SAARLAND UNIVERSITY

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

BACHELOR'S THESIS

TOWARDS A FORMAL PROOF OF THE COOK-LEVIN THEOREM

Author	**Supervisor**	**Advisor**
Lennard Gäher	Prof. Gert Smolka	Fabian Kunze

Reviewers
Prof. Gert Smolka
Prof. Markus Bläser

Submitted: April 9th, 2020

Eidesstattliche Erklärung

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

Statement in Lieu of an Oath

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

Einverständniserklärung

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

Declaration of Consent

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

Saarbrücken, April 9th, 2020

Abstract

In this thesis, we formally verify a proof of the Cook-Levin Theorem in the proof assistant Coq. The Cook-Levin Theorem states that the satisfiability problem SAT of conjunctive normal forms is NP-complete. This means that any language which is polynomial-time verifiable is reducible to SAT in polynomial time. Despite its importance, most proofs do not even attempt to verify the construction's correctness.

For the proof, one has to encode computations of a chosen reasonable computational model using Boolean formulas. We use the call-by-value λ-calculus L as the computational model as part of a larger effort to formalise the basics of complexity theory. It is much easier to write and verify L-programs than to construct Turing machines.

Thus, we need to reduce the computation of L-terms to SAT. As Turing machines are structurally simpler than the λ-calculus, we propose to use Turing machines as an intermediate problem in the reduction of L to SAT, performing the computation of a Turing machine from an L-term and the construction of a SAT-formula from a Turing machine in L.

In this work, we introduce the basics of formalising polynomial-time reductions in Coq. We present a polynomial-time reduction from Turing machines to SAT in L which is based on the original tableau construction by Cook and formally verify its correctness and resource usage. The original construction is adapted and factorised in order to make a formal proof feasible. We see this as a significant first step towards showing SAT to be NP-complete in L, leaving the reduction from L to Turing machines for future work.

Acknowledgements

Most importantly, I would like to thank Fabian for many fruitful discussions and ideas as well as the support he provided in our weekly meetings throughout the semester. I appreciate the valuable feedback he gave on various parts of this thesis.

I am very grateful to Professor Smolka for introducing me to computational logic and Coq through his ICL course and for allowing me to write this thesis at his chair. Moreover, I am thankful that he mentored and supported me for the past two years of my Bachelor's studies.

I am greatly indebted to my family for supporting me both emotionally and financially. Without them, neither I nor this thesis would exist. Additionally, I want to express my gratitude towards my friends for the time we spent together and for providing some distraction from work. Especially, I would like to thank Marcel and Timon for proof-reading this thesis.

Finally, I want to express my gratitude towards Yannick Forster for providing the LaTeX template for this thesis and Professor Bläser as well as Professor Smolka for reviewing it.

Contents

Chapter 1

Introduction

Since the advent of modern computational complexity theory in the 1960s and -70s, it has become one of the cornerstones of theoretical computer science. Important early results include the hierarchy theorems by Hartmanis and Stearns [15], essentially stating that one can compute more given more time or space. The next landmark result was the Cook-Levin Theorem, first proved by Stephen A. Cook in 1971 [6] and independently discovered by Leonid Levin in 1973 [21]. This result founded the important class of NP-complete problems.

NP is the class of all problems which are polynomial-time verifiable, i.e. for which it is efficiently decidable if a given certificate correctly proves that an element is a yes-instance of the problem. For two problems **P** and **Q**, **P** is polynomial-time reducible to **Q** if there is a function f transforming instances of **P** into instances of **Q** in polynomial time such that for any instance p of **P**, $p \in \mathbf{P}$ does hold if and only if $f(p) \in \mathbf{Q}$. If **Q** is contained in NP, then **P** is also in NP. A problem is NP-hard if any problem contained in NP can be reduced to it in polynomial time. An NP-complete problem is NP-hard and itself contained in NP.

The Cook-Levin Theorem makes a statement about the satisfiability problem of conjunctive normal forms **SAT**: given a formula N in conjunctive normal form, does there exist an assignment α to its variables such that the formula evaluates to true, i.e. $\alpha \models N$? According to the Cook-Levin Theorem, **SAT** is NP-complete.

The importance of this result was underlined by the subsequent discovery of 21 more NP-complete problems by Richard Karp in 1972 [18], whose NP-hardness was established by reductions from **SAT** (using transitivity). This showed that there are in fact many more problems which are interesting in practice and NP-complete. Thus, the important contribution of the Cook-Levin Theorem is not that specifically **SAT** is NP-complete, but that there exists a natural[1] NP-complete problem. If one

[1] We call a problem natural if it is independent of a model of computation and relevant in practice. This is not a precise definition, but captures the intuition.

has a natural problem such as **SAT** which has been shown to be NP-complete, it is relatively easy to show other important problems to be NP-complete.

Nowadays, the question whether P = NP, i.e. whether every problem which is verifiable in polynomial time is also decidable in polynomial time, is one of the biggest open questions of computer science [23]. If one were to find a polynomial-time decider for an NP-hard problem such as **SAT**, one could directly answer "P= NP!".

Formal Complexity Theory All of the results mentioned above are usually taught in undergraduate courses on theoretical computer science, but the proofs are very handwavy and omit many interesting details. From a mathematical perspective, this is not satisfying. Despite the huge importance of these results, no completely formal proof of any of them has been published[2]. There have been first steps towards formalising complexity-theoretic results by Asperti [3, 2], but they have been without reference to a concrete computational model. We will comment on this subject in Chapter 9.

While there is a mechanised proof verifying the translation of Turing machines to SAT formulas in the theorem prover ACL2 available [14], it does not include a running time analysis with respect to a computational model which has been shown to be reasonable[3]. Therefore, this proof does not show the NP-hardness of SAT, but only a part (although a significant one) of that.

