell if it returns the same answer for all permutations of the input graph. The implicit assumption of being always given *some* ordering of input graph nodes, is inherent in computation on Turing machines. We may now wonder: does this assumption limit us in the generality of programs we write? If someone forced us to not rely on this ordering in our programs, and write programs in a *permutation-invariant* way, would anything change in our expressive power? This turns out to be a very deep and difficult question.

We will say that a graph problem P is in the complexity class $\mathsf{inv\text{-}P}$ iff there is a polynomial-time Turing machine M such that for every graph G and every permutation π of vertices of G,

$$M(\text{enc}(\pi(G))) \leftrightarrow P(G).$$

That is: the problem $P(G)$ has such a decider M that it returns the correct answer regardless of how we label the input nodes. The class $\mathsf{inv\text{-}P}$ is also called P *on unordered structures.*

As motivated earlier, $\mathsf{FO}_{\text{graph}\leqslant}[\mathsf{LFP}]$ is strong enough to express any property from P. However, it turns out that when we take the order out, $\mathsf{FO}_{\text{graph}}[\mathsf{LFP}]$ **cannot** express every problem from $\mathsf{inv\text{-}P}$. Equivalently: that $\mathsf{FO}_{\text{graph}}[\mathsf{LFP}]$ cannot capture P [CFI92]. This means that there are graph problems that have a robust, order-invariant decider M, yet can't be expressed by a logical formula having access to the LFP operator, but not having access to the arithmetical predicates and ordering. The existence of a logic that characterizes $\mathsf{inv\text{-}P}$ (or P on unordered structures) remains a major open problem in computer science as of 2025. Good overviews of this problem are [Daw12] and [Imm99, Chapter 12, The Role of Ordering].

An important result that treats unordered structures is Fagin's theorem [Fag74] that states that the class of languages generated by sentences of existential second-order logic is precisely NP. Intuitively, ordering of the domain is not necessary to assume for Fagin's theorem because in NP, we can *guess* it.

5.3. Defining functional problems in logic

5.3.1. First-order queries (FO-reductions)

Despite logic being only able to naturally define decisive problems, some approaches have been used to reason logically about functions. Most importantly, to study completeness of problems in low complexity classes such as L, FO-reductions are used. They have a rather complicated definition, which we don't display here and for the details refer to [Imm86, Definition 1.26]. In the same work, even weaker notions of reducibility are studied: first-order projections (fops) and quantifier-free projections (qfps) are defined in [Imm99, Definition 11.7]. An interesting property of the complexity classes we are studying in this work is that their complete problems are already complete under surprisingly weak reductions. In [Imm99, Proposition 11.10] it is proved that SAT is NP-complete via fops and even via qfps.

5.3.2. Bit-graph definitions

An easy way to define a general function from Boolean functions is to use the Boolean functions to decide if "i-th bit of the output $f(x)$ is 0 or 1". For the definition to be precise, two more technicalities are needed (the output's length itself must be polynomial and easy to compute), which we will discuss while defining L-reductions in Definition 6.1.1. For now, we will just say that this style of definitions is very important for some of the results we will study in Chapter 8, as the "semantic" definition of definability of functions ([CN10, Definition V.4.12; CN08, Definition 5.37]) is precisely that.

5.3.3. FO and MSO transductions

Some works use the notion of FO-transductions, e.g. in [NMS21, Section 2], where they are also defined. They are, however, completely different from anything we study in this work and are defined as compositions of *copying*, *coloring* and *simple interpretations*, which we will not discuss. A similar, and much more important notion is of MSO transductions defined e.g. in [Cou94, Section 2]

5.4. Descriptive Complexity and programming languages

From the point of view of programming languages, the meaning of e.g. the result discussed in Section 5.2 is as follows: given a logical formula $\varphi \in \mathsf{FO}_{\mathrm{BIT}}[\mathsf{LFP}]$, we *know that there exists some* Turing machine of complexity P that will check for us if $w \models \varphi$ for any input w. There is a huge leap of faith, however — just that the machine *exists* and runs fast, we can't conclude that we will ever be able to actually use it. We have to inspect the proof of such a theorem and tell if we can compute the description itself of such a machine, and how fast we can do it. An illustrative example is infeasibility of using the below theorem to design a programming language for finite automata:

Theorem 5.11 ([Imm99, Theorem 1.36] Büchi-Elgot-Trakhtenbrot theorem)**.** *The set of finite models of boolean queries expressible in second-order monadic logic (which we skip the definition of) over the vocabulary $\tau_{\leqslant}$ is exactly the set of regular languages. In other words,* $\mathsf{MSO} = \mathsf{REG}$.

Remark 5.12 (Bibliography)*.* The theorem was originally proved by Büchi in [Büc60], Elgot in [Elg61] and Trakhtenbrot in [Tra62]. The statement from [Imm99, Theorem 1.36] is slightly

```
% Run at: https://swish.swi-prolog.org/
succ(bit1, bit2).
succ(bit2, bit3).
succ(bit3, bit4).
succ(bit4, bit5).

zero(bit1).
zero(bit3).
one(bit2).
one(bit4).
one(bit5).

exists_1  :- one(X).                              % true
exists_00 :- succ(X, Y), zero(X), zero(Y).        % false
no_00     :- \+ ( succ(X, Y), zero(X), zero(Y) ). % true
no_11     :- \+ ( succ(X, Y), one(X), one(Y) ).   % false
```

Listing 5.1: Prolog example

more accessible. It uses slightly different wording to ours. For the details of the definitions, refer to [Imm99, Proviso 1.14] and [Imm99, Proviso 1.15].

However, not always is this problem that difficult. Results from descriptive complexity have been crucial for the field of database theory, which we don't discuss here. The approach of using logic for programming has also coined the paradigm of logic programming.

5.4.1. Logic programming

The most famous example of a logic programming language is Prolog. Prolog doesn't have a reputation of a language well-suited for computational complexity analysis. However, it relates very well with the notions of deciding computational problems we discussed in this chapter. Please see Listing 5.1 for a demonstration how we can transfer our considerations from Example 5.2 to practical computation.

5.4.2. Datalog

Datalog is a programming language that uses the paradigm of logic programming, similarly to Prolog. Unlike Prolog, however, it *captures* the complexity class of P [Dan+01, Theorem 4.4] (on ordered, finite structures). That is, it is a language such that any query definable in it will be checkable in P for a given model. We have not investigated the subtleties of Datalog's implementation and whether the complexity doesn't blow up somewhere.

5.4.3. Logic as the type system

Could the results from descriptive complexity be used to design a specification mechanism (such as a type system[1]) for a programming language? While such a language would be very interesting and possibly have highly desired properties, our "typechecker" in the naive case would essentially be a proof checker for first order logic. This would be a very difficult system

[1] Language doesn't have to be typed to foster partial proofs of correctness; see e.g. [LP99], [Pau00].

to design properly. We were not successful in curating such a small set of basic programming instructions that, given a pre- and a post-condition in FO and an operation from that set, checking for the validity of such a triple would be an easy problem.

Nonetheless, this line of inquiry led to a broader investigation of type systems that enforce resource bounds — particularly those inspired by linear logic and implicit computational complexity. The results of this exploration are presented in Section 7.3.

Chapter 6

Reductions

Plenty of theorems of the form "problem P is *complete* for class C under reductions in class R" have been described in the literature. In this chapter we analyze when such reductions help capture complexity classes syntactically and when they fall short.

Theorem 6.1 ([Lad75])**.** *Define the circuit value problem (CVP) as the problem of finding the output of a Boolean circuit, given its representation as input. The circuit value problem is complete for* P *under* L *reductions.*

Remark 6.2. We define L-reductions formally in Definition 6.1.1. For the details of representation of Boolean circuits, see [Lad75].

Knowing Theorem 6.1, we could design a language for P in such a way: first, design a language for L which is (perhaps) simpler; then, as the compiler provider, include a standard library function P, which the L-functions from the language could call like an oracle to get a solution for P. In this chapter we try to understand if such a language would truly have the full power of polynomial-time functions. The core of this chapter is Theorem 6.10, stating that this holds at least for the class of logspace-computable functions and a particular notion of weak reductions. But the heart of this intuition is not developed until Theorem 8.9, where we get such characterizations for most of the complexity classes that we consider in this thesis.

6.1. Decisional versus functional complexity classes

We can't answer the question of expressive power of CVP + L-reductions by comparing it with any decisional class. Focusing solely on the complexity of Boolean functions as we did for now e.g. in Section 2.4 is not sufficient to reason about general functions with output. In this chapter we will introduce the still standard, but less talked about, *functional* complexity classes, that study the complexity of computing general functions $f : \{0,1\}^* \to \{0,1\}^*$ as introduced in Section 2.3. In complexity theory these are usually implicitly used to define *reductions*. The results about these classes transfer much better to our interests than results about the decisional complexity classes. As we discuss in Section 7.2, programming-language-like characterizations of decisional complexity classes are far more abundant and predate the characterizations of functional complexity classes. It is the latter, however, that is viable for our purposes of designing a programming language.

A thorough overview of complexity classes of functions is described in [Sel94]. A more thorough discussion of decision vs search is in [BG92].

Exponential-length output The difference between P and FP is obvious when we look at the below example:

Example 6.3. Running time of any Turing machine computing the function $x \to 2^x$ for input and output in binary is exponential. At the same time, given an input x, y, checking if $y = 2^x$ is easily in polynomial time.

This problem is, however, usually artificially mitigated by requiring the length $|f(x)|$ of the function's output to be polynomially bounded as in Definition 6.1.1.

Self-reducibility A common approach in the literature way of defining function problems is to require an external proof that $|f(x)|$ is polynomially bounded, then repeatedly decide if i-th bit of $f(x)$ is 0 or 1. For example, let's look at the below

Definition 6.1.1 ([AB09, Definition 4.16; AB07, Definition 4.14] L-reductions)**.** A function $f : \{0,1\}^* \to \{0,1\}^*$ is said to be *logspace computable* if

(i) the function f is *polynomially bounded*, meaning that there exists a constant $c > 0$ such that
$$|f(x)| \leq |x|^c \qquad \text{for all } x \in \{0,1\}^*;$$

(ii) the following two languages lie in L:
$$L_f = \{\, \langle x, i\rangle \mid f(x)_i = 1 \,\}, \qquad L'_f = \{\, \langle x, i\rangle \mid i \leq |f(x)| \,\}.$$

In other words, a deterministic $\mathcal{O}(\log|x|)$-space machine can, given (x, i), determine whether i is within the length of $f(x)$ and, if so, whether the i-th bit of $f(x)$ is 1.

A language B is *logspace reducible* to a language C, if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$ that is logspace computable and such that $x \in B$ iff $f(x) \in C$ for every $x \in \{0,1\}^*$.

Famously, we can also apply this trick to solve the problem `FSAT`, the corresponding functional problem to the `SAT` of finding a specific satisfying assignment with polynomially many calls to the decision procedure `SAT`. In general, many functional problems are solvable in polynomial time with polynomially many calls to their corresponding decisional problems. We say that problems with that property are *self-reducible*.

However, some search problems are unlikely to be self-reducible. A good example is the problem of integer factorization, which is still, as of November 2025, conjectured to not be in P even despite the breakthrough result from 2002 in which `PRIME` (decide if n is prime) was proved to be in P [AKS04]. A particularly important class of such problems is considered in Subsection 6.2.2. But first, let's define the classes FP and FNP.

6.2. Functional complexity classes

Definition 6.2.1 ([AB09, Section 17.2; AB07, Section 9.1] The class FP (Version 1))**.** FP consists of all functions
$$f : \{0,1\}^* \to \{0,1\}^*$$
that are computable by a deterministic polynomial-time Turing machine. In contrast to decision problems (which output a single bit), functions in FP may produce outputs of arbitrary polynomial length.

Remark 6.4. This definition is used e.g. in [Coo85] besides the work cited above. It is rather equivalent to Definition 6.2.2 below, also ubiquitous in the literature.

Definition 6.2.2 (FP (Version 2))**.** A binary relation $P(x, y)$ is in FP iff there exists a polynomial-time Turing machine M that, given an arbitrary input x:

(i) outputs some y such that $P(x, y)$ if any exists;

(ii) signals that no such y exists otherwise;

and additionally there exists a polynomial-time Turing machine (a "verifier") that, given an arbitrary input pair (x, y), decides if $P(x, y)$.

Remark 6.5. This definition is inspired by [Ric07, Section 28.10], where it is defined without the verifier requirement. However, this requirement is crucial. Let's consider two relations. Let it be undecidable if $H(x, y)$ for given x, y. Let $R(x, y)$ be such that $R(x, y) \iff (H(x, y) \vee y = 0)$. Notice that for every x, we can trivially find *some* y such that $R(x, y)$. But it is undecidable to test if $R(x, y)$ given an arbitrary input pair (x, y). Such a complexity class would not be contained in FNP for a sensible definition of that class (which we'll study later).

Our version makes it elegant to compare FP with FNP (defined later) — but only assuming that FNP is defined using nondeterministic Turing machines, which is not true in our case; we will use the *verifier*-style definition.

Remark 6.6 (P vs FP). These two classes are often identified due to similar properties. The notion of completeness for both of them, despite being differently defined, is practically the same due to the P computations being closed under being repeated for every bit of the output.

This conflation already appears in Stephen Cook's 1982 ACM Turing Award lecture [Coo83, Section 6], where the distinction between P-completeness and FP-completeness is not made explicit. The three examples cited there as establishing FP-completeness of certain functions $f(x)$ in fact only prove P-completeness of the associated decision problems of the form: given x and i, decide whether the i-th bit of $f(x)$ is zero.

The two classes, however, are not the same. Define $\mathsf{FP}^{\mathsf{NP}}[f(n)]$ to be the class of functions computable in polynomial time given access to call *an oracle* for an NP-complete problem at most $f(n)$ times. For a formal definition of an *Oracle Turing machine*, please refer to [AB09, Definition 3.4; AB07, Definition 3.7]. Define $\mathsf{P}^{\mathsf{NP}}[f(n)]$ analogously. In [Kre88, Theorem 4.1], it is proved that if $\mathsf{FP}^{\mathsf{NP}}[\mathcal{O}(\log n)] = \mathsf{FP}^{\mathsf{NP}}[n^{\mathcal{O}(1)}]$, then also $\mathsf{P} = \mathsf{NP}$. In turn, as noted in [Har93, discussion after Theorem 8], the corresponding result for $\mathsf{P}^{\mathsf{NP}}[\mathcal{O}(\log n)]$ versus $\mathsf{P}^{\mathsf{NP}}[n^{\mathcal{O}(1)}]$ is not known, and indeed fails relative to some oracles.

For a good discussion specifically on FP-completeness, which is relatively hard to find, there is an argument that finding the lexicographically first maximal clique in an undirected graph is NC^i-complete for FP in [Coo85, Proposition 6.1].

Remark 6.7 (Fine-grained reductions). Ideally, we would measure the time and space complexity of algorithms more precisely than up to a polynomial. In practice, we are the most interested in complexity bounds such as $\mathsf{DTIME}(\mathcal{O}(n^2))$. However, the field of studying syntactical characterizations of such functions is still mostly undiscovered. In [Jon93], some problems complete for nondeterministic linear time are studied. In [GS89], the notion of QL reductions is introduced to study complete problems for "nearly linear time", i.e. $\mathsf{DTIME}(n(\log n)^{\mathcal{O}(1)})$.

6.2.1. FNP

The definition of FNP is tricky to get right. A very good discussion of the awkwardness of the definitions is present in [htt17b]. For extensive discussion on the different definitions, see [htt20], [htt17a]. In Papadimitriou's book, it's defined in yet another way, as a class of function problems for NP, not in terms of a specific computational model.

Definition 6.2.3 (The class FNP)**.** A binary relation $P(x, y)$ is in FNP if there exist:

(i) a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that if, for a given x, there exists a solution y such that $P(x, y)$, then there also exists a "short" solution y' such that $P(x, y')$ and $|y'| \leqslant p(|x|)$;

(ii) a deterministic polynomial-time Turing machine M (a *verifier*), such that for every input pair (x, y),

$$P(x, y) \iff M(x, y) = 1.$$

Remark 6.8. This definition is in style of [Ric07, 28.10 and Theorem 28.9], where also the other, nondeterministic Turing machines-based definition is listed.

The other definition might come off as more intuitive: that a relation P is in FNP iff there is a nondeterministic polynomial-time algorithm that, given an arbitrary input x, can find some y such that $P(x, y)$ or signal that it doesn't exist [BD19]. However, as such nondeterministic Turing machines don't seem to be physically realisable, we don't want to introduce that computational model in this work.

6.2.2. NP vs FNP and the total search problems

Definition 6.2.4 (TFNP)**.** A binary relation $P(x, y)$ is in TFNP (total FNP) iff it is in FNP and for every x there exists at least one y such that $P(x, y)$.

An interesting example of a problem in TFNP is PIGEON defined below, for which we mathematically know that the answer exists, but finding it is not trivial.

Definition 6.2.5 (PIGEON)**.** Given a binary string encoding a Boolean circuit $C : \{0,1\}^n \to \{0,1\}^n$, return either an input x such that $C(x) = 0^n$, or two distinct inputs $x \neq y$ such that $C(x) = C(y)$.

Remark 6.9. This problem will be of our interest in Theorem 8.11, where we will discuss mathematical theories so weak that the pigeonhole principle is not their theorem. The intuition behind it is that the computational content of these theories is not strong enough to perform an exhaustive linear search of the whole domain.

The class PPP, a subclass of TFNP problems for which the solution is guaranteed to exist by the pigeonhole principle, is conjectured to not be equal to FP. If PPP = FP, then one-way permutations do not exist [Pap94, Proposition 3], which would have tremendous implications for cryptography.

The class TFNP is discussed in yet more detail in [Mor05, Section 1.1].