One of the reasons no full formalisation is available is that some of the details are very tedious. The prevalent computational model in complexity theory are Turing machines, which have a pleasant time and space usage behaviour as they can only modify a constant amount of data in a single computational step, but are otherwise very low-level and non-compositional [10]. Doing formal complexity theory using Turing machines thus seems to be a daunting task: as stated by Forster et al. [10], "Turing machines as model of computation are inherently infeasible for the formalisation of any computability- or complexity-theoretic result".

In contrast, results from computability theory have been successfully formalised in the proof assistant Coq [28]. Coq employs a constructive type theory based on the calculus of inductive constructions [27]. The key behind this success is that Coq only allows to define computable functions. It is thus unnecessary to employ an external model of computation when formalising computability theory in Coq, an approach known as *synthetic computability theory* [7]. Nowadays, a large library of undecidability results is available [12].

When formalising complexity-theoretic results, one cannot make use of this trick: in

[2] An unpublished formalisation of the Time Hierarchy Theorem using the same λ-calculus in Coq we are using is known to the author.

[3] in the sense that the induced complexity classes agree with the ones for Turing machines

complexity theory, not only the computability of functions is relevant, but also their time and space usage. For doing basic complexity theory, one at least needs to be able to state that a function runs in polynomial time. Especially for metaresults like the Hierarchy Theorems and the Cook-Levin Theorem, a concrete computational model seems to be unavoidable. However, instead of Turing machines, we use the call-by-value λ-calculus L [11], as proposed by Forster et al. in [10].

L is still low-level, but much closer to real functional programming languages than Turing machines are. For instance, inductive datatypes like the natural numbers or lists can be systematically encoded in L. Using L in Coq is additionally significantly eased by a certifying extraction mechanism [8]. It allows one to define functions and datatypes using the usual tools of Coq and semi-automatically derive equivalent L-terms together with certificates of their correctness. This mechanism can also be used to derive time bounds: during the extraction of functions, recurrence relations describing the running-time are generated automatically. These have to be solved by the user to obtain an explicit description of the running time.

The theoretical foundation of using L for the formalisation of complexity theory has been laid by Forster et al. in [9]. There it is shown that Turing machines and L can simulate each other with a polynomial overhead in time and a constant-factor overhead in space for decision problems[4], if one chooses the right resource measures for L. This result is crucial as it proves that many basic complexity classes like NP or P do correspond for Turing machines and L; in particular, NP-completeness for L and NP-completeness for Turing machines are equivalent.

Outline of the Cook-Levin Theorem In the following, we are concerned with the Cook-Levin Theorem. In order to prove it, one has to show that **any** problem contained in NP is polynomial-time reducible to a natural problem such as **SAT**. As we have no information about the concrete problem we are reducing from, we have to resort to a proof using properties of the chosen model of computation. Specifically, arbitrary computations have to be encoded using Boolean formulas. As we are using L as our computational model, it is necessary to encode arbitrary reduction chains of L.

The advantage of the greater abstraction provided by L compared to Turing machines now has its cost: it does not seem easy to directly encode L using Boolean formulas. In fact, there is evidence that, even when reducing to another natural problem than **SAT**, a direct reduction would still be difficult. We will elaborate on this in Remark 2.21. Thus, we propose to use Turing machines as an intermediate step for deriving a natural NP-complete problem for L: one first reduces the computation of L to Turing machines and then reduces Turing machines to **SAT**. Although one uses Turing machines as an intermediate problem, the reductions

[4]However, the result does not cover sublinear space or time.

still use L as the computational model.

Outline of this Thesis In this thesis, we formalise the reduction from Turing machines to **SAT**. We base the formalisation on a textbook proof by Sipser [24], which in turn is very similar to the original construction by Cook [6]. Our proof proceeds by encoding a bounded number of computational steps of a Turing machine in a tableau of a bounded size. Each row of the tableau represents one of the Turing machine's configurations with each element of the row corresponding to one symbol of a Turing machine tape. The tableau can be encoded using a number of Boolean variables and the conjunctive normal form then forces that the individual rows of the tableau form valid configuration changes.

In order to make the formalisation feasible, we factorise the proof into several intermediate problems and reductions. The key idea is to first reduce to a string-based problem **PR** that shares characteristics of both Turing machines and circuits. The reduction essentially generates an explicit representation of a Turing machine from a symbolic one. We then incrementally deal with encoding **PR** as a conjunctive normal form. First, we reduce to a binary alphabet using a substituting string homomorphism. **PR** over a binary alphabet can easily be encoded using a Boolean formula. In order to bring this formula into conjunctive normal form, we employ the Tseytin transformation [31].

In contrast to the existing ACL2 mechanisation [14], we include running time analyses in L, which is a reasonable computational model.

In Chapter 2, we introduce the needed preliminaries, among them the definition of Turing machines, L and the basic definitions one needs for complexity theory. We also elaborate on the specific changes one has to make to the well-known definitions of complexity on Turing machines. The basic techniques for doing polynomial-time reductions are explored in Chapter 3 on a simpler reduction from k-**SAT** to **Clique**. This is also the only chapter where we go into the details of the running time analyses. We give an informal outline of the chain of reductions from Turing machines to **SAT** in Chapter 4. Chapters 5 to 8 then give the details of the individual reductions.

Mechanisation All results presented from Chapter 3 onwards have been mechanised in the proof assistant Coq [28]. The definitions, lemmas and theorems are hyperlinked to a version of the development viewable in a webbrowser. The mechanisation of some of the reductions differs in small but notable ways to the presentation on paper, mainly for technical reasons. Therefore, the chapters on reductions usually contain a section outlining the differences and the motivation behind them. Readers not familiar with Coq may want to skip these sections. While the running times of the reductions have been verified in Coq, we do not go into these details on paper, with Chapter 3 being an exception. We give an overview on the full structure

of the mechanisation in Appendix A.

Contributions The proof of the Cook-Levin Theorem as presented by Sipser [24] is adapted in order to make a formalisation feasible. To that end the construction is changed and a new string-based intermediate problem is introduced. The whole construction including the polynomial running time is formalised in Coq.

Chapter 2

Preliminaries

In this chapter, we present the needed basic definitions, including notations, Turing machines and the call-by-value λ-calculus L.

2.1 Type Theory

We formalise the results in the constructive type theory of Coq [28], featuring inductive types and an impredicative universe of propositions $\mathbb{P}$. In this section we introduce the notations and concepts common in type theory. Readers not familiar with type theory may informally regard types as sets.

$\mathbb{B}$ is the type of Booleans with the two elements T and F. Natural numbers are accomodated by the inductively defined type $\mathbb{N}$ featuring the two constructors $\mathsf{O} : \mathbb{N}$ and $\mathsf{S} : \mathbb{N} \to \mathbb{N}$ giving the successor of a number. We use the common operations on natural numbers.

The type of options $\mathbb{O}(X)$ over X consists of the element $\emptyset$, denoting the absence of a value, and elements $^\circ x$ for $x : X$.

We write $\mathcal{L}(X)$ for the type of lists over X. Lists are constructed inductively using the cons constructor starting from the empty list $[\,]$: for an element $x : X$ and a list $A : \mathcal{L}(X)$, $x :: A$ is the list A prepended with x. For an arbitrary list A, $|A|$ is the length of A. The concatenation of two lists A and B is written as $A +\!\!+ B$. We use positions to refer to the elements of a list at a certain offset, starting at 0. The valid positions of a list A are the numbers $0, \ldots, |A| - 1$. Given a position i, the element of A at i is denoted by $A[i]$. Formally, this is an option value: if i is not a valid position, $A[i]$ is defined to be $\emptyset$. The list $A[i..]$ is the sublist of A containing all elements from position i onwards (and potentially no elements if $|A| \leqslant i$). Similarly, the list $A[..i)$ contains all elements up to (but excluding) position i. By a^n, we denote the list consisting of the n-fold repetition of the element a. Often, we need to apply a function $f : X \to Y$ to every element of a list $A : \mathcal{L}(X)$. We write $[\, f\, x \mid x \in A \,]$ for this list. Note that, in contrast to the use of this notation in set theory, the order of the list's elements

is preserved. Similarly, we use $[\ x \mid x \in A \wedge p\ x\ ]$, where $p : X \to \mathbb{B}$, for the list A filtered to contain only the elements for which the predicate p holds. Given two lists $A, B : \mathcal{L}(X)$, $A \subseteq B$ denotes that all elements of A are also contained in B (possibly with duplicates or in a diffferent order), i.e. $A \subseteq B \coloneqq \forall x, x \in A \to x \in B$. We use the synonym A^* to refer to $\mathcal{L}(A)$ in the context of strings.

The type of vectors of length n over X is written X^n. For length and subscripting, we use the same notations as for lists.

The product type $X \times Y$ of the types X and Y consists of pairs of elements of X and Y. The pair of $x : X$ and $y : Y$ is written as (x, y), while we use π_1, π_2 for the projections of a pair to its first and second component.

The sum type $X + Y$ of the types X and Y consists of the elements of X and the elements of Y. Formally, we have two injections $L : X \to X + Y$ and $R : Y \to X + Y$.

Sigma types $\mathbf{\Sigma}_x.p\ x$ allow us to define pairs where the type of the second component depends on the first component (therefore its inhabitants are also called *dependent pairs*). We write (x, s) for the dependent pair consisting of x and $s : p\ x$.

A type X is called *discrete* if equality on it is decidable, that is, there exists a function $\mathsf{eq}_X : X \to X \to \mathbb{B}$ such that $\mathsf{eq}_X\ a\ b = T$ if and only if $a = b$. We also write $a \stackrel{?}{=} b$ instead of $\mathsf{eq}_X\ a\ b$, omitting the type which can be inferred from the context. We use the type eqType to refer to those types which are discrete.

More generally, we extend the notation $a \stackrel{?}{=} b$ to other decidable binary predicates, for instance to $a \stackrel{?}{\leqslant} b$ for deciding the relation $\leqslant$.

We also need *finite types*. Finite types are discrete types with a finite number of elements. Formally, we require a list of all values of the type, in which each element occurs exactly once, as the witness that it is finite [22]. Given a finite type F and $e : F$, $\mathsf{index}\ e$ is the position of e in this list. $|F|$ is the cardinality of F. We refer to finite types using the type $\mathsf{finType}$. For any number n, there is a type F_n with exactly n elements $f_0, \ldots, f_{n-1}$. Finite types are closed under the type constructors $\mathbb{O}(\cdot)$, $\cdot + \cdot$ and $\cdot \times \cdot$. For instance, if A and B are finite types, then also $A \times B$ is a finite type.

Relations As is common in type theory, we model relations on a type X using binary predicates of type $X \to X \to \mathbb{P}$. For a relation R, we write $(a, b) \in R$ or $a\ R\ b$ for $R\ a\ b$. A relation R is included in another relation S, denoted $R \subseteq S$, if $\forall a\ b, (a, b) \in R \to (a, b) \in S$. R and S are equivalent if they mutually include each other: $R \equiv S \coloneqq R \subseteq S \wedge S \subseteq R$.