6.2.3. Language for FL

Theorem 6.10 ([Coo85, Proposition 4.1])**.** *We obtain precisely the class* FL *from the closure of* L *under* NC^1 *circuit reductions, symbolically:* $\mathsf{FL} = \mathsf{L}^*$

Remark 6.11. Originally, this theorem is proved with NC^1-reducibility meaning reducibility by U_{E^*}-uniform NC^1 circuits. We don't introduce these notions in this work (except for a brief discussion of U_{E^*}-uniformity in Section A.2)

An overview of problems complete for L is present in [CM87].

6.2.4. Language for FP

We can derive a similar result (but in a slightly different setting) to Theorem 6.10 for the class FP. For now we are just mentioning it. We will discuss this in more detail in Theorem 8.9.

The notion of P-completeness is defined formally e.g. in [AB07, Definition 6.25; AB09, Definition 6.28]. A very detailed description of one problem complete for P under L-reductions is in [Koz06].

6.2.5. Not-a-Language for FNP

There is a relatively agreed-upon notion of reductions between FNP problems:

Definition 6.2.6 ([Göb11, Slide 4; Gol21, Slide 5] Polynomial-time reductions for FNP)**.** Let `HardProblem`, `NewProblem` be search problems in FNP. We say that `HardProblem` (many-one) reduces to `NewProblem` if there exist f, g in FP such that:

(i) for an arbitrary input x for `HardProblem`, if $\texttt{NewProblem}(f(x), y)$, then also $\texttt{HardProblem}(x, g(y))$;

(ii) if, for a given input x, there exists $y_{\texttt{Hard}}$ such that $\texttt{HardProblem}(x, y_{\texttt{Hard}})$, then there also exists a $y_{\texttt{New}}$ such that $\texttt{NewProblem}(f(x), y_{\texttt{New}})$.

There are plenty of NP-complete problems described in the literature. However, it is unlikely that we will obtain FNP from taking the closure of an NP-complete problem under the just defined notion of reductions. The class $\mathsf{FP}^{\mathsf{NP}}$ is well-studied and nothing suggests it to be equal to FNP.

6.2.6. Semantic and syntactic complexity classes

Some of the popular complexity classes are defined in such a way that it is notoriously difficult to exhibit natural complete problems for them. A complexity class is called *syntactic* if it admits a natural notion of completeness, that is, if there exists a problem that is complete for the class under the chosen reductions. Classes for which no such complete problems are known are often called *semantic*. For example, in [GP18] the authors introduce a new complexity class PTFNP $\subseteq$ TFNP, prove that it has a complete problem, and therefore refer to PTFNP as a syntactic subclass of TFNP.

An interesting discussion of the absence of complete problems, centred around the class inv-P introduced in Section 5.2, is given in [Daw12]. For the class BPP, related issues are discussed in [htt17c]. Despite BPP being believed to be semantic, a characterization of it (not purely syntactic) was studied in [LT12]. Interestingly, PP has been characterized implicitly (i.e. syntactically) by Ugo Dal Lago: [DKO21].

Remark 6.12 (Bibliography). For probably the first published recognition of the widespread inconsistency between decisional and functional complexity classes in the literature, with examples of inconsistent places see [Lee91, Page 131] (and our bibliographical Remark A.1).

TFNP was first introduced in [MP91].

Digression: Oracle Turing machines and the technique of forcing

Notice that programming in this chapter we essentially try to introduce a new style of programming, where when we want to write a program in P, we write most of the functions in a (simpler) language for L-reductions, and sometimes pass their results to some special function implementing a P-complete algorithm. In the literature, such a "special function" is sometimes called an *oracle*. Please see [AB09, Definition 3.4; AB07, Definition 3.7] for a formal definition of Oracle Turing machines. This would introduce a new programming paradigm, *oracle-oriented* programming, utilizing the notion of *oracle* from the literature. In this thesis we were not able to showcase this paradigm, it will, however, be the ultimate direction of our further works.

Here we only want to hint that a strong technique called *forcing* can be used in a similar manner in both set theory and computational complexity. For information on the technique

of forcing in complexity theory, please see [Gro12] for discussion in context of the Baker-Gill-Solovay theorem[1]; [JTS12] for viewing forcing as a program transformation technique. For general information on the forcing technique, more focused on set theory, please see e.g. [Cho08].

[1]The Baker-Gill-Solovay theorem is studied e.g. in [AB07, Theorem 3.9; AB09, Theorem 3.7].

Chapter 7

Implicit Computational Complexity

Implicit computational complexity (ICC) studies how to guarantee resource bounds without appealing to external machine models. Instead of analysing running time or space after the fact, ICC designs languages and recursion schemes whose syntactic constraints ensure that every definable function belongs to a chosen complexity class. The aim is a foundation for programming languages that "build in" complexity guarantees by construction.

The goal of this chapter is to share a *negative* result about utilizing the developments from ICC to certify complexity *of standard algorithms*. In Subsection 7.2.4, we will introduce a programming language for FL based tightly on a characterization by an (untyped) function algebra due to Neergaard. In Subsection 7.3.1, we discuss a programming language `IntML`, due to Dal Lago and Schöpp, with linear types designed to capture FL and FNL complexity (nondeterministic version of FL, which we don't introduce formally).

Accessible introductions to ICC include the three-part presentation [Mar06a; Mar06b; Mar06c], the talk [Roc19], and the short overview [Dal12].

7.1. The failure of utilizing these ideas for certifying complexity

7.1.1. Problem 1: the well-known algorithms can't be transferred to the languages

Recall that, given a description of a Turing machine, the problem of deciding whether it halts is undecidable. Hence, deciding whether an arbitrary computer program belongs to the complexity class P is also undecidable. This is only a problem, however, if we take *arbitrary* programs as input. If we restrict attention to programs written in a special, limited programming language, we can easily design the language so that it does not admit constructs such as `while`-loops or general recursion, and therefore every program in it necessarily terminates. Moreover, membership in the syntax of such a language can be easy to decide. In this way, the undecidability problem is not resolved, but shifted to the difficulty of programming: given a `C` program implementing an algorithm, it becomes undecidable whether there exists a corresponding program in our restricted language.

This phenomenon shows up as the difference between *intensional* and *extensional* expressive power of characterizations of complexity classes. The equivalence proofs between ICC characterizations and the usual machine-based classes are purely extensional: they show that if a function belongs to, say, FL, then there exists some term in the characterization language that computes the same function, and conversely. For low complexity classes, the equivalence being only extensional becomes very apparent. The characterizations are intricate and require a completely different style of programming. The natural and well-known algorithms *do not*

transfer to the currently known languages characterizing these classes; new algorithms would have to be invented in the ICC formalisms. We will demonstrate an example of how easily we can lose intensional expressive power in Listing 7.1 and Remark 7.19.

For a more authoritative argumentation, please refer to [DM06, Slide 31].

7.1.2. Problem 2: every characterization is a new programming paradigm

As of 2025, the characterizations discovered are not uniform in style. They use completely different techniques and, in effect, would yield different programming languages for each of the complexity classes. There are characterizations that are able to capture multiple classes in a uniform way. In general, however, the field appears to be very fragmented with little overlap between the concepts. This is also pointed out in [DM06].

7.1.3. Scope of our research and history of the field

The modern study of ICC begins with two back-to-back breakthroughs: the introduction of tiered recursion by Leivant in [Lei91] and the introduction of safe recursion by Bellantoni and Cook in [BC92]. We have focused on the safe-recursive approach in this work, and we will not discuss in detail here the developments stemming from Leivant's branch. For that, we refer to [RL11] and to [Sim05] for a discussion of tiering as a recursion technique.

It is important to note that we are specifically interested in *capturing* a complexity class by a programming language. This is a stronger property than just having the guarantee that all functions of a programming language are contained in a complexity class; an interesting work studying the latter is [CS12].

We primarily focused on the characterizations of FL, with a lot of attention also given to the class FP and significantly less to other classes. In [Hof06] a good overview of languages for FL is presented, and in [Sch06a] the history of FL characterizations is traced.

Before the seminal works that founded the field of implicit complexity, many characterizations of complexity classes were already known. All of them suffered from at least one of two problems: either they characterized only a class of relations in a given complexity and not functions, or the characterization was not purely syntactic and required additional proofs. We will refer to the latter as being "explicit" instead of "implicit", as discussed in Subsection 7.2.2. In this explicit manner, uniform decision NC^1 was characterized in [CL90] and uniform decision NC in [All91]. Early explicit function algebras for FL appeared in [LM73] and [Lin74], and a function algebra with explicit bounds for FP was introduced in [Gur83]. For the more historical works, there is a remarkably good literature review in [Blo94, Section 1], despite that paper introducing new, unrelated characterizations.

These explicit characterizations formed the foundation on which modern ICC was built. They already tied complexity classes to restricted programming languages, but parts of the complexity proofs still had to be given manually: that the explicit bounds required by the characterizations really hold. Modern developments are almost always purely syntactic and thus can form a feasible basis for a programming language.

Indeed, in [Jon99], decision L and P were characterized by a fragment of Lisp. The same concept has been extended to account for nondeterminism in [Bon06]. The authors of [KV05] investigated both imperative and functional programming languages whose fragments yield hierarchies containing *decision* L, $\mathsf{LINSPACE}$, P, and PSPACE. Related contributions include [Kri05] and [Oit10]. In [LR15], an interesting original approach using coinduction is used to capture FL. See [DKO22] for implicit characterizations of counting classes such as $\#\mathsf{P}$ (not introduced here).

Please refer to [NW10] for overviews of the different characterizations of FP and FNC, and to [Bon+16] for NC^k.

7.2. Recursion-theoretic approach

In this section we focus on techniques from recursion theory that were successfully utilized in Implicit Computational Complexity.

7.2.1. Origins of recursion theory

While not the primary focus of this work, the field of recursion theory developed concepts that later became foundational for ICC. An important formal system studied there is *primitive recursion*.

Definition 7.2.1 (Primitive recursive functions)**.** PR is the smallest class of functions containing (i)–(iii) and closed under (iv) and (v):

(i) **(constants)** for every $n \in \mathbb{N}$ and $k \geq 0$, the k-ary constant function $c_n^{(k)}(\vec{x}) = n$;

(ii) **(successor)** $S(x) = x + 1$;

(iii) **(projections)** for $k \geq 1$ and $1 \leq i \leq k$, $\pi_i^{(k)}(x_1, \ldots, x_k) = x_i$;

(iv) **(composition)** if $h : \mathbb{N}^m \to \mathbb{N}$ and $g_1, \ldots, g_m : \mathbb{N}^k \to \mathbb{N}$ are in PR, then $f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x}))$ is in PR;

(v) **(primitive recursion)** if $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ are in PR, then the unique $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is in PR with:

$$f(0, \vec{x}) = g(\vec{x}), \qquad f(S(y), \vec{x}) = h(y,\, f(y, \vec{x}),\, \vec{x}).$$

Example 7.1 (Addition)**.** Define $\mathrm{Add} : \mathbb{N}^2 \to \mathbb{N}$ by primitive recursion:

$$\mathrm{Add}(0, x) = x, \qquad \mathrm{Add}(S(y), x) = S\big(\mathrm{Add}(y, x)\big).$$

Definition 7.2.2 (LOOP language)**.** Let $\mathrm{Var} = \{x_0, x_1, x_2, \ldots\}$. LOOP programs are generated by the grammar

$$P ::= x_i := 0 \;\mid\; x_i := x_i + 1 \;\mid\; P\,;\, P \;\mid\; \texttt{LOOP}\ x_i\ \texttt{DO}\ P\ \texttt{END},$$

where $x_i \in \mathrm{Var}$.
We assume standard semantics, with a remark that `LOOP` x_i `DO` P `END` repeats P exactly as many times as the value stored in x_i *at loop entry* (changes to x_i inside P do not change the iteration count).

Theorem 7.2 ([MR67] LOOP captures precisely PR)**.** *The functions computable by* LOOP *programs are precisely the primitive recursive functions.*

This simple connection actually satisfies our criteria of a "programming language capturing a complexity class", as the LOOP language captures exactly PR[1], the class of primitive recursive functions. Moreover, we can even stratify the primitive recursive functions into a hierarchy like in [Grz53].

Remark 7.3 (Bibliography)*.* Historically, the origins of primitive recursion can be traced back to [Gra61] and [Ded88], but the class was probably first considered as the primary object of study in [Sko23]. For the details of the historical origins, consult [Ada11].

[1] As recognized at `https://complexityzoo.net/Complexity_Zoo:P`.

7.2.2. Explicit characterizations

Some characterizations discussed in the literature contain so-called *explicit* conditions — e.g. they explicitly require a function to not grow faster than polynomially. Such conditions cannot be checked syntactically and must be enforced by an additional proof outside of the algebra. We call these characterizations explicit, as opposed to implicit. Despite not being convenient to use for our purpose, these concepts have been very important for the field and we will discuss one example in this subsection. For a good overview of implicit versus explicit characterizations, refer to [Aub15].

A famous example of an explicit characterization is Cobham's algebra for polynomial-time functions, using the recursion scheme defined below. For appending a binary digit to y, we will use the notation $S_0(y) = 2 \cdot y$ and $S_1(y) = 2 \cdot y + 1$; $S_0(0) = 0$. We will denote by $|x|$ the length of binary representation of x; in particular, $|0| = 1, |1| = 1, |2| = 2$.

Definition 7.2.3 ([Odi99, Definition VIII.2.14 (Page 174)])**.** A function f is defined from functions g, h_0, h_1, and s by *bounded primitive recursion on binary notation* if, for every $\vec{x}$ and $y \in \mathbb{N}$,

$$f(\vec{x}, 0) = g(\vec{x}); \tag{7.1}$$
$$f(\vec{x}, S_0(y)) = h_0(\vec{x}, y, f(\vec{x}, y)) \quad \text{if } y > 0; \tag{7.2}$$
$$f(\vec{x}, S_1(y)) = h_1(\vec{x}, y, f(\vec{x}, y)); \tag{7.3}$$
$$f(\vec{x}, y) \leq s(\vec{x}, y). \tag{7.4}$$

Remark 7.4. The recursion parameter y is written in binary, so the definition of $f(\vec{x}, y)$ unfolds only $\mathcal{O}(|y|)$ many steps. Moreover, the side condition $f(\vec{x}, y) \leq s(\vec{x}, y)$ implies that every value of $f(\vec{x}, y)$ is at most $s(\vec{x}, y)$. Hence, if the defining functions g, h_0, h_1 and the bounding function s are polynomially bounded (in the lengths of their arguments in binary), then f will also be polynomially bounded.

In this definition, the bound (7.4) is *explicit*: when one writes a function in this style, there is no obvious way to check mechanically that $f(\vec{x}, y) \leq s(\vec{x}, y)$ holds, other than by supplying a separate mathematical proof.

Definition 7.2.4 (Cobham's algebra for FP)**.** The class Cob is the smallest class of functions containing (i)–(iv) and closed under composition and bounded primitive recursion on binary notation:

(i) for every $k \geq 0$, the k-ary constant function $0(\vec{x}) = 0$;
(ii) the binary successor functions $S_0(x) = 2x$ and $S_1(x) = 2x + 1$;
(iii) for $k \geq 1$ and $1 \leq i \leq k$, projections $\pi_i^{(k)}(x_1, \ldots, x_k) = x_i$;
(iv) the weak exponential $(x, y) \mapsto x^{|y|}$ denoted $x \# y$; sometimes called *the smash function.*

Remark 7.5. This algebra was originally defined for decimal digits. Note that all of our initial functions are polynomially bounded in the lengths of their arguments. For the smash function we have, for $x, y \geq 1$,

$$x < 2^{|x|}, \qquad x^{|y|} < 2^{|x| \cdot |y|}, \qquad |x^{|y|}| \leq |x| \cdot |y| + 1,$$

hence

$$|x \# y| = |x^{|y|}| \leq |x| \cdot |y| + 1.$$

Composition preserves polynomial boundedness, and bounded recursion on binary notation also preserves it (recall Remark 7.4), so every function in Cob is polynomially bounded.

Theorem 7.6 ([Odi99, Proposition VIII.2.15 (Page 175)])**.** *The class* Cob *contains precisely the functions from* FP*.*

Remark 7.7. Cobham's characterization underlies the arithmetical theory PV, which is also designed to capture polynomial time reasoning in the style discussed in Chapter 8. We will not introduce PV in this work, but we will mention it once again in Subsection 8.3.1 to discuss another concept for a programming language. Please see [BC92, End of section 6] for a brief discussion.

Remark 7.8 (Bibliography). This algebra from Definition 7.2.4 was originally published in [Cob64] and never proved by the author to actually capture FP. In [Tou22, Remark 15.3.22] there is a discussion of several proofs of this theorem from the literature, each based on different, and in general non-equivalent, definitions. The proof we referenced in the header of Theorem 7.6 is due to Odifreddi.

7.2.3. Characterization of FP with safe-recursion

Bellantoni and Cook introduced a function algebra BC whose key innovation is the separation of arguments into *normal* inputs (controlling recursion depth) and *safe* inputs (being passed around without influencing that depth). We write $f(\vec{x};\vec{a})$, with normal inputs $\vec{x}$ to the left of the semicolon and safe inputs $\vec{a}$ to the right.

Definition 7.2.5 (Bellantoni and Cook's algebra for FP)**.** We will use the notation $a0$ for the binary representation of $2 \cdot a$. Similarly, we will use $a1$ for $2 \cdot a + 1$, and the more general ai when i is known from the context to be equal to either 0 or 1.