The n-th power R^n of a relation R is defined inductively:

$$\frac{}{R^0\ x\ x} \qquad \frac{R\ x\ y \quad R^n\ y\ z}{R^{1+n}\ x\ z}$$

R^n

For some proofs, it will be convenient to have an alternative definition where the successor case appends a new transition instead of prepending it:

$$\frac{}{{}^{0}\mathrm{R}\ x\ x} \qquad \frac{{}^{n}\mathrm{R}\ x\ y \quad \mathrm{R}\ y\ z}{{}^{1+n}\mathrm{R}\ x\ z}$$

$\boxed{{}^{n}\mathrm{R}}$

Proposition 2.1 *We have the following basic facts:*

1. *Transitivity:* $\mathrm{R}^n\ x\ y \to \mathrm{R}^m\ y\ z \to \mathrm{R}^{n+m}\ x\ z$
2. *Monotonicity:* $\mathrm{R} \subseteq \mathrm{S} \to \mathrm{R}^n\ x\ y \to \mathrm{S}^n\ x\ y$
3. *Congruence:* $\mathrm{R} \equiv \mathrm{S} \to \mathrm{R}^n\ x\ y \leftrightarrow \mathrm{S}^n\ x\ y$
4. *Agreement:* $\mathrm{R}^n\ x\ y \leftrightarrow {}^{n}\mathrm{R}\ x\ y$
5. *Additivity:* $\mathrm{R}^{n+m}\ x\ z \leftrightarrow \exists y, \mathrm{R}^n\ x\ y \wedge \mathrm{R}^m\ y\ z$

2.2 Turing Machines

In this section, we present the formalisation of deterministic Turing machines used throughout the thesis.

Turing machines can be regarded as finite automata with access to a fixed number of infinite tapes. Each tape has a head which can be moved sequentially. For every computational step, the Turing machine reads the content of the cells currently under the heads. It then transitions to a new state and can optionally write a symbol on each of the tapes, before potentially moving the heads one position to the left or to the right.

The following definitions are due to Asperti and Ricciotti [4]; for the Coq formalisation, we use the Turing machine framework [10].

Tapes We define two-sided infinite tapes over a finite type Σ, the tape alphabet. In contrast to usual presentations, Σ does not contain a special blank symbol that denotes unused regions of the tape. Instead, the definition only captures the finite region of the tape that is currently in use. This formalisation of tapes without blanks has the advantage that each possible tape is uniquely represented by exactly one element of tape_Σ.

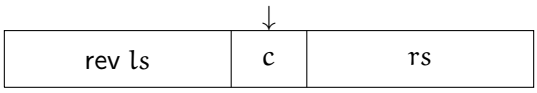
A tape can be in one of four states:

$\boxed{\mathsf{tape}_\Sigma}$

$$\begin{aligned}\mathsf{tape}_\Sigma :=\ & \\ & \mid \mathsf{niltape} \\ & \mid \mathsf{leftof}\ (c : \Sigma)\ (rs : \mathcal{L}(\Sigma)) \\ & \mid \mathsf{rightof}\ (c : \Sigma)\ (ls : \mathcal{L}(\Sigma)) \\ & \mid \mathsf{midtape}\ (ls : \mathcal{L}(\Sigma))\ (c : \Sigma)\ (rs : \mathcal{L}(\Sigma))\end{aligned}$$

A niltape is completely empty. In all other cases, the tape contains at least one symbol c. In the case of leftof c rs, the list of symbols c :: rs contains exactly the tape contents right of the head, and conversely, in the case of rightof c ls, the list c :: ls contains the tape contents left of the head. For these two cases, the tape does not currently reside on a symbol. Finally, midtape ls c rs models the case that the head resides on the symbol c and there are (possibly empty) parts of the tape ls and rs to the left and to the right.

The tapes are always interpreted such that the heads of the lists are closest to the Turing machine's head. If one were to imagine a tape as a linear sequence of symbols, midtape ls c rs would have the following shape:



We use the functions left, right : $\mathsf{tape}_\Sigma \to \mathcal{L}(\Sigma)$ and current : $\mathsf{tape}_\Sigma \to \mathbb{O}(\Sigma)$ to extract the contents of the tape left of the head, right of the head, or under the head.

Turing machines In each computational step, a Turing machine can optionally write a symbol and move the head on each of its tapes individually. These actions are captured by the type $\mathsf{Act}_\Sigma \coloneqq \mathbb{O}(\Sigma) \times \mathsf{move}$, where Σ is the tape alphabet and $\mathsf{move} := \mathsf{L} \mid \mathsf{R} \mid \mathsf{N}$ defines the possible movements.

Act_Σ

move

Definition 2.2 (Turing machines) *Given a finite type Σ and a number of tapes n, Turing machines of type $\mathit{mTM}\ \Sigma\ n$ are tuples $(Q, \delta, \mathit{start}, \mathit{halt})$, where Q is the finite type of states, $\delta : Q \times (\mathbb{O}(\Sigma))^n \to Q \times \mathit{Act}_\Sigma{}^n$ is the transition function, start is the initial state and $\mathit{halt} : Q \to \mathbb{B}$ defines the halting states.*

$\mathit{mTM}\ \Sigma\ n$

For the semantics of Turing machines, the values of the transition function for halting states, i.e. states q for which halt q = T, are irrelevant.

A configuration of a Turing machine M is a pair consisting of the current state and its tapes.

Definition 2.3 (Configurations) *Let $M : \mathit{mTM}\ \Sigma\ n$. The type of configurations over M is given by $\mathit{conf}_M \coloneqq Q_M \times (\mathit{tape}_\Sigma)^n$.*

conf_M

We give the full definition of the semantics of Turing machines in Appendix C and only define formally here how the Turing machine moves its heads (Figure 2.1). The rest of the definitions follows what one would intuitively expect.

In the presentation on paper, it will suffice to assume a transition relation $\succ$ on configurations such that $(q, tp) \succ (q', tp')$ holds if and only if halt q = F and (q', tp') is the successor configuration of (q, tp) according to the transition function.