The class BC is the smallest class of functions on non-negative integers that contains (i)–(v) and is closed under (vi) and (vii):

(i) **(constant)** $0(;) = 0$;

(ii) **(projection)** for $m, n \geq 0$ and $1 \leq j \leq m + n$,

$$\pi_j(x_1, \ldots, x_m;\, a_1, \ldots, a_n) = \begin{cases} x_j & \text{if } j \leq m, \\ a_{j-m} & \text{otherwise;} \end{cases}$$

(iii) **(successors)** $s_i(;a) = 2a + i$ for $i \in \{0, 1\}$;

(iv) **(predecessor)** $p(;0) = 0$ and $p(;ai) = a$;

(v) **(conditional)**

$$C(;a,b,c) = \begin{cases} b & \text{if } a \bmod 2 = 0, \\ c & \text{otherwise;} \end{cases}$$

(vi) **(predicative recursion on notation (*safe* recursion))** if $g, h_0, h_1 \in \mathsf{BC}$, then also $f \in \mathsf{BC}$ with

$$\begin{aligned} f(0,\vec{x};\vec{a}) &= g(\vec{x};\vec{a}), \\ f(yi,\vec{x};\vec{a}) &= h_i(y,\vec{x};\vec{a}, f(y,\vec{x};\vec{a})) \qquad \text{for } i \in \{0,1\} \end{aligned}$$

(vii) **(safe composition)** if $h, \vec{r}, \vec{t} \in \mathsf{BC}$ with each component of $\vec{r}$ taking only normal arguments and each component of $\vec{t}$ taking both normal and safe arguments, then also $f \in \mathsf{BC}$ with

$$f(\vec{x};\vec{a}) = h\big(\vec{r}(\vec{x};\,);\ \vec{t}(\vec{x};\vec{a})\big).$$

Theorem 7.9 ([BC92, Theorem 3.3, 4.2])**.** *Let $f(\vec{x})$ be a function in* FP*. Then $f(\vec{x};) \in \mathsf{BC}$. Let $f(\vec{x};\vec{y})$ be a function in* BC*. Then $f(\vec{x},\vec{y}) \in \mathsf{FP}$.*

Remark 7.10 (Intuition). The key idea of safe recursion is that safe arguments can never flow back into normal positions. In safe composition, the normal arguments of h are obtained only from functions that themselves take only normal inputs. In the recursion scheme, the recursive value $f(y, \vec{x}; \vec{a})$ is passed to h_i as a safe argument. As a result, the depth of recursion can depend only on the normal inputs, and not on values computed during the recursion.

Remark 7.11. The theorem is formulated in terms of computation on non-negative integers, but the proof transfers to the case of general binary strings (e.g. being able to start with a zero) [BC92].

Remark 7.12 (Formalization of polynomial time functions). Interestingly, in [HN11] the authors claim formalizing a proof that the Bellantoni-Cook algebra formulated on binary strings (recall Remark 7.11) captures precisely FP.

This sparked our interest, as it could suggest that the authors have formalized the notion of an FP function in a proof assistant. In particular, one could hope for a compiler translating BC programs into Turing machines running in FP. This is not the case, however, as they have only formalized the proof that the class BC is the same as the class Cob.[2]

Remark 7.13. In [BC92] it is also shown how to readily use their safe recursive algebra to characterize functions from FL with "small output", but this characterization relied on using unary representation of natural numbers on input, which is more of a hack than a true characterization of this class.

7.2.4. Characterization of FL with affine safe recursion

Møller-Neergaard refined the safe recursion discipline to capture FL by strengthening the treatment of safe data: each safe value may be *used* at most once. An analogy from quantum computing is that after we "measure" a value by, for example, testing one of its bits in a conditional, it disappears (goes out of scope). As in the Bellantoni-Cook setting, arguments are split into normal and safe ones, and recursion is permitted only on the normal arguments. In addition, the composition and recursion schemes are designed so that safe arguments are never duplicated, and recursion has a course-of-value flavour: successive recursive calls are allowed to "jump back" along the recursion chain and do not need to visit every intermediate value. These restrictions together yield a function algebra that characterizes FL. As the precise definition is quite technical and will not be used later, we refer the interested reader to Definition B.0.1 and Theorem B.1 in Appendix B.

This algebra was very important for us, as its description in [Nee04] explicitly names it as a programming language. It seemed very promising indeed to be implemented on a computer as a simple, useful programming language.

However, in practice we found it very difficult to program in this algebra. Neergaard's system can define every function in FL, but not every *algorithmic technique* commonly used to implement FL Turing machines. In other words, it matches FL extensionally, but its intensional expressive power (cf. Subsection 7.1.1) is quite limited. It seemed unlikely that we could add support for structures such as pairs and lists to this programming language. In an unpublished technical report by the author, accessible at [Møl04], it was discussed that the type of *pairs* of numbers seems not to be implementable in this language. The property of the whole data structure *disappearing* after we check one bit of it seems to render implementing typical data structures very difficult.

[2]Link to the source code: `https://github.com/davidnowak/bellantonicook`.

Neergaard's original code from 2004 The author's original code of the interpreter referenced in the paper is not publicly accessible. We have managed to obtain the original Moscow ML code from 2004 from the author on a permissive license, port it to a modern version of SML/NJ and release it with the author's permission.[3]

Remark 7.14 (Bibliography). The technical report [Møl04] has never been published and seems to be a preliminary version of the author's publication from the same year. Despite that, it contains crucial insight on the limits of this characterization. There are papers closely related to Neergaard's publication and necessary to also be studied while exploring Neergaard's work. Please also see [MM03]. There is also [MO00b], but it seems inaccessible online. Probably the contents of that work are similar to [MO04]

Remark 7.15 (Origins of this thesis). For insight into the timeline of this thesis, it is worth noting that we have first discovered this paper ourselves around February 2023.

7.3. Linear types

The type systems of most popular programming languages are fairly weak: they do not provide direct support for verifying high-level correctness or complexity properties. Functional programming languages with strong type systems exist, allowing us to verify quite useful correctness properties — one example is the language Haskell. Going yet stronger, Lean and Rocq are at the same time programming languages and proof assistants, i.e. their type systems allow us to prove even very abstract mathematical properties of the functions they define. However, none of the mainstream languages allows us to enforce useful computational complexity properties of programs.

One of the properties that is notoriously difficult to enforce is preventing data from being copied[4] or discarded without being used.[5] We will call variables that are *non-copyable affine*, and reserve the term *linear* for variables that must be used *exactly once*, i.e. they may be neither duplicated nor discarded. The affine requirement can be enforced by making the variable go out of scope just after it was used. Affine variables are important in logspace computations, where copying large enough data is simply not implementable. Linear variables, on the other hand, are reminiscent of the requirement that every class construction must be matched by a corresponding destruction operation on every execution path. We will see an example of affinity enforced by linear types in Haskell in Listing 7.1.

These ideas have been studied for decades in proof theory. There, a proof system may or may not allow the so-called *structural* rules: weakening, contraction and exchange. Proof systems that lack one of these rules are studied in the field of linear logic, introduced in [Gir87] with a clear intent to be used in computer science. The semantics of linear logic has a natural interpretation in terms of *resources*: a proposition may be used exactly once, or at most once, rather than freely duplicated and discarded. The concepts from linear logic have been carried over almost directly to *linear type systems*, which, in (typically functional) programming languages, can control the ability to clone and discard data.

As it turns out, this level of control is also enough to limit the computational complexity of definable functions. The class FL has been captured by a variant of affine logic in [Sch07]; it was also studied in [Sch06b], [LS10] and [Maz15]. The class FP was characterized in [LM93]. Most importantly for us, the programming language `IntML` was introduced in [DS16], which we will discuss in Subsection 7.3.1.

[3]The code is available at: `https://github.com/ruplet/neergaard-logspace-characterization`.

[4]We can delete the copy constructor in C++, but this is of course bypassable.

[5]A linter can detect unused variables, but this does not enforce a guarantee.

Remark 7.16. The formal bridge between linear logics and linear type systems is the Curry-Howard correspondence, also known as "propositions as types" or "proofs as programs". The connections between logic, type systems and complexity theory are already well explored in the literature, e.g. [Ben15]. We will not repeat these definitions here. A good introduction to the Curry-Howard correspondence is the book by Sørensen and Urzyczyn [SU06].

Remark 7.17. Controlling the computational complexity of programs through the complexity of their specification was already discussed in Chapter 5, where the control went through model theory. Here, the restrictions are entirely in the world of proof theory, which is more directly connected to computation than model theory.

Remark 7.18 (Support for linear types in mainstream programming languages). Some mainstream programming languages offer some support for linear types. Haskell (GHC $\geq$ 9.0.1) supports a limited version of linear types. In Rust, affine reasoning can be expressed through the ownership and borrowing mechanism. Going less mainstream, Idris 2, $F^\star$ and $Q^\star$ (a quantum programming language) also support different variants of linear types.

Remark 7.19. In Listing 7.1, the type system prevents us from using a linear argument more than once. The function `collatzBad2` fails to type-check precisely because the argument `x` would be consumed by `evenBits x` and then used again in the branches. This effect makes most of the standard algorithms not transferrable to this formalism.

7.3.1. IntML

In 2013, Dal Lago and Schöpp introduced `IntML`, a functional language with a linear type system that characterizes FL [DS16]. An implementation of `IntML` is available on GitHub.[6] To the best of our knowledge, it remains the only language within the linear-logic branch of ICC that has both a working implementation and some potential for (academic) practical use.

From the point of view of this thesis, however, linear-logic-based approaches — including `IntML` — run into the *same* issue of intensional vs. extensional expressive power discussed in Subsection 7.1.1. These systems characterize classes such as FL and FP *extensionally*. The characterizations capture the right functions, but not the usual *algorithmic techniques* used to implement them. `IntML` looks like a very good starting point for a practical programming language. However, it would still be very hard to use it to certify the complexity of standard algorithms.

In this thesis, we will not pursue this line further as a practical basis for certifying the complexity.

[6] `https://github.com/uelis/IntML`. Following private communication with the authors, a permissive license was added to the repository, as it was not included originally.

```haskell
{-# LANGUAGE LinearTypes #-} -- compilation: `ghc Linear.hs`, ghc >= 9.0.1
module Linear where
import Prelude

-- Define own bitstring type for ints, as operations from Prelude
-- on Int are *not* linear and will not typecheck.
data Bit  = Zero | One
data Bits = Nil | B0 Bits | B1 Bits  -- Prepend 0 or 1 as the LSB.
const_5 :: Bits = B1 (B0 (B1 Nil))
const_6 :: Bits = B0 (B1 (B1 Nil))

burn :: Bits %1-> ()
burn Nil      = ()
burn (B0 xs)  = burn xs
burn (B1 xs)  = burn xs

evenBits :: Bits %1-> Prelude.Bool
evenBits Nil      = Prelude.True
evenBits (B0 xs)  = case burn xs of () -> Prelude.True
evenBits (B1 xs)  = case burn xs of () -> Prelude.False

half :: Bits %1-> Bits
half Nil      = Nil
half (B0 xs)  = xs
half (B1 xs)  = xs

plus2x1 :: Bits %1-> Bits
plus2x1 x = B1 x

branchConst :: Bits %1-> Bits
branchConst x =
  if evenBits x
    then half const_5
    else plus2x1 const_6

-- collatzBad2 :: Bits %1-> Bits
-- collatzBad2 x =
--   if evenBits x
--     then half x       -- ERROR: x already consumed by 'evenBits x'
--     else plus2x1 x    -- ERROR: x already consumed by 'evenBits x'
```

Listing 7.1: Example of linear types in Haskell

Chapter 8

Bounded arithmetic

In mathematics we typically assume some (pretty strong) foundational axioms we rely on to prove theorems. If we choose set theory as the foundation (as we usually do), a debatable concept is whether we should use the axiom of choice or not. More popularly in computer science, we often want to be explicit about using Kőnig's lemma[1] and Ramsey's theorem. If we think of the concept we introduced in Chapter 6 (i.e. writing a program in a language in which calls to the computation-heavy oracle are explicit), we would often ask ourselves the same question — can we write the program without relying on that function, i.e. write the program in a lower complexity? It turns out the similarity is not a coincidence, and to explore this connection further we need to study *bounded arithmetic*, in particular the theories $\mathrm{I}\Delta_0$ and V^0.

In Section 8.4 we describe the main technical contribution of this thesis: (i) a Lean 4 formalization of the theory $I\Delta_0$, defined below, which is compatible with existing code in Mathlib; and (ii) initial work on code extraction from proofs in two-sorted arithmetical theories that capture complexity classes, implemented by means of metaprogramming in Lean 4.

8.1. Single-sorted logic and $\mathrm{I}\Delta_0$

Definition 8.1.1 ([CN10, Definition III.1.1])**.** Recall that a logical formula may have free variables (Definition 2.1.5). By the *universal closure* of a formula $\varphi(x_1, \ldots, x_n)$ with free variables $x_1, \ldots, x_n$ we will denote the sentence $\forall x_1. \ldots \forall x_n. \varphi(x_1, \ldots, x_n)$. A *theory* over a vocabulary $\mathcal{L}$ is a set $\mathcal{T}$ of $\mathcal{L}$-formulas that is closed under logical consequence and under universal closure.

Note that we have not defined "logical consequence"; it depends on the choice of a particular *proof system* (also called *proof calculus*). We will not define proof systems in detail here. All results in this chapter assume standard Gentzen-style proof calculus for classical logic, LK [CN10, Section II.2.3] for single-sorted logic and LK^2 [CN10, Section IV.4] for two-sorted logic.

Definition 8.1.2 ([CN10, Definition II.2.3])**.** The vocabulary of arithmetic is

$$\mathcal{L}_A = \langle 0, 1, +, \cdot\ ;\ =, \leq \rangle.$$

Here $0, 1$ are constant symbols; $+$ and $\cdot$ are binary function symbols; and $=$ and $\leq$ are binary predicate symbols. We will implicitly assume the standard interpretation of these symbols

[1] Note that Kőnig's lemma is a form of countable choice from finite sets.

as the appropriate functions on natural numbers whenever talking about the semantics of $\mathcal{L}_A$-formulas.

Definition 8.1.3 ([CN10, Figure 1] Axioms 1-BASIC of Peano arithmetic)**.**

B1. $x + 1 \neq 0$	B5. $x \cdot 0 = 0$
B2. $x + 1 = y + 1 \to x = y$	B6. $x \cdot (y + 1) = (x \cdot y) + x$
B3. $x + 0 = x$	B7. $(x \leq y \wedge y \leq x) \to x = y$
B4. $x + (y + 1) = (x + y) + 1$	B8. $x \leq x + y$
C. $0 + 1 = 1$	

Remark 8.1. Axiom C is provable with induction (defined below) from B1–B8. It is included in the axioms, because then all true quantifier-free sentences over $\mathcal{L}_A$ are provable from 1-BASIC [CN10, Lemma III.1.3].

Definition 8.1.4 ([CN10, Definition III.1.4] Induction Scheme)**.** Let Φ be a set of formulas. The Φ-*IND axioms* are all formulas of the form

$$(\varphi(0) \ \wedge \ \forall x.\, (\varphi(x) \to \varphi(x+1))) \ \to \ \forall z.\, \varphi(z), \tag{8.1}$$

where φ ranges over formulas in Φ. Note that $\varphi(x)$ may have free variables other than x.

Definition 8.1.5 ([CN10, Definition III.1.5] Peano Arithmetic)**.** The theory PA has as axioms $\mathrm{B1}, \ldots, \mathrm{B8}$, together with the Φ-IND axioms, where Φ is the set of all $\mathcal{L}_A$-formulas.

Peano Arithmetic is a powerful theory capable of formalizing the major theorems of number theory. We define subsystems of PA by restricting the induction axioms to certain sets of formulas.

Recall the notion of a *bounded* two-sorted formula (Notation 2). We define bounded single-sorted formulas analogously to be the formulas in which every quantifier is of the form $\forall x \leq t.\, \varphi(x)$ or $\exists x \leq t.\, \varphi(x)$, where x does not occur in t.

Definition 8.1.6 ([CN10, Definition III.1.7] IOPEN, $\mathrm{I}\Delta_0$, $\mathrm{I}\Sigma_1$)**.** Let OPEN be the set of *open* (i.e. quantifier-free) formulas, let Δ_0 be the set of bounded formulas, and let Σ_1 be the set of formulas of the form $\exists \vec{x}.\, \varphi$, where φ is bounded and $\vec{x}$ is a (possibly empty) tuple of variables.

The theories IOPEN, $\mathrm{I}\Delta_0$, and $\mathrm{I}\Sigma_1$ are the subsystems of PA obtained by restricting the induction scheme so that Φ is OPEN, Δ_0, and Σ_1, respectively.

Lemma 8.2 ([CN10, Example III.1.8])**.** *The following formulas (and their universal closures) are theorems of* IOPEN*:*

O1. $(x + y) + z = x + (y + z)$	*(Associativity of +)*
O2. $x + y = y + x$	*(Commutativity of +)*
O3. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	*(Distributive law)*
O4. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$	*(Associativity of ·)*
O5. $x \cdot y = y \cdot x$	*(Commutativity of ·)*
O6. $x + z = y + z \to x = y$	*(Cancellation for +)*
O7. $0 \leq x$	
O8. $x \leq 0 \to x = 0$	
O9. $x \leq x$	
O10. $x \neq x + 1$	

Lemma 8.3 ([CN10, Example III.1.9])**.** *The following formulas (and their universal closures) are theorems of* $\mathrm{I}\Delta_0$*:*

D1. $x \neq 0 \to \exists y \leq x.\, (x = y + 1)$	*(Predecessor)*
D2. $\exists z.\, (x + z = y \ \vee \ y + z = x)$	
D3. $x \leq y \leftrightarrow \exists z.\, (x + z = y)$	
D4. $(x \leq y \wedge y \leq z) \to x \leq z$	*(Transitivity)*
D5. $x \leq y \vee y \leq x$	*(Total order)*
D6. $x \leq y \leftrightarrow x + z \leq y + z$	
D7. $x \leq y \to x \cdot z \leq y \cdot z$	
D8. $x \leq y + 1 \leftrightarrow (x \leq y \vee x = y + 1)$	*(Discreteness 1)*
D9. $x < y \leftrightarrow x + 1 \leq y$	*(Discreteness 2)*
D10. $x \cdot z = y \cdot z \wedge z \neq 0 \to x = y$	*(Cancellation for $\cdot$)*

Using the above lemmas as building blocks, we can prove quite a few nontrivial theorems. We will now introduce the core notion of arithmetic — what does it mean to *define* a function *in a theory.*

Definition 8.1.7 ([CN10, Definition III.3.2] Predicates and Functions definable in a Theory)**.** Let $\mathcal{T}$ be a theory with vocabulary $\mathcal{L}$, and let Φ be a set of $\mathcal{L}$-formulas.

(i) a predicate symbol $P(x) \notin \mathcal{L}$ is Φ-*definable in* $\mathcal{T}$ if there exists an $\mathcal{L}$-formula $\varphi(x) \in \Phi$ such that

$$P(x) \ \leftrightarrow \ \varphi(x). \tag{8.2}$$

(ii) a function symbol $f(x) \notin \mathcal{L}$ is Φ-*definable in* $\mathcal{T}$ if there exists a formula $\varphi(x, y) \in \Phi$ such that

$$\mathcal{T} \vdash \forall x.\, \exists! y.\, \varphi(x, y), \tag{8.3}$$

and moreover

$$y = f(x) \ \leftrightarrow \ \varphi(x, y). \tag{8.4}$$

We call (8.2) a *defining axiom* for $P(x)$ and (8.4) a *defining axiom* for $f(x)$. A symbol is *definable in* $\mathcal{T}$ if it is Φ-definable in $\mathcal{T}$ for some Φ.

Definition 8.1.8. We will say that a function is *provably total* in $\mathcal{T}$ iff it is Σ_1-definable in $\mathcal{T}$.

In [CN10, Section III.3] it is argued that: the functions $\lfloor x/y \rfloor$, $\lfloor \sqrt{x} \rfloor$, $\max(0, x - y)$, $x \bmod y$ are definable in $\mathrm{I}\Delta_0$; relation $x \mid y$ is definable in $\mathrm{I}\Delta_0$, and, interestingly, the relation $\exp(x, y)$ where $\exp(x, y)$ iff $y = 2^x$, is also definable in $\mathrm{I}\Delta_0$. We don't introduce the specific logical formula defining the relation $\exp(x, y)$, as it is complicated and discussed in [CN10, Section III.3]. For a different point of view on these problems, for example in [Jum95] it is shown that Euler's φ function is provably total in $\mathrm{I}\Delta_0$. However, the limits of expressive power of $\mathrm{I}\Delta_0$ are low.

Theorem 8.4 ([CN10, Section III.2])**.**

$$\mathrm{I}\Delta_0 \nvdash \forall x\, \exists y.\, \exp(x, y).$$

Note that PA *easily proves* $\forall x.\, \exists y.\, \exp(x, y)$.

It is interesting to study the theory $\mathrm{I}\Delta_0 + \exp$ of $\mathrm{I}\Delta_0$ axioms with an additional axiom stating that the exponential function is definable. As it turns out, this theory enables us to reason about syntactic constructs such as coding of sets and sequences or context-free grammar parsing [HP93, Chapter V, Section 3][2].

It turns out that a function is Σ_1-definable in $\mathrm{I}\Delta_0$ iff it is in FLTH, the functional version of linear-time hierarchy [CN10, Theorem III.4.8]; for the definition of LTH, refer to [CN10, Section III.4.1] — as this complexity class is far from what we call "feasible" in this work, we don't introduce the details here. Instead, we will now introduce a theory with a good computational complexity characterization.

8.2. Two-sorted logic and V^0

Definition 8.2.1 (Axioms of 2-BASIC)**.**

B1. $x + 1 \neq 0$	B7. $(x \leq y \wedge y \leq x) \rightarrow x = y$
B2. $x + 1 = y + 1 \rightarrow x = y$	B8. $x \leq x + y$
B3. $x + 0 = x$	B9. $0 \leq x$
B4. $x + (y + 1) = (x + y) + 1$	B10. $x \leq y \vee y \leq x$
B5. $x \cdot 0 = 0$	B11. $x \leq y \leftrightarrow x < y + 1$
B6. $x \cdot (y + 1) = (x \cdot y) + x$	B12. $x \neq 0 \rightarrow \exists y \leq x.\, (y + 1 = x)$
L1. $X(y) \rightarrow y < \lvert X\rvert$	L2. $y + 1 = \lvert X\rvert \rightarrow X(y)$
SE. $\big(\lvert X\rvert = \lvert Y\rvert \wedge \forall i < \lvert X\rvert.\, (X(i) \leftrightarrow Y(i))\big) \rightarrow X = Y$	

Definition 8.2.2 ([CN10, Definition V.1.2] Comprehension Axiom)**.** Let Φ be a set of formulas. The *comprehension axiom scheme for* Φ, denoted Φ-COMP, consists of all formulas of the form

$$\exists X \leq y.\, \forall z < y.\, \big(X(z) \leftrightarrow \varphi(z)\big), \tag{8.5}$$

where $\varphi(z) \in \Phi$ and X does not occur free in $\varphi(z)$. In (8.5), the formula $\varphi(z)$ may have free variables of both sorts in addition to z. We are mainly interested in the cases where Φ is one of the classes Σ_i^B.

Definition 8.2.3 (V^i)**.** For $i \geq 0$, the theory V^i has vocabulary $\mathcal{L}_A^2$ and is axiomatized by 2-BASIC together with Σ_i^B-COMP.

Note that there are no explicit induction axioms for V^i.

Theorem 8.5 ([CN10, Corollary V.1.8])**.** *Induction for Σ_i^B formulas is provable in V^i. In particular, induction for Δ_0 formulas is provable in the theory V^0.*

Note that this implies that any theorem φ provable in $\mathrm{I}\Delta_0$ is also provable in V^0.

Theorem 8.6 ([CN10, Theorem V.1.9])**.** *For every formula φ in the vocabulary $\mathcal{L}_A$ of single-sorted arithmetic, if $\mathsf{V}^0 \vdash \varphi$, then also $\mathrm{I}\Delta_0 \vdash \varphi$. In other words, V^0 is a* conservative extension *of $\mathrm{I}\Delta_0$.*

Remark 8.7 ([CN10, Section IV.3] Two-sorted complexity classes)*.* When operating in two-sorted logic, we need to redefine what does it mean for a relation to be in a complexity class. We will think of numerical arguments x_i of a relation $R(\vec{x}, \vec{X})$ to be passed to the deciding Turing machine in unary representation. The string arguments X_i representing finite sets

[2]Note that they use the name $\mathrm{I}\Sigma_0 + \Omega_1$ instead of $\mathrm{I}\Delta_0 + \exp$ which is the same.

of numbers are passed as follows. For a string argument S define $S(i) = 1$ when $i \in S$, 0 otherwise. Then the representation $\ulcorner S \urcorner$ of S, when the largest member of S is n, is defined as the following concatenation of bits:

$$\ulcorner S \urcorner = S(n)S(n-1)\dots S(1)S(0)$$

If S is empty then $\ulcorner S \urcorner$ is the empty string. Note that $|\ulcorner S \urcorner|$ is the same as our interpretation of S inside of the theory: $|S| = \max(S) + 1$ or 0 if S is empty.

We will write $(x)_1$ to denote unary representation of x, i.e. 1^x, and $(x)_2$ to denote binary representation. The ultimate input to the Turing machine deciding if $R(\vec{x}, \vec{X})$ for $|\vec{x}| = n, |\vec{X}| = N, |X_i| = N_i$ is:

$$(n)_1\ 0 \quad (x_1)_1\ 0\ (x_2)_1\ 0\ \dots\ 0\ (x_n)_1\ 0 \quad (N)_1\ 0 \quad (N_1)_1\ 0\ \ulcorner X_1 \urcorner\ 0\ \dots\ 0\ (N_N)_1\ 0\ \ulcorner X_N \urcorner$$

Note that a purely numerical relation $R(x)$ is in two-sorted polynomial time iff it is computed in time $2^{\mathcal{O}(n)}$ for $n = |(x)_2|$. The notion of polynomial-time complexity for relations with only string arguments $R(\vec{X})$ coincides with our standard intuition.

Definition 8.2.4 ([CN10, Definition V.2.1])**.** A number function f or string function F is *p-bounded* iff there exists a polynomial $p(x, y)$ such that, for all inputs x, Y,

$$f(x, Y) \ \leq\ p(x, |Y|) \qquad \text{or} \qquad |F(x, Y)| \ \leq\ p(x, |Y|),$$

respectively.

Definition 8.2.5 ([CN10, Definition V.2.3] Two-sorted functional complexity classes)**.** Let C be a two-sorted complexity class of relations. The corresponding *function class* FC consists of:

(i) all p-bounded number functions whose graphs belong to C; and

(ii) all p-bounded string functions whose bit graphs belong to C.

Note that the classes FAC^0, FP, FL are defined in a different way to what we have used earlier. However, the difference will not matter in this work.

We don't repeat the definitions of definability in a theory for the two-sorted case [CN10, Definition V.4.1]. Recall the definition of Σ_0^B formulas (Definition 2.2.3).

Theorem 8.8 ([CN10, Corollary V.5.3])**.** *A function is in* FAC^0 *iff it is* Σ_0^B*-definable in* V^0*.*

Definition 8.2.6. The theory VC for a complexity class C has vocabulary $\mathcal{L}_A^2$ and is axiomatized by the axioms of V^0 and one additional axiom depending on the choice of the class C. The additional axiom can be thought of as adding an oracle for a C-complete problem to V^0. We skip the (lengthy) technicalities of [CN10, Definition IX.2.1].

The theorem below is the central result of our interest in this thesis.

Theorem 8.9 ([CN10, Theorem IX.2.3])**.** *A function is provably total (in the two-sorted sense) in* VC *iff it is in* FC*.*

Remark 8.10. By adding a single axiom to the theory of V^0, we can obtain arithmetical hierarchies in which the functions that we can define and prove correct are precisely the functions from a given complexity class $\mathsf{C} = \mathsf{FTC}^0, \mathsf{FNC}^1, \mathsf{FL}, \mathsf{FP}$.

This way, we obtain theories with very nice properties. They foster certification of complexity of an algorithm (if the proof of correctness itself is feasible, see Subsection 8.3.2). At the same time, they enable us to prove theorems about the correctness of functions defined. In [Bus+25], the authors formalize the breakthrough result $\mathsf{L} = \mathsf{SL}$ of [Rei08] inside of the weak theory of bounded arithmetic VL. The complexity of computational content of proofs of the Discrete Jordan Curve Theorem is examined in [NC12]. Expander construction in VNC^1 was conducted in [Bus+20].

Another elegant property of these theories is that the proof of a problem not being solvable in a given complexity is exactly a proof of independence of the axiom (corresponding to the problem) from the theory (corresponding to the complexity class).

Theorem 8.11 ([CN08, Corollary 7.21; CN10, Corollary VII.2.4] Independence of PHP from VAC^0)**.**

$$\mathsf{VAC}^0 \nvdash \mathit{PHP}$$

Remark 8.12 (Bibliography). The field of bounded arithmetic was initiated by Samuel Buss in his PhD thesis: [Bus86a], in which the theories S^1_2 were introduced to capture reasoning about the polynomial-time hierarchy PH (not introduced in our thesis). The first theory designed to capture polynomial-time reasoning was the equational theory PV (as in: polynomially-verifiable [proofs]) from [Coo75]. The two-sorted logic language for capturing complexity classes has been introduced by Zambella in [Zam96].

Intuitionistic counterparts such as IS^1_2 for S^1_2 and IPV for PV have also been studied. However, much less is known about their expressive power. For the relation of IS^1_2 and S^1_2, please see [Bus86b]. In particular, [Bus86b, Conjecture 3] asks: if $\mathrm{IS}^1_2 \vdash \exists y.\, \phi(y, c)$, then is it true that there is a function f, provably correct in S^1_2, such that f *computes* the Gödel encoding of that IS^1_2 proof? In [CU93, Corollary 8.19], that conjecture is answered affirmatively. The intuitionistic version IPV of the theory PV is discussed in some detail in [CU93].

For a good introduction to *bounded reverse mathematics*, with a very thorough overview of arithmetical theories corresponding to complexity classes below FP, refer to [Ngu08].

8.3. Programming in bounded arithmetic

It is not trivial to turn concepts from bounded arithmetic into a programming language.

One way is to design a programming language in such a way that reasoning about its programs can be done conveniently in a given bounded arithmetical theory. This underpins the programming language IMP(PV), introduced in [Li25] and discussed in slightly more detail in Subsection 8.3.1.

Another way, which we base our considerations on in this chapter, is by means of *code extraction*: first, provide the user with a proof assistant whose strength is limited to an appropriate arithmetical theory. Then, try to turn proofs of theorems of the form $\forall x \exists! y.\, \varphi(x, y)$, formalized in the proof assistant, into computational procedures to find the unique y, given x as input. This has been traditionally studied e.g. under the name of *Kleene realizability*. However, as of December 2025, no tool has been created that allows one to extract code *with complexity bounds*. Later in this chapter we will focus on restricting the power of a proof assistant to the arithmetical theory of interest, with the intent to transform the representations of proofs in that theory into computational procedures that are both proven correct and have a strict complexity bound, imposed by the proof being conducted in a bounded arithmetical theory.

This approach is very closely related to the notion of the Curry-Howard, or "proofs-as-programs", correspondence, which we briefly discussed in Remark 7.16. If the proof of the existence of y is conducted using intuitionistic logical reasoning only, it should be easy to convert such a proof into a program. We briefly discuss the expressiveness of intuitionistic bounded arithmetical theories in Remark 8.12. Note that for the purpose of code extraction, we do not have to limit the whole reasoning to the intuitionistic fragment of the theory — once we have intuitionistically proved *the existence* of y for a given x, we already know how to *compute* the y, and thus the proof that $\varphi(x, y)$ can be done using classical reasoning.

In general, the computational problem of finding a concrete y from a classical proof that $\exists y.\ \ldots$ is known as the *witnessing* problem. The complexity of it depends entirely on the proof system used. This is explored in more detail e.g. in [CN10, Section X.3].

For the purpose of this thesis, we have demonstrated how does it look like to extract a computational procedure operating on binary strings from a proof in Lean. Moreover, the proof we extracted from was conducted in our formalization. It is discussed alongside the file `BoundedArithmetic/Extraction.lean` below.

8.3.1. Programming language IMP(PV)

An idea for a programming language for FP based on bounded arithmetic was discussed in [Li25]. The language they discuss is IMP(PV), based on the equational theory PV, which is different from the theories we have discussed so far. They show how to design an imperative programming language with Hoare logic as the verification mechanism (i.e. as a type system). For this approach to be implementable in practice, a *formalization* of PV is necessary. Recall that, as noted in Remark 7.7, such a programming language would be very tightly coupled with the function algebra Cob. Given our considerations from Subsection 7.1.1, there is a significant risk that the convenience of such a language would be limited by the lack of intensional expressiveness of Cob and of characterizations from ICC. However, in [Li25] the authors show how to apply relatively advanced programming techniques inside IMP(PV).

Remark 8.13. We will not introduce the definition of IMP(PV) here, but will only note that it fundamentally relies on the notion of provability of theorems inside an arithmetical theory. In other words, this programming language characterizes FP in an *explicit* way, in the sense of Subsection 7.2.2. Before implementing such a programming language, we would need to formalize the arithmetical theory in a computer, which we study in Section 8.4.

8.3.2. Complexity of algorithm vs complexity of proof

Even when an algorithm is simple, it is not always trivial to "feasibly" prove that it computes the correct result. In our setting, this results in knowing that a problem can be solved in a complexity class C, but not knowing whether the corresponding function can be defined in the theory VC (i.e. proved total and correct). See [CN10, Section IX.7.3; CN08, Section 9G.3] for a discussion of the open problem whether the breakthrough result that binary integer division is in DLOGTIME-uniform TC^0 [HAM02] means that it can also be proved in the corresponding VTC^0 theory. Note that this particular problem apparently has been solved (affirmatively) in [Jeř22].

8.4. Formalization of bounded arithmetic

We present a feasible way towards formalization of results from bounded arithmetic. A key benefit of the theories studied in bounded arithmetic, compared with systems from Implicit

Computational Complexity, is that many difficult theorems from mainstream mathematics have already been reproven within them. That is, the expressiveness problems we discussed in Subsection 7.1.1 seem to affect bounded arithmetic less than the characterizations from ICC. As we will discuss in Remark 8.17, some work exists on formalizing bounded arithmetic in proof assistants; to our best information, as of November 2025, our work was the only one that studied the possibility of code extraction from proofs in two-sorted arithmetical theories. We will first discuss the core concept behind our design of the formalization, and only then proceed to investigate the individual details of it.

The code in Listing 8.1 is written in Lean 4. In it, we define `ArithModel`, which takes a type and a few proofs as arguments, and returns a `Prop`; i.e. a property of the arguments, that may or may not hold and has to be proved separately. A `structure` in Lean 4 for our purposes is simply a named tuple (here with fields `b1`, `b2`, ...). We should think of the argument `num` as a potential model of some logic `Arith` and of the other arguments (which are *typeclasses*) as interpretations of symbols from the signature in `num`. In particular, an element of type `[Zero num]` is a proof that we know *some* constant "0" of type `num`; of type `[Add num]` that we know some function "+ : `num` → `num` → `num`" etc. The typeclass arguments are most of the time inferred by Lean and if we once prove them for some type `num'`, Lean will probably be able to find the appropriate proof and insert it as an implicit argument automatically. We could get by without relying on the Lean-specific `[Zero]`, `[One]`, ... typeclasses, and instead take simply `zero:num` as argument. However, doing it this way enables us to readily use the built-in constant "0" and Lean will most of the times infer the correct type for us, that is treat the literal "0" in our context as a constant of type `num` and not the built-in Lean type `Nat` of natural numbers which also exposes this typeclass. Note that we are not taking the equality relation as argument, we use the standard equality defined in the Lean 4 logic itself.