$c \succ c'$

$$
\begin{aligned}
\mathsf{tape\_move}\ \mathsf{L}\ (\mathsf{leftof}\ r\ rs) &\coloneqq \mathsf{leftof}\ r\ rs \\
\mathsf{tape\_move}\ \mathsf{R}\ (\mathsf{leftof}\ r\ rs) &\coloneqq \mathsf{midtape}\ [\,]\ r\ rs \\
\mathsf{tape\_move}\ \mathsf{L}\ (\mathsf{midtape}\ [\,]\ c\ rs) &\coloneqq \mathsf{leftof}\ c\ rs \\
\mathsf{tape\_move}\ \mathsf{R}\ (\mathsf{midtape}\ ls\ c\ [\,]) &\coloneqq \mathsf{rightof}\ c\ ls \\
\mathsf{tape\_move}\ \mathsf{L}\ (\mathsf{midtape}\ (l :: ls)\ c\ rs) &\coloneqq \mathsf{midtape}\ ls\ l\ (c :: rs) \\
\mathsf{tape\_move}\ \mathsf{R}\ (\mathsf{midtape}\ ls\ c\ (r :: rs)) &\coloneqq \mathsf{midtape}\ (c :: ls)\ r\ rs \\
\mathsf{tape\_move}\ \mathsf{L}\ (\mathsf{rightof}\ l\ ls) &\coloneqq \mathsf{midtape}\ ls\ l\ [\,] \\
\mathsf{tape\_move}\ \mathsf{R}\ (\mathsf{rightof}\ l\ ls) &\coloneqq \mathsf{rightof}\ l\ ls \\
\mathsf{tape\_move}\ \_\ \mathsf{niltape} &\coloneqq \mathsf{niltape} \\
\mathsf{tape\_move}\ \mathsf{N}\ tp &\coloneqq tp
\end{aligned}
$$

Figure 2.1: Turing machine tape movements. Note how the tape does not change if the Turing machine wants to move the head one more symbol beyond the used tape region if the head currently is not on a symbol. This means that it is not possible for the head to reside two or more symbols beyond the used tape region.

Definition 2.4 (Termination Relation)

$c \rhd^k c'$
$c \rhd^{\leqslant k} c'$

$$
\begin{aligned}
(q, tp) \rhd^k (q', tp') &\coloneqq (q, tp) \succ^k (q', tp') \wedge \mathit{halt}\ q' = \mathsf{T} \\
(q, tp) \rhd^{\leqslant k} (q', tp') &\coloneqq \exists l \leqslant k, (q, tp) \rhd^l (q', tp')
\end{aligned}
$$

We only need single-tape Turing machines throughout this thesis and therefore we restrict this notation to single-tape machines. The following result states that with each computational step, a Turing machine can take up at most one additional tape cell.

Lemma 2.5 ("Time Bounds Space")
Assume a Turing machine $M : \mathit{mTM}\ \Sigma\ 1$ *and* $(q, tp) \succ (q', tp')$. *Then* $\mathit{sizeOfTape}\ tp' \leqslant 1 + \mathit{sizeOfTape}\ tp$, *where* $\mathit{sizeOfTape} : \mathit{tape}_\Sigma \to \mathbb{N}$ *describes the number of symbols contained on a tape.*

2.3 The λ-calculus L

L is an untyped λ-calculus with weak call-by-value reduction and is the underlying computational model we are using throughout this thesis. This section intends to give a brief overview of the most foundational aspects of L in order to justify using L as a computational model. The rest of this thesis can, however, be read without delving into the details of L. For a more thorough treatment, the interested reader

is referred to [11], where L is introduced in detail in the context of computability theory.

L is defined over terms s, t, u : term $\coloneqq x \mid \lambda x.s \mid s\ t$[1], i.e. only λ-abstractions and applications are part of the core language. λ-expressions are called abstractions. A term s is closed if all its variables are bound. A closed abstraction is a procedure. term

2.3.1 Semantics of L

For the semantics, we start by defining the reduction relation. Let $s[x := t]$ be the term that is obtained by replacing every free occurrence of the variable x in s by the term t. L features a weak call-by-value reduction $\succ$. This means that reduction is not possible below binders (i.e. lambdas) and arguments need to be fully reduced to a value (a λ-abstraction) before β-reduction (i.e. function application) is possible.

$$\frac{s \succ s'}{s\ t \succ s'\ t} \quad \frac{t \succ t'}{s\ t \succ s\ t'} \quad \frac{}{(\lambda x.s)(\lambda y.t) \succ s[x := \lambda y.t]}$$

The last rule is the interesting one: only if both sides of an application have been fully evaluated to an abstraction can we do a β-reduction.

Note that the reduction relation is not deterministic as we have not specified an evaluation order for applications. This does not pose a problem, however, as L can be shown to be uniformly confluent:

Fact 2.6 (Fact 7 in [11]) *If $s \succ t_1$ and $s \succ t_2$, then either $t_1 = t_2$ or $t_1 \succ u$ and $t_2 \succ u$ for some u.*

We say that a term t is normal if it cannot be reduced further according to $\succ$. t is a normal form of s if $s \succ^* t$ and t is normal, where $\succ^*$ is the reflexive-transitive closure of $\succ$. Uniform confluence implies that normal forms are unique, if they exist. Moreover, every term s which has a normal form t always normalises to t in the same number of steps:

Corollary 2.7 (Uniform Normalisation, Fact 29 in [26]) *Assume that $s \succ^m t$ and $s \succ^n u$ and let t be normal. Then $n \leqslant m$ and $u \succ^{m-n} t$.*

Thus, the result of evaluation is deterministic, only the way to get there is nondeterministic.

2.3.2 Encoding of Inductive Datatypes and Recursive Functions

While L is very simple in nature and does not have built-in suppport for Booleans or natural numbers, for instance, one can easily encode inductive datatypes using procedures. One way to do this is to use Scott encodings. The Scott encodings

[1] Formally, De Bruijn indices are used instead of named binders.

of elements of an inductive datatype are procedures with one argument for each of the datatype's constructors. As an example, we look at the encoding of natural numbers, which feature two constructors $O : \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$. Their Scott encoding looks as follows:

$$\overline{O} \coloneqq \lambda a.\lambda b.a \qquad\qquad \overline{S\ n} \coloneqq \lambda a.\lambda b.b\ \overline{n}$$

The basic idea is that the encoding allows to easily match on an element of the datatype by passing it suitable arguments which will be used for the different cases. For instance, we have that for procedures s, t:

$$\overline{O}\ s\ t \succ^2 s \qquad\qquad \overline{S\ n}\ s\ t \succ^2 t\ \overline{n}$$

This encoding can be derived systematically for arbitrary datatypes such as lists. We use the notation $\overline{x}$ to denote the encoding of $x : X$ for an L-encodable type X[2]. Of course, having inductive datatypes is not of much use if one cannot define recursive functions on them. Luckily, there is a function with which recursive terms can be obtained:

Fact 2.8 (Fact 6 from [11]) *There is a function $\rho : \mathsf{term} \to \mathsf{term}$ such that (1) $\rho\ s$ is a procedure if s is closed and (2) $(\rho\ u)v \succ^3 u(\rho\ u)v$ for all procedures u, v.*

The procedure u can be seen as a step function taking the function to call for recursion as the first argument. With the help of the recursion operator ρ, one can define recursive functions on Scott encodings by directly translating the recursive equations one would use in a functional programming language.

This systematic encoding has been utilised by Forster and Kunze [8] to develop a certifying extraction mechanism which can automatically generate Scott encodings of inductive datatypes defined in Coq as well as encodings of recursive and non-recursive functions on these datatypes. The correctness of these encodings, in the sense that the encoded terms behave similarly to the original terms, is derived fully automatically. The mechanism thus allows one to program functions for L without having to directly work with L.

Remark 2.9 *Not every type definable in Coq can be extracted: types living in the impredicative universe of propositions $\mathbb{P}$ or consisting of propositional parts have no corresponding object in L and thus extraction fails on them. We will see examples of this in Section 3.2.*

[2] Usually, this means that there exists a Scott-encoding for X, although in some cases, one can also derive an encoding for types without a direct Scott encoding.

2.3.3 Time and Space Measures for L

As we want to do complexity theory in L, we have to define time and space measures for it. It is crucial that the induced cost models are reasonable with respect to the invariance thesis [25], in the sense that Turing machines and L can simulate each other with a polynomial overhead in time and a constant-factor overhead in space. For reasonable computational models, well-known classes like P, NP, and $\mathsf{LogSpace}$ are machine-independent.

While reasonable measures for Turing machines are quite intuitive, taking the number of steps as the time measure and the maximum number of cells used on a tape as the space measure, the picture is not as clear for the λ-calculus. Historically, a number of different measures have been explored for the variety of possible evaluation strategies [1].

The seemingly natural resource measures for the λ-calculus are the number of β-reduction steps for time and the maximum size of terms encountered during a reduction to a normal form for space, where we write $\|s\|$ for the size of a term s. However, these measures are a bit unintuitive: there exist terms that exhibit linear time but exponential space usage [9]. The reason is that, when doing substitution, terms get duplicated if an argument variable occurs multiple times. This problem is known as **size explosion**. A Turing machine implementing a substitution strategy will thus have an exponential time usage (by Lemma 2.5). Size explosion can be mitigated if one does not perform a naive substitution-based evaluation strategy, but instead uses environments that store the values of variables on a heap, thus avoiding duplication. Sadly, due to the needed pointers into the heap, this strategy does have a linear-factor space overhead on some terms, a problem known as **pointer explosion**.

$\boxed{\|s\|}$

Nevertheless, these resource measures have been shown to be reasonable for L by Forster et al. in [9], a non-trivial result. While simulating Turing machines in the λ-calculus within the desired overhead is relatively easy, the efficient simulation of L using Turing machines is difficult due to size explosion and pointer explosion. The authors solve this by interleaving substitution- and heap-based strategies. Their result is, however, limited to decision problems:

Theorem 2.10 (Theorem 2 of [9]) *Let Σ be a finite alphabet with $\{\mathsf{T},\mathsf{F}\} \subseteq \Sigma$ and let $f : \Sigma^* \to \{\mathsf{T},\mathsf{F}\}$ be a function. Let $t, s \in \Omega(n)$.*

- *If f is computable by L in time t and space s, then f is computable by a Turing machine in time $\mathcal{O}(\mathit{poly}(t(n)))$ and space $\mathcal{O}(s(n))$.*
- *If f is computable by a Turing machine in time t and space s, then f is computable by L in time $\mathcal{O}(\mathit{poly}(t(n)))$ and space $\mathcal{O}(s(n))$.*

This suffices for the complexity theory of decision problems for at least linear time and space usage; in particular, the result shows that one can expect to be able to define the classes P and NP in a way which agrees with the usual definitions for Turing machines. The intuition behind this result is that computations which exhibit the size explosion problem but terminate in T or F can be compressed to use only polynomial space by applying the preceding theorem twice.

Remark 2.11 *For the proof of the reasonability result, L is defined with a deterministic left-to-right reduction in order to evade the problem of distinct reduction paths having a different space usage. This does not matter for us as we will only be concerned with the size of the results of computations in this thesis.*

Based on the natural time measure, the certifying extraction mechanism [8] we already mentioned can also automatically generate recurrences describing the running time of an extracted function. It is up to the user to solve these recurrences to obtain explicit time bounds. For the running time analyses in this thesis, we make heavy use of this functionality.

2.4 Basic Notions of Complexity Theory

Now that we can use L for basic complexity theory, we define the usual notions like polynomial-time reductions and the class NP. The definitions and results in this section are due to Fabian Kunze. We leave out some of the technical details on paper.

Space and Time Complexities We start by formally defining basics like the $\mathcal{O}$ notation and what it means for a function to be in some complexity class.