For the fixed type `num`, its constants and functions, we can try to prove that it has the property of being an `ArithModel`. For that, we need to prove the eight properties `b1`–`b8`. They are then *bundled* in the structure `ArithModel` and having one variable h : `ArithModel num`, we can refer to the individual properties using dot-notation like in the theorem `leq_refl`, where we, instead of taking a large number of arguments, take only `h` and then refer to its individual sub-proofs as `h.b3`, `h.b8`.

Note that this approach is in contrast to actually formalizing the theory directly by defining the proof system *inside* of Lean 4, which is called a *deep embedding* of the proof system. This is also not a simple interpretation of arithmetic as a subset of the theory of Lean (a *shallow embedding*), as we are not directly translating bounded arithmetic into the language of type theory. We discuss deep and shallow embedding in slightly more detail in its own paragraph.

Our style of formalization of a theory is based on quantifying over every possible model of it. The quantification is done entirely in the Lean's type theory with its full strength — but because we are quantifying over every possible model of the theory, most of Lean's axioms don't leak into our weak logic, i.e. it is not easy to "cheat" by using Lean's type theory axioms to prove theorems in arithmetic. Nevertheless, some of the logic does leak in — as Lean is fundamentally classical, it allows proofs by contradiction. For the purpose of this thesis, this is a desired property, as the theories we are formalizing are classical. The potential existence of "cheats" is not a big problem for us, as it would probably be easy to spot them in the proofs claimed to be done in bounded arithmetic. We don't assume a setting where the proof is created by an adversary whose purpose is to cheat. Instead, our goal is to provide a framework to digitalize the existing proofs in bounded arithmetic.

Remark 8.14 (Limitations). This style of formalization will fundamentally not be able to derive proofs about the proof system itself, as the proof system used in our formalization *is*

```
-- ‘ArithModel‘ takes 6 arguments and returns a ‘Prop‘: a logical statement.
-- The arguments are an object ‘num‘ of type ‘Type‘ and 5 proofs
-- of the required properties of ‘num‘ - to have designated elements
-- ‘0‘, ‘1‘ and to have well-defined ‘+‘, ‘*‘, ... .
structure ArithModel
  (num : Type) [Zero num] [One num] [Add num] [Mul num] [LE num]
where
  b1  : ∀ x : num, x + (1 : num) ≠ 0
  b2  : ∀ x y : num, x + 1 = y + 1 → x = y
  b3  : ∀ x : num, x + 0 = x
  b4  : ∀ x y : num, x + (y + 1) = (x + y) + 1
  b5  : ∀ x : num, x * 0 = 0
  b6  : ∀ x y : num, x * (y + 1) = (x * y) + x
  b7  : ∀ x y : num, x ≤ y ∧ y ≤ x → x = y
  b8  : ∀ x y : num, x ≤ x + y

-- Introduce the 6 variables below to automatically take them as arguments
-- in the functions that use them.
variable (num : Type) [Zero num] [One num] [Add num] [Mul num] [LE num]

-- The function ‘leq_refl‘ takes 8 arguments, but Lean is able to
-- automatically infer ‘[Zero]‘ etc., and we don’t have to pass them explicitly.
-- It returns an object of type ‘x ≤ x‘;
-- the type of the type ‘x ≤ x‘ itself is ‘Prop‘.
-- Note that the return type *depends* on the value of ‘x‘ which is an argument.
theorem leq_refl (h : ArithModel num) : ∀ x : num, x <= x := by
  intro x
  conv => right; rw [<- h.b3 x]
  apply h.b8

-- As ‘ArithModel num‘ is a ‘Prop‘, it can be true; it also can be false.
-- We can prove that ‘ArithModel Nat‘ holds for a Lean built-in
-- type ‘Nat‘ of natural numbers.
theorem NatModel : ArithModel Nat :=
by
  refine {b1 := ?b1, b2 := ?b2, b3 := ?b3, b4 := ?b4,
          b5 := ?b5, b6 := ?b6, b7 := ?b7, b8 := ?b8}
  · intro x; simp -- this proves b1 for Nat
  · intro x y h
    exact Nat.add_right_cancel h -- this proves b2 for Nat
  repeat sorry -- we skip the rest of proofs for the sake of demo
```

Listing 8.1: Core theory behind formalization of arithmetic in Lean.

Lean. However, this will be possible in a formalization based on a deep embedding such as in Coq Library for First-Order Logic which is discussed below.

Another problem that is easy to spot in Listing 8.1 is that we haven't shown how to add an axiom scheme to the formalization. This is technically difficult and accounts for the vast majority of the work on formalization. One intuitive way to achieve induction *only* for Δ_0 formulas would be to detect when a user-provided Lean term such as $\forall x, x \geq x$ is of the correct shape. This would be a form of a shallow embedding for logical formulas. However, this way we encounter a huge limitation related to what the type `Prop` is: any term of that type is *just* a logical statement and we have no means of deconstructing it to the original formula. For the type-theory based proof assistants, the only way of reasoning about the complexity of logical formulas is to operate in a deep embedding of them. We made use of the fact that the independence of Continuum Hypothesis has been formalized a few years ago.[3] A part of the authors' code ended up in Lean 4's Mathlib as the `Mathlib.ModelTheory` library. It turned out that we could successfully use this library to formalize the operations on logical formulas we need to have to manipulate the formulas for the axiom schemes.

In our formalization, we use a procedure that, given a deep embedding of a logical formula, outputs a `Prop` denoting that the *interpretation* of that formula *holds* in a given model. In other words, the model *realizes* the formula. This is implemented by the function `Mathlib.ModelTheory.Semantics.Realize`. To require that the axiom scheme of induction for Δ_0 formulas holds in `num`, we need to take as argument a proof that for any φ of the type of deeply embedded formulas, if φ represents a Δ_0 formula, then `num` realizes induction for φ.

A difficult technical problem we encounter while implementing this functionality is that even for simple formulas, the operations of binding variables by quantifiers, renaming variables, substituting a term for a variable etc., compile down to very complicated Lean expressions. The expressions need to be simplified before we can use them for proving, because otherwise Lean cannot tell if they have the required type. In our experience, it could easily take one hour to simplify (using tactics such as `conv`, `simp`, `rw`) one instance of an axiom scheme before we could start proving a theorem with it. The term of the deeply embedded formula easily grew so large that it required *hundreds* of simplification operations — all executed in the right order. We have paid a lot of attention to providing useful auxiliary lemmas to simplify these formulas automatically. This part is very brittle and is the main part of the so-called `simp`-interface of our library: the transformations we can and cannot use by default in the tactic `simp`. If designed improperly, this tactic can, in the process of simplification of a hypothesis, accidentally weaken it and render the goal not provable anymore. Please see Listing 8.2 for a sample from our formalization of actual theorems of $I\Delta_0$ — without the occurrence of our simplifying macro, this example would be unreadable and tens of lines longer.

All of our code is publicly available at `https://github.com/ruplet/formalization-of-bounded-arithmetic` and all file paths mentioned below refer to this repository.

8.4.1. Our contribution

The showcase files of this thesis are `BoundedArithmetic/IOPEN.lean` and `BoundedArithmetic/IDelta0.lean`. These files contain the definitions of the IOPEN and $I\Delta_0$ theories, respectively, and then prove their basic properties that we have introduced in Lemma 8.2 and Lemma 8.3. The most important property of these formalizations we aimed to achieve was for them to be of didactic value, i.e. to make it very explicit what axioms we rely on and how we conduct

[3] Lean code available here: `https://github.com/flypitch/flypitch`.

proofs inside the weak theories. We strived to conduct our proofs one-to-one as suggested in [CN10], the book that we hugely relied on, to make it clear that we can transfer this style of reasoning from paper to computer.

In the file `BoundedArithmetic/Algebra.lean` we demonstrate a huge benefit of the formalization style we have chosen — the integration of natural numbers as defined in $\mathrm{I}\Delta_0$ with the algebraic structures from the mainstream `Mathlib.Algebra`. The effect of such integration is that we will *not* have to re-prove all the theorems about number theory in bounded arithmetic; after proving the basic properties, we can derive that the natural numbers modeled by $\mathrm{I}\Delta_0$ satisfy the properties of a semiring; of a commutative monoid; that the addition is right-cancellable, etc., "unlocking" more and more theorems from `Mathlib`.

In the file `BoundedArithmetic/AxiomSchemes.lean` we present readable definitions of the axiom schemes of induction and comprehension, and the aforementioned macros that *reduce* them.

The file `BoundedArithmetic/V0.lean` is a record of our work with two-sorted logic and the comprehension axiom scheme for it. The `Mathlib.ModelTheory` library for now only supports single-sorted logic. Nevertheless, we can use it to reason about two-sorted logic by utilizing the idea from [CN10, Section IV.5]. We have initially started with extending this library to support many sorts, but this was too much work for our time horizon. Thus, we decided to focus on single-sorted theories with axiom schemes and the finitely axiomatizable fragments of two-sorted theories.[4]

In the file `BoundedArithmetic/VTC0.lean`, we sketch the proof of the two-sorted theory VTC^0 proving the pigeonhole principle `PHP` (note that in Theorem 8.11 we discussed this theorem not being provable in V^0), without relying on any complicated deep-embedding for the axiom schemes (we introduce single instances of them as additional axioms instead). We strived to make the formalization go one-to-one with the original [CN08, Corollary 7.21; CN10, Corollary VII.2.4].

In the folder `syntax-metaprogramming`, we present examples of how a simple proof theory can be deeply embedded in the logic of a proof assistant. We present a short demonstration of how a proof looks in the deeply embedded logical framework of Coq Library for First-Order Logic in the file `syntax-metaprogramming/roq-fol-proof-mode.v`. In the other `*.lean` files, we present how to deeply embed a simple Hilbert-style proof system on the meta-level of Lean — i.e. how to use metaprogramming in Lean 4 to obtain syntactic abstraction for the embedded logic.

Comparison: available proof assistants Modern proof assistants are generally built on very strong underlying logics. Some systems, such as Metamath, Twelf, LF, or Isabelle, can be used as frameworks to formalize a variety of object logics and thus to reason *about* logics themselves. Note we did not mention Isabelle/HOL here, since it is a particular (and very strong) logic implemented *inside* of Isabelle. One could instead work in Isabelle/Pure, which only introduces the most basic axioms, but it requires expertise in Isabelle to ensure that an axiomatization of bounded arithmetic does not "cheat" by exploiting additional axioms of Isabelle/Pure to conduct reasoning stronger than those of the weak theory. The other metamathematical frameworks mentioned above did not seem to offer a clear advantage over the approach we ultimately adopted. We therefore chose to formalize in two powerful and well-supported systems, Rocq and Lean, both based on dependent type theory.

Remark 8.15. The Rocq proof assistant used to be called *Coq* until March 2025. While the

[4] Actually, the theory V^0 *is* finitely axiomatizable, but since most existing proofs use axiom schemes, we would lose the property of translating proofs faithfully from paper to computer.

old name is now deprecated, existing publications and code written before the name change still use the name Coq. The historical versions of the system have not been renamed: for example, we install "`coq.8.20.1`" and not "`rocq...`". Unfortunately, all results that we cite in this thesis were obtained using versions released under the name Coq. We retain the name Coq when referring to such prior work, and use the name Rocq when discussing the proof assistant that we have actually used ourselves.

Comparison: deep and shallow embeddings To formalize a logic in a proof assistant, the most obvious way is to define the proof system from scratch inside of it, then prove properties about the proof system and design a good UI to make it usable for actual proving. This approach was chosen by the authors of Coq Library for First-Order Logic.[5][6] Going this way is a tremendous amount of work — in fact, it cannot be much simpler than essentially rewriting Rocq from scratch.

However, it is not obvious how to go around this problem. It is impossible to disable the axioms assumed by the foundations of Rocq and Lean. We certainly wouldn't like to mix $\mathrm{I}\Delta_0$ induction with the axioms of dependent type theory and hope for the best. There is one very elegant approach we can use to formalize a *finitely* axiomatizable theory. We discuss this in detail in Listing 8.1. We also cite two resources for more information on the notion of deep and shallow embeddings. Please see [AK16] for formalization of type theory inside of type theory; [PKL22] for discussion specifically on deep and shallow embeddings.

Remark 8.16 (Origins of this thesis). We have only started investigating bounded arithmetic for our problem in March of 2025 and it was then unclear if the results from this field will transfer to anything actually computational. Only in July 2025 did we decide to implement the formalization and quickly obtained a formalization of a prototype finitely axiomatizable theory like in Listing 8.1. In August of 2025 we obtained a working version of the full formalization of $\mathrm{I}\Delta_0$. In the first week of September 2025 we have presented our project at AITP2025 conference in Aussois, France: the main motivation behind the presentation was the importance of creating proof assistants with theories that are *weaker* than Rocq and Lean's and admitting better proof-search or code-extraction procedures. Only later in September of 2025 as part of us visiting Yannick Forster at INRIA Paris, we have achieved a usable state of the formalization with a well-thought `simp`-set for $\mathrm{I}\Delta_0$ and interesting properties proved for V^0 and VTC^0. The existence of the `Mathlib.ModelTheory` library was the thing that enabled us to achieve our formalization in the scope of this thesis.

Remark 8.17 (Related works). There is very little work available on the formalization of arithmetic. A formalization of consistency of Peano arithmetic in Coq was presented in [OCo05]. A formalization of the so-called *Hydra battles* related to unprovability in Peano arithmetic was shown in [Cas+22]. There is an impressive ongoing project of formalization of bounded arithmetic in the model-theoretical style in the Lean community.[7] Their approach doesn't align with our goal of certifying complexity, as their focus is on other arithmetical theories, which differ significantly from what we need. Somewhat related, some work on intuitionistic logic in Lean has also been done in [Tru24] even though Lean is not a natural environment for intuitionistic thinking, as it assumes classical axioms very deeply in its standard libraries, unlike Rocq which is constructive at heart.

[5] Available here: `https://github.com/uds-psl/coq-library-fol/`.

[6] Studying this approach was the topic of my research visit at INRIA in September 2025, funded by the ZSM IDUB program.

[7] `https://github.com/FormalizedFormalLogic/Foundation`

```
-- D1. x ≠ 0 → ∃ y ≤ x, x = y + 1  (Predecessor)
-- proof: induction on x
theorem pred_exists :
  ∀ {x : M}, x ≠ 0 → ∃ y ≤ x, x = y + 1 :=
by
  -- peano.Formula (Vars2 .y .x) is the type of deeply-embedded formulas
  -- with two free variables, "y" and "x"
  let ind1 : peano.Formula (Vars2 .y .x) := x =' (y + 1)
  -- similarly below, "y" cannot be free in ‘ind2‘.
  -- ‘iBdEx’ t phi‘ denotes ‘∃ var ≤ t . phi‘ for some variable name ‘var‘
  let ind2 : peano.Formula (Vars1 .x) :=
    (Formula.iBdEx' x (display2 .y ind1).flip)
  -- ‘display1‘ takes a formula with 1 free variable and makes it explicit
  -- that it is intended to be bound by the induction axiom scheme
  let ind := idelta0.delta0_induction $ display1 $ (x ≠' 0) ⟹ ind2

  -- prove that the formula actually is Delta0
  unfold ind2 ind1 at ind
  specialize ind (by
    rw [IsDelta0.display1]
    rw [IsDelta0.of_open.imp]
    · constructor
      · unfold Term.neq
        rw [IsDelta0.of_open.not]
        constructor; constructor; constructor
        constructor; constructor
      · constructor
    · unfold Term.neq
      rw [IsOpen.not]
      constructor; constructor
  )
  -- use our custom macro to simplify ‘ind‘ to use it in proving
  simp_induction at ind
  -- proceed with the proof by induction; name subgoals; clear helpers
  apply ind ?base ?step <;> clear ind ind1 ind2
  · simp only [IsEmpty.forall_iff]
  · intro a hind h
    exists a
    constructor
    · exact B8 a 1
    · rfl
```

Listing 8.2: One actual proof formalized in our system

Appendix A

Uniformity

In this chapter, we focus on the descriptions of uniformity conditions used for families of circuits. There is a very thorough overview of uniformity conditions for complexity classes below NC^1 in [MIS90].

A.1. FO-uniformity

The definition of first-order uniformity is rather technical and is based on the notion of first-order queries, introduced in [Imm99, Definition 5.16]. One of the results that we discussed which uses this notion is Theorem 5.8.

A.2. U_{E^*}-uniformity

The notions of U_E and U_{E^*}-uniformity (sometimes also called E^*-uniformity) were studied in early work on circuit uniformity. These notions are defined in terms of the *direct connection language* and the *extended connection language* of a circuit family; see [Vol99, Definitions 2.24 and 2.43].

Since then, U_E- and U_{E^*}-uniformity have largely been displaced by DLOGTIME-uniformity. Interesting arguments about why DLOGTIME uniformity is the most reasonable to consider are presented in a breakthrough paper proving that binary integer division is in (DLOGTIME-uniform) TC^0 [HAM02].

A.3. DLOGTIME-uniformity

For very low complexity classes, the most commonly used notion of uniformity is DLOGTIME-uniformity, based on random-access Turing machines. Similarly as in Section A.2, this notion is also based on direct and extended connection language of a circuit family. A circuit family is DLOGTIME-uniform when we can decide its direct connection language on a random-access Turing machine in logarithmic time. For the details, we refer to: [MIS90, Section 6].

DLOGTIME-uniformity has some very elegant properties. The strength of such Turing machines is well-studied. It's mentioned that $\mathsf{AC}^0_0 \subseteq \mathsf{DLOGTIME} \subseteq \mathsf{AC}^0_2$ in [Lee91, Page 141], where AC^0_k denotes AC^0 circuits of depth k. It has also been shown that the class DLOGTIME-uniform NC^1 is equal to ALOGTIME (alternating logarithmic time, which we don't introduce here) and also equal to NC^1-uniform NC^1: [MIS90, Lemma 6.2].

Remark A.1 (Bibliography). We refer to a source that is difficult to access (e.g. the above [Lee91, Page 141]): this is Chapter 2 "A Catalog of Complexity Classes" by David S. Johnson [Joh91], which appeared in January 1991 in "Handbook of theoretical computer science (vol. A): algorithms and complexity", edited by Jan van Leeuwen [Lee91].

Appendix B

Definition of Neergaard's safe affine recursion

We continue to write functions as $f(\vec{x};\vec{y})$, with normal inputs $\vec{x}$ and safe inputs $\vec{y}$. For $\delta \in \mathbb{N}$, put $\texttt{shift}(y,\delta) = \lfloor y/2^{\delta} \rfloor$, i.e. the number obtained from y by dropping its δ least significant bits. Let $\mathbb{N}_2$ denote the natural numbers in binary notation. For $m, n \geq 0$, we write $\mathbb{N}_2^{m,n}$ for the set of functions with m normal and n safe arguments. All the arguments and the return value are in $\mathbb{N}_2$.

Definition B.0.1 ([Nee04] Neergaard's $\mathsf{BC}_{\varepsilon}^{-}$ algebra)**.** The class $\mathsf{BC}_{\varepsilon}^{-}$ is the smallest class of functions over $\mathbb{N}_2$ that contains the initial functions (i)–(v) and is closed under (vi) and (vii) below:

(i) **(zero)** $0(;) = \varepsilon$;

(ii) **(predecessor)** $p(;\varepsilon) = \varepsilon, \quad p(;yb) = y \quad \text{for } b \in \{0,1\}$;

(iii) **(projections)** for $m, n \geq 0$ and $1 \leq j \leq m+n$, $\pi_j^{m,n}(x_1,\ldots,x_m;x_{m+1},\ldots,x_{m+n}) = x_j$;

(iv) **(successors)** $s_0(;y) = y0, \quad s_1(;y) = y1$;

(v) **(conditional)**

$$c(;y_1,y_2,y_3) = \begin{cases} y_2 & \text{if } y_1 \text{ ends in 1 (i.e. } y_1 = z1 \text{ for some } z), \\ y_3 & \text{otherwise;} \end{cases}$$

(vi) **(safe affine composition)** let $f \in \mathsf{BC}_{\varepsilon}^{-}$ have arity (M,N), let $g_1,\ldots,g_M \in \mathsf{BC}_{\varepsilon}^{-}$ with g_i of arity $(m,0)$, and let $h_1,\ldots,h_N \in \mathsf{BC}_{\varepsilon}^{-}$ with h_i of arity (m,n_i). Put $n := n_1+\cdots+n_N$. The *safe affine composition* $f \circ \langle g_1,\ldots,g_M;h_1,\ldots,h_N\rangle$ is the function F of arity (m,n) defined by

$$F(\vec{x};\vec{y}) = f\Big(g_1(\vec{x};),\ldots,g_M(\vec{x};)\ ;\ h_1(\vec{x};\vec{y}_1),\ldots,h_N(\vec{x};\vec{y}_N)\Big),$$

where each $\vec{y}_j$ is a (possibly empty) subtuple of the safe inputs $\vec{y}$ and every safe variable occurs in *at most* one of the tuples $\vec{y}_1,\ldots,\vec{y}_N$. In particular, no safe input may be duplicated, but some safe inputs may be unused.

(vii) **(safe affine course-of-value recursion)** let $g, h_0, h_1, d_0, d_1 \in \mathsf{BC}_{\varepsilon}^{-}$ be such that

$$g : \mathbb{N}_2^{m,n} \to \mathbb{N}_2, \quad h_0, h_1 : \mathbb{N}_2^{m+1,1}, \quad d_0, d_1 : \mathbb{N}_2^{m+1,0}.$$

```
-- Proposition 1. Let m and n by numbers in binary. Right shift shiftR(m : n) of
-- m by |n| and selection of bit |n| from m are definable in BCeps-.

-- shiftR(n : m) = m >> |n|
shiftR :: Func
shiftR =
  let g = Proj 0 1 1 in
  let d = oneNormalToZero in
  -- h(timer : recursive) = tail(recursive)
  let h = Composition 0 1 1 [1] Tail [] [Proj 1 1 2] in
  Recursion 0 1 g h h d d
```

Listing B.1: Function `shiftR` from the original paper.

Safe affine course-of-value recursion of them is the function $f : \mathbb{N}_2^{m+1,n}$ given by

$$f(0, \vec{x}; \vec{y}) = g(\vec{x}; \vec{y}),$$
$$f(tb, \vec{x}; \vec{y}) = h_b\Big(t, \vec{x};\ f(\mathtt{shift}(t, \delta_b), \vec{x}; \vec{y})\Big)$$

for $b \in \{0, 1\}$, where $\delta_b = |d_b(t, \vec{x};\ )|$ depends only on the normal arguments $(y, \vec{x})$. Here the recursive value $f(\mathtt{shift}(t, \delta_b), \vec{x}; \vec{y})$ is supplied to h_b as its unique safe argument and can therefore be used at most once.

Theorem B.1 ([Nee04, Theorem 1])**.** *For any function definable in* $\mathsf{BC}_\varepsilon^-$ *there is a Turing machine evaluating the function in* FL*. The Turing machine can be constructed from the function expression in logarithmic space in the size of the* $\mathsf{BC}_\varepsilon^-$*-expression.*

Remark B.2 (Intuition)*.* The recursive result is passed back in a safe position and can be used only once. Once a safe value is "measured" by the conditional operator, it must be recomputed; we *cannot* duplicate it. Dropping the affinity constraint collapses back to the original BC algebra for FP.

Remark B.3 (Bibliography)*.* In [MO00a], with further refinements in [MO04], BC^- (note the lack of ε subscript) was introduced, an algebra that was contained in FL, but was not known (and unlikely) to be FL-complete. In [Nee04] this was improved to the result that $\mathsf{BC}_\varepsilon^- = \mathsf{FL}$, with a short discussion that using course-of-value affine recursion instead of predicative affine recursion seems to be the reason why $\mathsf{BC}_\varepsilon^-$ is FL-complete, whereas BC^- is probably not.

Writing programs in $\mathsf{BC}_\varepsilon^-$ is not reminiscent of any mainstream programming language. The types being linear make most of the standard programming techniques not usable. As part of our work, we have implemented an interpreter for the algebra in Haskell and tried to reproduce some of the propositions from [Nee04]. We present one listing, just to give a hint of how the corresponding function look like; this is a Haskell representation of a function from the algebra, not a programming language directly for the algebra. We present the full source code as attachment to this thesis.

Remark B.4. The example in Listing B.1 shows the implementation of a function that shifts bit to the right (dropping least-significant bits). The implementation is standard, but it's worth to note that in the original paper, there is an error — the function `shiftR` originally was given with flipped safe and normal arguments, making it impossible to implement. We outline our proof by contradiction for this fact. Consider a function $\mathtt{cond}'(;\mathtt{x},\mathtt{y},\mathtt{z})$ returning

y if x is empty, z otherwise. We can prove directly that this function is not implementable in BC^- due to the algebra not being able to differentiate between an empty string and a string beginning with a sufficiently long prefix of zeros. Assume the original `shiftR` is implementable. Then we can implement `cond′` using `shiftR`, which is a contradiction.

Bibliography

[AB07] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. USA, 2007. URL: https://theory.cs.princeton.edu/complexity/book.pdf.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 0521424267.

[Ada11] Rod Adams. *An Early History of Recursive Functions and Computability. From Gödel to Turing*. Reprint/revision of the author's 1983 Ph.D. thesis. Boston: Docent Press, May 28, 2011. 297 pp. ISBN: 978-0-9837004-0-1. URL: https://books.google.com/books?id=G9YoeRIVSnwC (visited on 10/29/2025).

[ADK25] Melissa Antonelli, Arnaud Durand, and Juha Kontinen. *Characterizing Small Circuit Classes from FAC^0 to FAC^1 via Discrete Ordinary Differential Equations*. 2025. arXiv: 2506.23404 [cs.CC]. URL: https://arxiv.org/abs/2506.23404.

[AK16] Thorsten Altenkirch and Ambrus Kaposi. "Type theory in type theory using quotient inductive types". In: *SIGPLAN Not.* 51.1 (Jan. 2016), pp. 18–29. ISSN: 0362-1340. DOI: 10.1145/2914770.2837638. URL: http://web.archive.org/web/20250815043643/https://people.cs.nott.ac.uk/psztxa/publ/tt-in-tt.pdf.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. "PRIMES Is in P". In: *Annals of Mathematics* 160.2 (2004), pp. 781–793. ISSN: 0003486X. URL: http://web.archive.org/web/20250303132722/https://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf (visited on 11/19/2025).

[All91] Bill Allen. "Arithmetizing Uniform NC". In: *Annals of Pure and Applied Logic* 53.1 (1991), pp. 1–50. ISSN: 0168-0072. DOI: https://doi.org/10.1016/0168-0072(91)90057-S. URL: https://www.sciencedirect.com/science/article/pii/016800729190057S.

[AR12] Andrea Asperti and Wilmer Ricciotti. "Formalizing Turing Machines". In: *Logic, Language, Information and Computation*. Ed. by Luke Ong and Ruy de Queiroz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–25. ISBN: 978-3-642-32621-9.

[Aub15] Clément Aubert. "An in-between "implicit" and "explicit" complexity: Automata". In: *DICE 2015 - Developments in Implicit Computational Complexity*. Londres, United Kingdom, Apr. 2015. URL: https://hal.science/hal-01111737.

[Bal25] Paweł Balawender. *Seminar Presentation on the Paper "A Formalization of Borel Determinacy in Lean"*. Seminar talk; the paper presented is not by the author. Mar. 2025. URL: https://github.com/ruplet/oracles/blob/f25def5b38a4b1ee60d81c1dcacc765e4fb53595/seminar-presentation-borel-determinacy/prezentacja.pdf (visited on 11/08/2025).

[Bau06] Andrej Bauer. "First Steps in Synthetic Computability Theory". In: *Electronic Notes in Theoretical Computer Science* 155 (2006). Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI), pp. 5–31. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.11.049. URL: https://www.sciencedirect.com/science/article/pii/S1571066106001861.

[BC92] Stephen Bellantoni and Stephen Cook. "A new recursion-theoretic characterization of the polytime functions". In: *Comput. Complex.* 2.2 (Dec. 1992), pp. 97–110. ISSN: 1016-3328. DOI: 10.1007/BF01201998. URL: https://doi.org/10.1007/BF01201998.

[BD19] Olivier Bournez and Arnaud Durand. "Recursion Schemes, Discrete Differential Equations and Characterization of Polynomial Time Computations". In: *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*. Ed. by Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen. Vol. 138. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 23:1–23:14. ISBN: 978-3-95977-117-7. DOI: 10.4230/LIPIcs.MFCS.2019.23. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2019.23.

[Ben15] Erika Benedetti. "Linear logic, type assignment systems and implicit computational complexity". In: (Feb. 2015). URL: http://web.archive.org/web/20240218234232/https://theses.hal.science/tel-01123737v1/file/2015ENSL0981.pdf.

[BG92] M. Bellare and S. Goldwasser. *The Complexity of Decision versus Search*. Tech. rep. USA, 1992. URL: http://web.archive.org/web/20250714180631/https://cseweb.ucsd.edu/~mihir/papers/compip.pdf.

[Blo94] Stephen Bloch. "Function-algebraic characterizations of log and polylog parallel time". In: *Computational Complexity* 4.2 (June 1994), pp. 175–205. ISSN: 1420-8954. DOI: 10.1007/BF01202288. URL: https://doi.org/10.1007/BF01202288.

[Bob+11] François Bobot et al. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, Aug. 2011, pp. 53–64.

[Boj18] Mikołaj Bojańczyk. *Polyregular Functions*. 2018. arXiv: 1810.08760 [cs.FL]. URL: https://arxiv.org/abs/1810.08760.

[Bon+16] Guillaume Bonfante et al. "Two function algebras defining functions in NC^k Boolean circuits". In: *Inf. Comput.* 248.C (June 2016), pp. 82–103. ISSN: 0890-5401. DOI: 10.1016/j.ic.2015.12.009. URL: https://doi.org/10.1016/j.ic.2015.12.009.

[Bon06] Guillaume Bonfante. "Some Programming Languages for Logspace and Ptime". In: *Algebraic Methodology and Software Technology*. Ed. by Michael Johnson and Varmo Vene. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 66–80. ISBN: 978-3-540-35636-3.

[Büc60] J. Richard Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 6 (1960), pp. 66–92. URL: http://web.archive.org/web/20251118183104/https://2024.sci-hub.st/173/04978b7adf46fde189eab42885cdc1d8/10.1002@malq.19600060105.pdf.

[Bus+20] Sam Buss et al. "Expander construction in VNC1". In: *Annals of Pure and Applied Logic* 171.7 (2020), p. 102796. ISSN: 0168-0072. DOI: https://doi.org/10.1016/j.apal.2020.102796. URL: https://www.sciencedirect.com/science/article/pii/S0168007220300208.

[Bus+25] Sam Buss et al. *A Logspace Constructive Proof of L=SL*. 2025. arXiv: 2511.12011 [cs.LO]. URL: https://arxiv.org/abs/2511.12011.

[Bus86a] Samuel R. Buss. *Bounded Arithmetic*. Studies in Proof Theory: Lecture Notes 3. Naples: Bibliopolis, 1986. ISBN: 8870881504. URL: http://web.archive.org/web/20240421184047/https://mathweb.ucsd.edu/~sbuss/ResearchWeb/BAthesis/Buss_Thesis_OCR.pdf.

[Bus86b] Samuel R. Buss. "The polynomial hierarchy and intuitionistic Bounded Arithmetic". In: *Structure in Complexity Theory*. Ed. by Alan L. Selman. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 77–103. ISBN: 978-3-540-39825-7.

[Cas+22] Pierre Castéran et al. "Hydras & Co.: Formalized mathematics in Coq for inspiration and entertainment". In: *Journées Francophones des Langages Applicatifs: JFLA 2022*. St-Médard d'Excideuil, France, June 2022. URL: https://hal.science/hal-03404668.

[CFI92] Jin-Yi Cai, Martin Fürer, and Neil Immerman. "An optimal lower bound on the number of variables for graph identification". In: *Combinatorica* 12.4 (1992), pp. 389–410. ISSN: 1439-6912. DOI: 10.1007/BF01305232. URL: https://doi.org/10.1007/BF01305232.

[Cho08] Timothy Y. Chow. *A beginner's guide to forcing*. 2008. arXiv: 0712.1320 [math.LO]. URL: https://arxiv.org/abs/0712.1320.

[Cia16] Alberto Ciaffaglione. "Towards Turing computability via coinduction". In: *Science of Computer Programming* 126 (2016). Selected Papers from the 17th Brazilian Symposium on Formal Methods (SBMF 2014), pp. 31–51. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2016.02.004. URL: https://www.sciencedirect.com/science/article/pii/S0167642316000484.

[CL90] Kevin J. Compton and Claude Laflamme. "An algebra and a logic for NC1". In: *Information and Computation* 87.1 (1990). Special Issue: Selections from 1988 IEEE Symposium on Logic in Computer Science, pp. 241–263. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(90)90063-N. URL: https://www.sciencedirect.com/science/article/pii/089054019090063N.

[CM87] Stephen A Cook and Pierre McKenzie. "Problems complete for deterministic logarithmic space". In: *Journal of Algorithms* 8.3 (1987), pp. 385–394. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(87)90018-6. eprint: https://web.archive.org/web/20161213142928/https://www.cs.utoronto.ca/~sacook/homepage/cook_mckenzie.pdf. URL: http://web.archive.org/web/20250714180631/https://cseweb.ucsd.edu/~mihir/papers/compip.pdf.

[CN08] Stephen A. Cook and Phuong Nguyen. “Logical Foundations of Proof Complexity”. Draft manuscript. 2008. URL: http://web.archive.org/web/20250719042008/https://www.cs.toronto.edu/~sacook/homepage/book/ (visited on 09/03/2025).

[CN10] Stephen Cook and Phuong Nguyen. *Logical Foundations of Proof Complexity*. Cambridge University Press, 2010. DOI: 10.1017/CBO9780511676277. (Visited on 09/03/2025).

[Cob64] Alan Cobham. “The Intrinsic Computational Difficulty of Functions”. In: *Logic, methodology and philosophy of science*. Ed. by Yehoshua Bar-Hillel. North-Holland Pub. Co., 1964, pp. 24–30. URL: https://web.archive.org/web/20240121142633/https://www.cs.toronto.edu/~sacook/homepage/cobham_intrinsic.pdf.

[Coo71] Stephen A. Cook. “Characterizations of Pushdown Machines in Terms of Time-Bounded Computers”. In: *J. ACM* 18.1 (Jan. 1971), pp. 4–18. ISSN: 0004-5411. DOI: 10.1145/321623.321625. URL: https://doi.org/10.1145/321623.321625.

[Coo75] Stephen A. Cook. “Feasibly constructive proofs and the propositional calculus (Preliminary Version)”. In: *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*. STOC ’75. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1975, pp. 83–97. ISBN: 9781450374194. DOI: 10.1145/800116.803756. URL: https://doi.org/10.1145/800116.803756.

[Coo83] Stephen A. Cook. “An overview of computational complexity”. In: *Commun. ACM* 26.6 (June 1983), pp. 400–408. ISSN: 0001-0782. DOI: 10.1145/358141.358144. URL: https://doi.org/10.1145/358141.358144.

[Coo85] Stephen A. Cook. “A taxonomy of problems with fast parallel algorithms”. In: *Information and Control* 64.1 (1985). International Conference on Foundations of Computation Theory, pp. 2–22. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(85)80041-3. eprint: https://web.archive.org/web/20190320200248/https://core.ac.uk/download/pdf/81978561.pdf. URL: https://www.sciencedirect.com/science/article/pii/S0019995885800413.