Definition 2.12 ($\mathcal{O}$) *Let $f, g : \mathbb{N} \to \mathbb{N}$. $f \in \mathcal{O}(g)$ if there are c, n_0 such that for all $n \geqslant n_0$, it holds that $f\,n \leqslant c \cdot g\,n$.*

This is the usual definition and the expected properties for addition and multiplication can be proved. We say that a function $f : \mathbb{N} \to \mathbb{N}$ is polynomially bounded if there exists $n : \mathbb{N}$ such that $f \in \mathcal{O}(\lambda x.x^n)$. Moreover, f is monotonic if $\forall x\,y, x \leqslant y \to f\,x \leqslant f\,y$.

Definition 2.13 (Polynomial-time Computable Functions) *Assume types X, Y which are L-encodable. Let $f : X \to Y$ be an L-computable function. f is computable in polynomial time if there exists a function $f_t : \mathbb{N} \to \mathbb{N}$ such that*

- *for all $x : X$, the number of reduction steps of f's encoding on the encoding of x is bounded by $f_t(\|\overline{x}\|)$, where $\|\overline{x}\|$ is the size of x's encoding,*
- *f_t is monotonic and polynomially bounded,*

- *and the output size of* f *is polynomially bounded, i.e. there exists a function* $f_s : \mathbb{N} \to \mathbb{N}$ *which is monotonic and polynomially bounded such that* $\forall x, \left\|\overline{f\ x}\right\| \leqslant f_s(\|\overline{x}\|)$.

Monotonicity would not strictly be required for the bounding functions, but is quite convenient. We need the condition that the output size of f is polynomially bounded in order to avoid size-exploding terms. A size-exploding function does not have the properties one would intuitively expect of a polynomial-time computable function.

P and NP We continue with the definition of the important classes P and NP. A problem is a predicate $Q : X \to \mathbb{P}$ for an L-encodable type X.

Definition 2.14 (Decidable Problems) *Let* $Q : X \to \mathbb{P}$ *for an L-encodable type* X. Q *is L-decidable if there exists a L-computable function* $f : X \to \mathbb{B}$ *such that* $\forall x, Q\ x \leftrightarrow f\ x = \mathsf{T}$. *Moreover,* Q *is decidable in time* $f_t : \mathbb{N} \to \mathbb{N}$ *if additionally the number of reduction steps of* f*'s encoding on the encoding of* x *is bounded by* $f_t(\|\overline{x}\|)$.

Definition 2.15 (Polynomial-time Decidable Problems) *Let* $Q : X \to \mathbb{P}$ *for an L-encodable type* X. Q *is polynomial-time decidable if there exists a monotonic and polynomially bounded function* $f_Q : \mathbb{N} \to \mathbb{N}$ *such that* Q *is decidable in time* f_Q.

P is the class of problems for which there exists a polynomial-time decider.

Definition 2.16 (P) $Q \in \mathsf{P} :\equiv Q$ is polynomial-time decidable

For NP, we do not use the usual definition via nondeterminism. While nondeterministic additions to the λ-calculus have been explored in the literature, for instance via the addition of a new combinator allowing to nondeterministically guess a single bit [19], using nondeterminism would make formal reasoning much harder. Instead, we adapt the well-known alternative verifier characterisation of NP, where there must exist a verifier which decides whether a certificate of polynomial size correctly proves that an instance is a yes-instance. Intuitively, the verifier characterisation moves the nondeterminism into the input.

Definition 2.17 (NP) *Let* $Q : X \to \mathbb{P}$ *and* X *be an* L*-encodable type.* $Q \in \mathsf{NP}$ *if there exists a verifier* $R : X \to \mathsf{term} \to \mathbb{P}$ *such that:*

- $\lambda(x, y).R\ x\ y$ *is polynomial-time decidable*
- *and there exists a monotonic and polynomially-bounded function* $f_Q : \mathbb{N} \to \mathbb{N}$ *bounding the size of certificates, such that*
  - *if* $R\ x\ y$, *then* $Q\ x$,
  - *and for all* x *with* $Q\ x$, *there exists* y *such that* $R\ x\ y$ *and* $\|\overline{y}\| \leqslant f_Q(\|\overline{x}\|)$.

Note that we require that the type of certificates is term. This does not pose a restriction in practice as we can still use any L-encodable type X (since these types have an encoding as a term) whose encoding can be decoded in polynomial time, i.e. for which there exists a function $f : \mathsf{term} \to \mathbb{O}(X)$ running in polynomial time. It turns out that the Scott encoding of any inductive datatype can in practice be decoded in linear time.

Fact 2.18 (NP **subsumes** P) *If* $Q \in P$ *then* $Q \in NP$.

Polynomial-time Reductions Next, we consider polynomial-time reductions. The definition mostly corresponds to the usual one on Turing machines.

Definition 2.19 (**Polynomial-time Reductions**) *Assume problems* $P : X \to \mathbb{P}$ *and* $Q : Y \to \mathbb{P}$ *for L-encodable types* X, Y. P *reduces to* Q *in polynomial-time, written* $P \preceq_p Q$, *if there exists a polynomial-time computable function* $f : X \to Y$ *satisfying the property* $\forall x, P\ x \leftrightarrow Q(f\ x)$.

Note that, since we require f to be polynomial-time computable, its output size also needs to be polynomial in its input size. This is the only condition which changes compared to Turing machines.

$\preceq_p$ is transitive and inclusion in NP transfers backwards along reductions, i.e. if $P \preceq_p Q$ and $Q \in \mathsf{NP}$, then $P \in \mathsf{NP}$.