[Cou94] Bruno Courcelle. “Monadic second-order definable graph transductions: a survey”. In: *Theoretical Computer Science* 126.1 (1994), pp. 53–75. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(94)90268-2. URL: https://www.sciencedirect.com/science/article/pii/0304397594902682.

[CS12] Jacek Chrzaszcz and Aleksy Schubert. “ML with PTIME complexity guarantees”. In: *Computer Science Logic (CSL’12) - 26th International Workshop/21st Annual Conference of the EACSL*. Ed. by Patrick Cégielski and Arnaud Durand. Vol. 16. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012, pp. 198–212. ISBN: 978-3-939897-42-2. DOI: 10.4230/LIPIcs.CSL.2012.198. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2012.198.

[CU93] Stephen Cook and Alasdair Urquhart. “Functional interpretations of feasibly constructive arithmetic”. In: *Annals of Pure and Applied Logic* 63.2 (1993), pp. 103–200. ISSN: 0168-0072. DOI: https://doi.org/10.1016/0168-0072(93)90044-E. URL: https://www.sciencedirect.com/science/article/pii/0168007293900 44E.

[Dal12] Ugo Dal Lago. "A Short Introduction to Implicit Computational Complexity". In: *Lectures on Logic and Computation: ESSLLI 2010 Copenhagen, Denmark, August 2010, ESSLLI 2011, Ljubljana, Slovenia, August 2011, Selected Lecture Notes*. Ed. by Nick Bezhanishvili and Valentin Goranko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 89–109. ISBN: 978-3-642-31485-8. DOI: 10.1007/978-3-642-31485-8_3. URL: https://doi.org/10.1007/978-3-642-31485-8_3.

[Dan+01] Evgeny Dantsin et al. "Complexity and expressive power of logic programming". In: *ACM Comput. Surv.* 33.3 (Sept. 2001), pp. 374–425. ISSN: 0360-0300. DOI: 10.1145/502807.502810. URL: http://web.archive.org/web/20240415065931/https://dai.fmph.uniba.sk/~sefranek/bak/dantsin97complexity.pdf.

[Daw12] Anuj Dawar. *On Syntactic and Semantic Complexity Classes*. Presentation at the Spitalfields Day, Isaac Newton Institute. Available online via Newton Institute archive. Jan. 2012. URL: https://web.archive.org/web/20210515020503/https://www.newton.ac.uk/files/seminar/20120109163017301-152985.pdf.

[Ded88] Richard Dedekind. *Was sind und was sollen die Zahlen?* German. Erstausgabe 1888; Druckausgabe: Braunschweig, 4. Aufl., 1918. Aachen: semantics Kommunikationsmanagement GmbH, 1888. URL: https://archive.org/details/WasSindUndWasSollenDieZahlen.

[DKO21] Ugo Dal Lago, Reinhard Kahle, and Isabel Oitavem. "A Recursion-Theoretic Characterization of the Probabilistic Class PP". In: *46th International Symposium on Mathematical Foundations of Computer Science (MFCS 2021)*. Ed. by Filippo Bonchi and Simon J. Puglisi. Vol. 202. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 35:1–35:12. ISBN: 978-3-95977-201-3. DOI: 10.4230/LIPIcs.MFCS.2021.35. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2021.35.

[DKO22] Ugo Dal Lago, Reinhard Kahle, and Isabel Oitavem. "Implicit recursion-theoretic characterizations of counting classes". In: *Arch. Math. Log.* 61.7-8 (Nov. 2022), pp. 1129–1144. ISSN: 0933-5846. DOI: 10.1007/s00153-022-00828-4. URL: https://doi.org/10.1007/s00153-022-00828-4.

[DM06] Ugo Dal Lago and Simone Martini. *Implicit Computational Complexity: A μ-Tutorial*. Tutorial talk at the FOLLIA meeting. FOLLIA meeting, January 19, 2006. Università di Bologna, Dipartimento di Scienze dell'Informazione, Jan. 2006. URL: http://web.archive.org/web/20221225041239/http://www.cs.unibo.it/~martini/TALKS/follia2006.pdf.

[DS16] Ugo Dal Lago and Ulrich Schöpp. "Computation by interaction for space-bounded functional programming". In: *Information and Computation* 248 (2016). Development on Implicit Computational Complexity (DICE 2013), pp. 150–194. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2015.04.006. URL: https://www.sciencedirect.com/science/article/pii/S089054011500142X.

[EH99] Joost Engelfriet and Hendrik Jan Hoogeboom. *MSO definable string transductions and two-way finite state transducers*. 1999. arXiv: cs/9906007 [cs.LO]. URL: https://arxiv.org/abs/cs/9906007.

[Elg61] Calvin C. Elgot. “Decision Problems of Finite Automata Design and Related Arithmetics”. In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51. ISSN: 00029947. URL: http://web.archive.org/web/20250709105150/https://www.ams.org/journals/tran/1961-098-01/S0002-9947-1961-0139530-9/S0002-9947-1961-0139530-9.pdf (visited on 11/18/2025).

[Fag74] Ronald Fagin. “Generalized First-Order Spectra and Polynomial-Time Recognizable Sets”. In: *Complexity of Computation*. SIAM-AMS Proceedings 7. American Mathematical Society, 1974, pp. 43–73.

[FKW20] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. “Verified programming of Turing machines in Coq”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 114–128. ISBN: 9781450370974. DOI: 10.1145/3372885.3373816. URL: https://doi.org/10.1145/3372885.3373816.

[For25] Yannick Forster. “Synthetic Mathematics for the Mechanisation of Computability Theory and Logic”. In: *33rd EACSL Annual Conference on Computer Science Logic (CSL 2025)*. Ed. by Jörg Endrullis and Sylvain Schmitz. Vol. 326. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 3:1–3:2. ISBN: 978-3-95977-362-1. DOI: 10.4230/LIPIcs.CSL.2025.3. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2025.3.

[Gir87] Jean-Yves Girard. “Linear logic”. In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4. URL: https://www.sciencedirect.com/science/article/pii/0304397587900454.

[Göb11] Andreas-Nikolas Göbel. “The Computational Compl xity of Computing Nash Equilibrium”. Seminar presentation, Algorithmic Game Theory. June 2011. URL: https://old.corelab.ntua.gr/courses/gametheory/old/1011/slides/agob-slides.pdf.

[Gol21] Paul W. Goldberg. “Search Problems, and Total Search Problems”. Computational Complexity, slides 15; Hilary Term (HT) 2021. 2021. URL: https://www.cs.ox.ac.uk/people/paul.goldberg/CC/2020-21/slides15.pdf.

[GP18] Paul W. Goldberg and Christos H. Papadimitriou. “Towards a Unified Complexity Theory of Total Functions”. In: *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*. Ed. by Anna R. Karlin. Vol. 94. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 37:1–37:20. DOI: 10.4230/LIPICS.ITCS.2018.37. URL: https://doi.org/10.4230/LIPIcs.ITCS.2018.37.

[Gra61] Hermann Grassmann. *Lehrbuch der Arithmetik für höhere Lehranstalten*. German. Verlag von T. C. F. Enslin (Adolph Enslin), 1861, p. 233. URL: https://archive.org/details/lehrbuchderarit00grasgoog/page/n48/mode/2up.

[Gro12] Joshua Grochow. *Forcing method used in Baker-Gill-Solovay Relativization paper and Cohens Proof of Continuum Hypothesis Independence*. Theoretical Computer Science Stack Exchange. Oct. 2012. eprint: https://cstheory.stackexchange.com/q/14093. URL: https://cstheory.stackexchange.com/q/14093.

[Grz53] Andrzej Grzegorczyk. *Some classes of recursive functions*. eng. source: http://eudml.org/doc/219317. Warszawa: Instytut Matematyczny Polskiej Akademi Nauk, 1953. URL: http://web.archive.org/web/20250806010342/http://matwbn.icm.edu.pl/ksiazki/rm/rm04/rm0401.pdf.

[GS89] Yuri Gurevich and Saharon Shelah. "Nearly linear time". In: *Logic at Botik '89*. Ed. by Albert R. Meyer and Michael A. Taitslin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 108–118. ISBN: 978-3-540-46180-7.

[Gur12] Yuri Gurevich. "What Is an Algorithm?" In: *SOFSEM 2012: Theory and Practice of Computer Science*. Ed. by Mária Bieliková et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 31–42. ISBN: 978-3-642-27660-6. URL: http://web.archive.org/web/20240515063009/https://web.eecs.umich.edu/~gurevich/Opera/209.pdf.

[Gur83] Yuri Gurevich. "Algebras of feasible functions". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. 1983, pp. 210–214. DOI: 10.1109/SFCS.1983.5.

[HAM02] William Hesse, Eric Allender, and David A. Mix Barrington. "Uniform constant-depth threshold circuits for division and iterated multiplication". In: *Journal of Computer and System Sciences* 65.4 (2002). Special Issue on Complexity 2001, pp. 695–716. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(02)00025-9. URL: https://www.sciencedirect.com/science/article/pii/S0022000002000259.

[Har93] Juris Hartmanis. "Sparse Complete Sets for NP and the Optimal Collapse of the Polynomial Hierarchy". In: *Current Trends in Theoretical Computer Science*. 1993, pp. 403–411. DOI: 10.1142/9789812794499_0029. eprint: https://dblp.org/rec/series/wsscs/Hartmanis93b.html. URL: https://books.google.pl/books?hl=en&lr=&id=kVhZDTKYa8MC&oi=fnd&pg=PA403&ots=igK0eGCAtC&sig=hnXMtnMszE1gpgVW0a8qaH2-7es&redir_esc=y#v=onepage&q&f=false.

[HK96] Gerd Hillebrand and Paris Kanellakis. "On the Expressive Power of Simply Typed and Let-Polymorphic Lambda Calculi". In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. LICS '96. USA: IEEE Computer Society, 1996, p. 253. ISBN: 0818674636. URL: http://web.archive.org/web/20210506213016/http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.8845&rep=rep1&type=pdf.

[HKM96] Gerd G. Hillebrand, Paris C. Kanellakis, and Harry G. Mairson. "Database Query Languages Embedded in the Typed Lambda Calculus". In: *Information and Computation* 127.2 (1996), pp. 117–144. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1996.0055. URL: https://www.sciencedirect.com/science/article/pii/S0890540196900553.

[HN11] Sylvain Heraud and David Nowak. "A Formalization of Polytime Functions". In: *Interactive Theorem Proving*. Ed. by Marko van Eekelen et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 119–134. ISBN: 978-3-642-22863-6.

[Hof06] Martin Hofmann. *Programming Languages for Logarithmic Space*. Presentation at GEOCAL '06, Spitalfields Day, Luminy. Geometric Complexity and Language (GEOCAL) Workshop. Feb. 13, 2006. URL: https://web.archive.org/web/20240714044259/https://www-lipn.univ-paris13.fr/~baillot/GEOCAL06/SLIDES/Hofmann1302.pdf.

[HP93] Petr Hájek and Pavel Pudlák. *Metamathematics of First-Order Arithmetic*. 1st ed. Perspectives in Mathematical Logic. Softcover reprint published 17 March 1998. Part of the Springer Book Archive. Berlin, Heidelberg: Springer-Verlag, 1993, pp. XIV+460. ISBN: 978-3-540-63648-9. URL: https://projecteuclid.org/eBooks/perspectives-in-logic/Metamathematics-of-First-Order-Arithmetic/toc/pl/1235421926.

[htt17a] Auberon (https://cs.stackexchange.com/users/26862/auberon). *Function problems and $FP \subseteq FNP$*. Computer Science Stack Exchange. Mar. 2017. eprint: https://cs.stackexchange.com/q/71617. URL: https://cs.stackexchange.com/q/71617.

[htt17b] Joshua Grochow (https://cstheory.stackexchange.com/users/129/joshua-grochow). *What exactly are the classes FP, FNP and TFNP?* Theoretical Computer Science Stack Exchange. Mar. 2017. eprint: https://cstheory.stackexchange.com/q/37813. URL: https://cstheory.stackexchange.com/q/37813.

[htt17c] Kaveh (https://mathoverflow.net/users/7507/kaveh). *Is there a syntactic characterization for BPP, BQP, or QMA?* MathOverflow. Apr. 2017. eprint: https://mathoverflow.net/q/35236. URL: https://mathoverflow.net/q/35236.

[htt20] Auberon (https://cstheory.stackexchange.com/users/37790/auberon). *What exactly are the classes FP, FNP and TFNP?* Theoretical Computer Science Stack Exchange. June 2020. eprint: https://cstheory.stackexchange.com/q/37812. URL: https://cstheory.stackexchange.com/q/37812.

[Imm86] Neil Immerman. "Relational queries computable in polynomial time". In: *Information and Control* 68.1 (1986), pp. 86–104. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(86)80029-8. URL: https://www.sciencedirect.com/science/article/pii/S0019995886800298.

[Imm87] Neil Immerman. "Languages that Capture Complexity Classes". In: *SIAM Journal on Computing* 16.4 (1987), pp. 760–778. DOI: 10.1137/0216051. eprint: https://doi.org/10.1137/0216051. URL: https://doi.org/10.1137/0216051.

[Imm99] Neil Immerman. *Descriptive Complexity*. Springer New York, NY, 1999, p. 268. ISBN: 978-1-4612-0539-5. DOI: 10.1007/978-1-4612-0539-5. URL: https://doi.org/10.1007/978-1-4612-0539-5.

[Jeř22] Emil Jeřábek. "Iterated Multiplication in VTC^0". In: *Archive for Mathematical Logic* 61.5 (July 2022), pp. 705–767. ISSN: 1432-0665. DOI: 10.1007/s00153-021-00810-6. URL: https://doi.org/10.1007/s00153-021-00810-6.

[Joh91] David S. Johnson. "A catalog of complexity classes". In: *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. Cambridge, MA, USA: MIT Press, 1991, pp. 67–161. ISBN: 0444880712. URL: http://web.archive.org/web/20231202034313/https://www.dbai.tuwien.ac.at/staff/pichler/complexity/johnson1990.pdf.

[Jon93] Neil D. Jones. "Constant time factors do matter". In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '93. San Diego, California, USA: Association for Computing Machinery, 1993, pp. 602–611. ISBN: 0897915917. DOI: 10.1145/167088.167244. URL: https://dl.acm.org/doi/pdf/10.1145/167088.167244.

[Jon99] Neil D. Jones. "LOGSPACE and PTIME characterized by programming languages". In: *Theoretical Computer Science* 228.1 (1999), pp. 151–174. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(98)00357-0. URL: https://www.sciencedirect.com/science/article/pii/S0304397598003570.

[JTS12] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. "Extending Type Theory with Forcing". In: *LICS 2012 : Logic In Computer Science*. Dubrovnik, Croatia, June 2012. URL: https://hal.science/hal-00685150.

[Jum95] Marc Jumelet. "Euler's φ-function in the context of $I\Delta_0$". In: *Archive for Mathematical Logic* 34.3 (June 1995), pp. 197–209. ISSN: 1432-0665. DOI: 10.1007/BF01375521. URL: http://web.archive.org/web/20240910062630/https://dspace.library.uu.nl/bitstream/handle/1874/27697/preprint108.pdf?sequence=1&isAllowed=y.

[KN04] L. Kristiansen and K.-H. Niggl. "On the computational complexity of imperative programming languages". In: *Theoretical Computer Science* 318.1 (2004). Implicit Computational Complexity, pp. 139–161. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2003.10.016. URL: https://www.sciencedirect.com/science/article/pii/S0304397503005218.

[Koz06] Dexter C. Kozen. "The Circuit Value Problem". In: *Theory of Computation*. London: Springer London, 2006, pp. 30–34. ISBN: 978-1-84628-477-9. DOI: 10.1007/1-84628-477-5_6. URL: https://doi.org/10.1007/1-84628-477-5_6.

[Kre88] Mark W. Krentel. "The complexity of optimization problems". In: *Journal of Computer and System Sciences* 36.3 (1988), pp. 490–509. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(88)90039-6. URL: https://www.sciencedirect.com/science/article/pii/0022000088900396.

[Kri05] Lars Kristiansen. "Neat function algebraic characterizations of logspace and linspace". In: *Computational Complexity* 14.1 (Apr. 2005), pp. 72–88. ISSN: 1420-8954. DOI: 10.1007/s00037-005-0191-0. URL: https://doi.org/10.1007/s00037-005-0191-0.

[Kud96] Manfred Kudlek. "Small deterministic Turing machines". In: *Theoretical Computer Science* 168.2 (1996), pp. 241–255. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(96)00078-3. URL: https://www.sciencedirect.com/science/article/pii/S0304397596000783.

[Kum+14] Ramana Kumar et al. "CakeML: A Verified Implementation of ML". In: *Principles of Programming Languages (POPL)*. ACM Press, Jan. 2014, pp. 179–191. DOI: 10.1145/2535838.2535841. URL: https://cakeml.org/popl14.pdf.

[KV05] Lars Kristiansen and Paul J. Voda. "Programming languages capturing complexity classes". In: *Nordic J. of Computing* 12.2 (Apr. 2005), pp. 89–115. ISSN: 1236-6064. URL: https://www.researchgate.net/publication/220673222_Programming_Languages_Capturing_Complexity_Classes.

[Lad75] Richard E. Ladner. "The circuit value problem is log space complete for P". In: *SIGACT News* 7.1 (Jan. 1975), pp. 18–20. ISSN: 0163-5700. DOI: 10.1145/990518.990519. URL: https://doi.org/10.1145/990518.990519.

[Lee91] Jan van Leeuwen, ed. *Handbook of theoretical computer science (vol. A): algorithms and complexity*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0444880712.

[Leh+23] Nico Lehmann et al. “Flux: Liquid Types for Rust”. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591283. URL: https://doi.org/10.1145/3591283.

[Lei10] K. Rustan M. Leino. “Dafny: an automatic program verifier for functional correctness”. In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370. ISBN: 3642175104.

[Lei91] D. Leivant. “A foundational delineation of computational feasibility”. In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. 1991, pp. 2–11. DOI: 10.1109/LICS.1991.151625.

[Ler09] Xavier Leroy. “A formally verified compiler back-end”. In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446. URL: http://xavierleroy.org/publi/compcert-backend.pdf.

[Li25] Jiatu Li. *An Introduction to Feasible Mathematics and Bounded Arithmetic for Computer Scientists*. Tech. rep. TR25-086. ISSN 1433-8092. Electronic Colloquium on Computational Complexity (ECCC), Report No. 86, July 2025. URL: https://eccc.weizmann.ac.il/report/2025/086/.

[Lin74] John C. Lind. *Computing in Logarithmic Space*. Technical Memo LCS-TM-052. Available from MIT DSpace. Massachusetts Institute of Technology, Project MAC, Sept. 1974. URL: https://dspace.mit.edu/handle/1721.1/148880.

[LM73] John Lind and Albert R. Meyer. “A characterization of log-space computable functions”. In: *SIGACT News* 5.3 (July 1973), pp. 26–29. ISSN: 0163-5700. DOI: 10.1145/1008293.1008295. URL: https://doi.org/10.1145/1008293.1008295.

[LM93] Daniel Leivant and Jean-Yves Marion. “Lambda calculus characterizations of poly-time”. In: *Fundamenta Informaticae, Special Issue on Feasible Mathematics*. Vol. 19. 1-2. 1993, pp. 167–180.

[LP99] Leslie Lamport and Lawrence C. Paulson. “Should your specification language be typed”. In: *ACM Trans. Program. Lang. Syst.* 21.3 (May 1999), pp. 502–526. ISSN: 0164-0925. DOI: 10.1145/319301.319317. URL: https://doi.org/10.1145/319301.319317.

[LR15] Daniel Leivant and Ramyaa Ramyaa. *The Computational Contents of Ramified Corecurrence*. Ed. by Andrew Pitts. Berlin, Heidelberg, 2015.

[LS10] Ugo Dal Lago and Ulrich Schöpp. “Functional Programming in Sublinear Space”. In: *European Symposium on Programming (ESOP 2010)*. Vol. 6012. LNCS. Springer, 2010, pp. 205–225. URL: http://web.archive.org/web/20240428171536/https://ulrichschoepp.de/Docs/esop10.pdf.

[LT12] Ugo Dal Lago and Paolo Parisen Toldin. *An Higher-Order Characterization of Probabilistic Polynomial Time (Long Version)*. 2012. arXiv: 1202.3317 [cs.LO]. URL: https://arxiv.org/abs/1202.3317.

[Mar06a] Simone Martini. “Implicit Computational Complexity, part 1”. In: *Bertinoro International Spring School for Graduate Studies in Computer Science*. Accessed: 26 August 2025. Bertinoro, Italy, Mar. 2006. URL: http://web.archive.org/web/20240722203715/https://www.cs.unibo.it/~martini/BISS/martini-1.pdf.

[Mar06b] Simone Martini. “Implicit Computational Complexity, part 2”. In: *Bertinoro International Spring School for Graduate Studies in Computer Science*. Accessed: 26 August 2025. Bertinoro, Italy, Mar. 2006. URL: http://web.archive.org/web/20240807183053/http://www.cs.unibo.it/~martini/BISS/martini-2.pdf.

[Mar06c] Simone Martini. “Implicit Computational Complexity, part 3”. In: *Bertinoro International Spring School for Graduate Studies in Computer Science*. Accessed: 26 August 2025. Bertinoro, Italy, Mar. 2006. URL: http://web.archive.org/web/20240416100620/http://www.cs.unibo.it/~martini/BISS/martini-3.pdf.

[Maz15] Damiano Mazza. “Simple Parsimonious Types and Logarithmic Space”. In: *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*. Ed. by Stephan Kreutzer. Vol. 41. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 24–40. ISBN: 978-3-939897-90-3. DOI: 10.4230/LIPIcs.CSL.2015.24. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2015.24.

[Maz18] Damiano Mazza. *Can typed lambda calculi express *all* algorithms below a given complexity?* Theoretical Computer Science Stack Exchange. Apr. 3, 2018. eprint: https://cstheory.stackexchange.com/q/27863. URL: https://cstheory.stackexchange.com/q/27863.

[MIS90] David A. Mix Barrington, Neil Immerman, and Howard Straubing. “On uniformity within NC1”. In: *Journal of Computer and System Sciences* 41.3 (1990), pp. 274–306. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(90)90022-D. URL: https://www.sciencedirect.com/science/article/pii/002200009090022D.

[MM03] Peter Møller Neergaard and Harry G. Mairson. “How Light Is Safe Recursion? Compositional Translations Between Languages of Polynomial Time”. Unpublished manuscript, Brandeis University, https://www.researchgate.net/publication/239691888_How_Light_Is_Safe_Recursion_Compositional_Translations_Between_Languages_of_Polynomial_Time. 2003. URL: http://web.archive.org/web/20250411115047/https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a1670570dbe8ff1f464a826db9b79e0054108c35.

[MO00a] Andrzej S Murawski and CHL Ong. “Can safe recursion be interpreted in light logic”. In: *Second International Workshop on Implicit Computational Complexity*. 2000.

[MO00b] Andrzej S. Murawski and C.-H. Luke Ong. “Can Safe Recursion Be Interpreted in Light Logic?” In: *2nd International Workshop on Implicit Computational Complexity*. June 2000.

[MO04] A.S. Murawski and C.-H.L. Ong. “On an interpretation of safe recursion in light affine logic”. In: *Theoretical Computer Science* 318.1 (2004). Implicit Computational Complexity, pp. 197–223. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2003.10.017. URL: https://www.sciencedirect.com/science/article/pii/S030439750300522X.

[Møl04] Peter Møller Neergaard. BC_{ε}^{-} : *A recursion-theoretic characterization of LOGSPACE*. Technical Report. Preliminary version. Brandeis University, Mar. 2004. URL: http://web.archive.org/web/20250411114452/https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d00d19870bb43abb4b0098e382ddc7f54d2ddf48.

[Mor05] Tsuyoshi Morioka. "Logical approaches to the complexity of search problems: proof complexity, quantified propositional calculus, and bounded arithmetic". AAINR02726. PhD thesis. CAN, 2005. ISBN: 0494027266. URL: https://eccc.weizmann.ac.il/resources/pdf/morioka.pdf.

[MP19] Anca Muscholl and Gabriele Puppis. "The Many Facets of String Transducers". In: *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*. Ed. by Rolf Niedermeier and Christophe Paul. Vol. 126. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 2:1–2:21. ISBN: 978-3-95977-100-9. DOI: 10.4230/LIPIcs.STACS.2019.2. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2019.2.

[MP91] Nimrod Megiddo and Christos H. Papadimitriou. "On total functions, existence theorems and computational complexity". In: *Theoretical Computer Science* 81.2 (1991), pp. 317–324. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(91)90200-L. URL: https://www.sciencedirect.com/science/article/pii/030439759190200L.

[MR67] Albert R. Meyer and Dennis M. Ritchie. "The complexity of loop programs". In: *Proceedings of the 1967 22nd National Conference*. ACM '67. Washington, D.C., USA: Association for Computing Machinery, 1967, pp. 465–469. ISBN: 9781450374941. DOI: 10.1145/800196.806014. URL: https://doi.org/10.1145/800196.806014.

[NC12] Phuong Nguyen and Stephen Cook. "The Complexity of Proving the Discrete Jordan Curve Theorem". In: *ACM Trans. Comput. Logic* 13.1 (Jan. 2012). ISSN: 1529-3785. DOI: 10.1145/2071368.2071377. URL: https://doi.org/10.1145/2071368.2071377.

[Nee04] Peter Møller Neergaard. "A Functional Language for Logarithmic Space". In: *Programming Languages and Systems*. Ed. by Wei-Ngan Chin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 311–326. ISBN: 978-3-540-30477-7. URL: https://doi.org/10.1007/978-3-540-30477-7_21.

[Ngu08] Phuong Nguyen. "Bounded Reverse Mathematics". Electronic Colloquium on Computational Complexity (ECCC) Books, Lectures and Surveys. PhD thesis. Graduate Department of Computer Science, University of Toronto, 2008. URL: https://eccc.weizmann.ac.il/static/books/Bounded_Reverse_Mathematics/.

[NMS21] Jaroslav Nesetril, Patrice Ossona de Mendez, and Sebastian Siebertz. *Structural properties of the first-order transduction quasiorder*. 2021. arXiv: 2010.02607 [math.CO]. URL: https://arxiv.org/abs/2010.02607.

[NW10] Karl-Heinz Niggl and Henning Wunderlich. “Implicit characterizations of FPTIME and NC revisited”. In: *The Journal of Logic and Algebraic Programming* 79.1 (2010). Special Issue: Logic, Computability and Topology in Computer Science: A New Perspective for Old Disciplines, pp. 47–60. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2009.02.005. URL: https://www.sciencedirect.com/science/article/pii/S1567832609000113.

[OCo05] Russell O’Connor. “Essential Incompleteness of Arithmetic Verified by Coq”. In: *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 2005, pp. 245–260. ISBN: 9783540318200. DOI: 10.1007/11541868_16. URL: http://dx.doi.org/10.1007/11541868_16.

[Odi99] Piergiorgio Odifreddi. *Classical Recursion Theory, Volume II*. Vol. 143. Studies in Logic and the Foundations of Mathematics. Amsterdam: Elsevier, 1999, pp. xvi+949. ISBN: 044450205X. URL: https://web.archive.org/web/20240103230629/http://www.piergiorgioodifreddi.it/wp-content/uploads/2010/10/CRT2.pdf.

[Oit10] Isabel Oitavem. “Logspace without Bounds”. In: *Ways of Proof Theory*. Ed. by Ralf Schindler. Berlin, Boston: De Gruyter, 2010, pp. 355–362. ISBN: 9783110324907. DOI: 10.1515/9783110324907.355. URL: https://doi.org/10.1515/9783110324907.355 (visited on 08/27/2025).

[Pap94] Christos H. Papadimitriou. “On the complexity of the parity argument and other inefficient proofs of existence”. In: *Journal of Computer and System Sciences* 48.3 (1994), pp. 498–532. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(05)80063-7. URL: https://www.sciencedirect.com/science/article/pii/S0022000005800637.

[Pau00] Lawrence C. Paulson. *Set Theory for Verification: II. Induction and Recursion*. 2000. arXiv: cs/9511102 [cs.LO]. URL: https://arxiv.org/abs/cs/9511102.

[PKL22] Jacob Prinz, G. A. Kavvos, and Leonidas Lampropoulos. “Deeper Shallow Embeddings”. In: *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Ed. by June Andronick and Leonardo de Moura. Vol. 237. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 28:1–28:18. ISBN: 978-3-95977-252-5. DOI: 10.4230/LIPIcs.ITP.2022.28. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2022.28.

[Rei08] Omer Reingold. “Undirected connectivity in log-space”. In: *J. ACM* 55.4 (Sept. 2008). ISSN: 0004-5411. DOI: 10.1145/1391289.1391291. URL: https://doi.org/10.1145/1391289.1391291.

[Ric07] Elaine Rich. *Automata, Computability and Complexity: Theory and Applications*. Prentice-Hall, Sept. 2007. ISBN: 978-0132288064. URL: http://web.archive.org/web/20240407185804/https://www.cs.utexas.edu/~ear/cs341/automatabook/AutomataTheoryBook.pdf.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. “Liquid types”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 159–169. ISBN: 9781595938602. DOI: 10.1145/1375581.1375602. URL: https://web.archive.org/web/20250723043647/https://goto.ucsd.edu/~rjhala/liquid/liquid_types.pdf.

[RL11] Ramyaa and Daniel Leivant. “Ramified corecurrence and logspace”. In: *Electronic Notes in Theoretical Computer Science* 276 (2011), pp. 247–261.

[Roc19] Simona Ronchi Della Rocca. “Logic and Implicit Computational Complexity”. In: *12th Panhellenic Logic Symposium*. Emerita Professor. Anogeia, Crete, Greece, June 2019. URL: http://web.archive.org/web/20240711151956/http://panhellenic-logic-symposium.org/12/slides/Day1_Ronchi.pdf (visited on 08/26/2025).

[Rog96] Yurii Rogozhin. “Small universal Turing machines”. In: *Theoretical Computer Science* 168.2 (1996), pp. 215–240. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(96)00077-1. URL: https://www.sciencedirect.com/science/article/pii/S0304397596000771.

[Sch05] Helmut Schmid. “A Programming Language for Finite State Transducers”. In: *Finite-State Methods and Natural Language Processing, 5th International Workshop, FSMNLP 2005, Helsinki, Finland, September 1-2, 2005. Revised Papers*. Ed. by Anssi Yli-Jyrä, Lauri Karttunen, and Juhani Karhumäki. Vol. 4002. Lecture Notes in Computer Science. Springer, 2005, pp. 308–309. DOI: 10.1007/11780885_38. URL: https://doi.org/10.1007/11780885_38.

[Sch06a] Ulrich Schöpp. *Space-efficiency and the Geometry of Interaction*. Presentation at GEOCAL ’06, Luminy. Part of the GEOCAL workshop series. Feb. 16, 2006. URL: https://web.archive.org/web/20240507113929/https://www-lipn.univ-paris13.fr/~baillot/GEOCAL06/SLIDES/Schoepp.pdf.

[Sch06b] Ulrich Schöpp. “Space-Efficient Computation by Interaction”. In: *Computer Science Logic*. Ed. by Zoltán Ésik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 606–621. ISBN: 978-3-540-45459-5. URL: http://web.archive.org/web/20240428171601/https://ulrichschoepp.de/Docs/seci.pdf.

[Sch07] Ulrich Schopp. “Stratified Bounded Affine Logic for Logarithmic Space”. In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 2007, pp. 411–420. DOI: 10.1109/LICS.2007.45. URL: https://ulrichschoepp.de/Docs/sbal_conf.pdf.

[Sea+25] Remy Seassau et al. “Formal Semantics and Program Logics for a Fragment of OCaml”. In: *Proc. ACM Program. Lang.* 9.ICFP (Aug. 2025). DOI: 10.1145/3747509. URL: https://doi.org/10.1145/3747509.

[Sel94] Alan L. Selman. “A taxonomy of complexity classes of functions”. In: *Journal of Computer and System Sciences* 48.2 (1994), pp. 357–381. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(05)80009-1. URL: https://www.sciencedirect.com/science/article/pii/S0022000005800091.

[Sim05] Harold Simmons. “Tiering as a Recursion Technique”. In: *The Bulletin of Symbolic Logic* 11.3 (2005), pp. 321–350. ISSN: 10798986. URL: http://www.jstor.org/stable/1578737 (visited on 11/29/2025).

[Sko23] Thoralf Skolem. “The foundations of elementary arithmetic established by means of the recursive mode of thought”. In: *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Ed. by Jean van Heijenoort. English translation from 1967 of Skolem’s 1923 paper. Cambridge, MA: Harvard University Press, 1923, pp. 302–333. ISBN: 9780674324497.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. USA: Elsevier Science Inc., 2006. ISBN: 0444520775.

[Sud77] I. H. Sudborough. "Some remarks on multihead automata". eng. In: *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications* 11.3 (1977), pp. 181–195. URL: http://eudml.org/doc/92050.

[Tao10] Terence Tao. *A Computational Perspective on Set Theory*. Originally published on the blog *What's new*. Archived at http://web.archive.org/web/20250822155936/https://terrytao.wordpress.com/2010/03/19/a-computational-perspective-on-set-theory/. Mar. 2010. URL: https://terrytao.wordpress.com/2010/03/19/a-computational-perspective-on-set-theory/ (visited on 08/22/2025).

[Tou22] George Tourlakis. *Computability*. 1st ed. Hardcover ISBN 978-3-030-83201-8 (pub. 2022-08-03); Softcover ISBN 978-3-030-83204-9 (pub. 2023-08-03); eBook ISBN 978-3-030-83202-5 (pub. 2022-08-02); Springer Nature Switzerland AG. Cham: Springer, 2022, pp. XXVII+637. ISBN: 978-3-030-83201-8. DOI: 10.1007/978-3-030-83202-5. URL: https://doi.org/10.1007/978-3-030-83202-5.

[Tra62] B. A. Trakhtenbrot. "Finite automata and monadic second order logic". Russian. In: *Siberian Mathematical Journal* 3 (1962). Russian original; English translation in: Amer. Math. Soc. Transl., Ser. 2, 59 (1966), 23–55, pp. 101–131.

[Tru24] Dafina Trufaş. "Intuitionistic Propositional Logic in Lean". In: *Electronic Proceedings in Theoretical Computer Science* 410 (Oct. 2024), pp. 133–149. ISSN: 2075-2180. DOI: 10.4204/eptcs.410.9. URL: http://dx.doi.org/10.4204/EPTCS.410.9.

[Var82] Moshe Y. Vardi. "The complexity of relational query languages (Extended Abstract)". In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. STOC '82. San Francisco, California, USA: Association for Computing Machinery, 1982, pp. 137–146. ISBN: 0897910702. DOI: 10.1145/800070.802186. URL: https://doi.org/10.1145/800070.802186.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Berlin, Heidelberg: Springer-Verlag, 1999. ISBN: 3540643109.

[Zak07] Mateusz Zakrzewski. *Definable functions in the simply typed lambda-calculus*. 2007. arXiv: cs/0701022 [cs.LO]. URL: https://arxiv.org/abs/cs/0701022.

[Zam96] Domenico Zambella. "Notes on Polynomially Bounded Arithmetic". In: *The Journal of Symbolic Logic* 61.3 (1996), pp. 942–966. ISSN: 00224812. URL: http://www.jstor.org/stable/2275794 (visited on 11/23/2025).