Definition 2.20 (NP**-hardness and** NP**-completeness**) *A problem* $Q : X \to \mathbb{P}$ *is NP-hard if for any* Y *whose encoding* $\lambda\, x.\overline{x}$ *can be computed in polynomial time any problem* $P : Y \to \mathbb{P}$ *can be reduced to* Q *in polynomial time, i.e.* $P \preceq_p Q$. *It is NP-complete if, in addition,* $Q \in \mathsf{NP}$.

The requirement that Y needs to be encodable in polynomial time is new compared to Turing machines. Intuitively, this corresponds to re-encoding the alphabet of Turing machines from an arbitrary alphabet Σ to, for instance, a binary alphabet, which is needed if one wants to employ a universal Turing machine. This change of alphabet is trivially polynomial-time computable. Similarly, we need a common input format for a universal L-term – the type of L-terms seems to be a reasonable input type. In practice, the Scott encoding is linear-time computable, although a formal proof of this meta-result seems to be hard.

NP hardness transfers along polynomial-time reductions: if $P \preceq_p Q$ and P is NP-hard, then Q is NP-hard.

Remark 2.21 *We close this chapter by giving an intuitive explanation why one should not expect there to be an easy proof for the existence of a natural* (*i.e. machine-independent*) *and intuitive NP-complete problem in our setting of complexity theory in L.*

Assume that Q *is a natural* NP*-complete problem. We conjecture that, if there is a simple verifier for* Q *using L,* Q *also has a simple Turing machine verifier. This is supported by the fact that Karp's 21* NP*-complete problems* [18] *all have relatively simple verifiers using Turing machines. Thus we obtain a direct reduction from* Q *to the computation of Turing machines. Since* Q *is* NP*-hard, we can reduce the computation of L-terms to* Q*. By transititivity of reduction, we get a polynomial-time-overhead simulation of L using Turing machines. The two combined reductions thus solve L's size explosion problem. For a direct polynomial overhead simulation, one however needs a heap-based simulation of L.*

This indicates that the NP*-hardness proof of* Q *is at least as advanced as the heap-based simulation of L, which is arguably not very simple. Motivated by this reasoning, we propose to use Turing machines as an intermediate problem for proving a natural problem to be* NP*-hard: Turing machines are much more expressive than most natural problems, thus they seem to be a good reduction target candidate for solving the size explosion problem.*

Chapter 3

Reducing k-SAT to Clique

In this chapter, we give a first formalisation of a polynomial-time reduction from the k-**SAT** problem to the **Clique** problem on undirected graphs. The main purpose of this chapter is to introduce the style of proofs we use. In particular, we comment on some of the more technical aspects of doing the running time analysis in Coq so that we can focus on the less technical aspects in the later chapters.

The satisfiablity problem **SAT** on CNFs is well-known: given a Boolean formula in conjunctive normal form, we determine whether there exists a satisfying assignment to the variables. k-**SAT** restricts the definition to CNFs with a clause size of k for a fixed k. **Clique** is a graph-based problem. Given an undirected graph $G = (V, E)$, a k-clique is a set of vertices $C \subseteq V$ such that $|C| = k$ and all possible edges between vertices of C are present in E. For a graph G and a number k, the **Clique** problem asks whether there exists a k-clique in the graph G.

The reduction from k-**SAT** to **Clique** is an example of a relatively simple reduction: in [5], it is one of the first examples of a polynomial-time reduction in an undergraduate course on theoretical computer science. There, the full proof on paper does not even span half a page. In particular because of this apparent simplicity on paper, we deem this reduction to be an interesting first exploration of the techniques we will later use on a larger scale.

3.1 Satisfiability of Conjunctive Normal Forms (CNF)

We start by formalising the satisfiablity problem **SAT**. A literal is either a Boolean variable or its negation. The disjunction of a number of literals is a clause. Clauses can be combined conjunctively to form a conjunctive normal form.

$$v : \mathsf{var} \coloneqq \mathbb{N} \qquad \boxed{\mathsf{var}}$$
$$l : \mathsf{literal} \coloneqq \mathbb{B} \times \mathbb{N} \qquad \boxed{\mathsf{literal}}$$
$$C : \mathsf{clause} \coloneqq \mathcal{L}(\mathsf{literal}) \qquad \boxed{\mathsf{clause}}$$
$$N : \mathsf{cnf} \coloneqq \mathcal{L}(\mathsf{clause}) \qquad \boxed{\mathsf{cnf}}$$

Here, the Boolean b of a literal (b, v) denotes the literal's *sign*. If the sign is negative (i.e. $b = \mathsf{F}$), the literal represents v's negation, otherwise it represents v.

An assignment $a :$ to a CNF assigns a Boolean value to each of its variables. We choose to model an assignment as the list of variables which are assigned the value T. All other variables are implicitly assigned the value F: $\mathsf{assgn} \coloneqq \mathcal{L}(\mathsf{var})$. $\boxed{\mathsf{assgn}}$

One can define evaluation functions $\mathcal{E}$ for variables, literals, clauses, and CNFs in a straightforward way. We will mainly use the function $\mathcal{E} : \mathsf{assgn} \to \mathsf{cnf} \to \mathbb{B}$ for CNFs, but refer to the other functions using the same identifier. Which function we mean will always be clear from the context and the used metavariables. $\boxed{\mathcal{E}}$

We have the following characterisations of evaluation:

Lemma 3.1 (Evaluation Equivalences)

1. $\mathcal{E}\ a\ v = \mathsf{T} \leftrightarrow v \in a$
2. $\mathcal{E}\ a\ (b, v) = \mathsf{T} \leftrightarrow \mathcal{E}\ a\ v = b$
3. $\mathcal{E}\ a\ C = \mathsf{T} \leftrightarrow \exists l \in C, \mathcal{E}\ a\ l = \mathsf{T}$
4. $\mathcal{E}\ a\ N = \mathsf{T} \leftrightarrow \forall C \in N, \mathcal{E}\ a\ C = \mathsf{T}$

We say that a satisfies the CNF N, denoted $a \models N$, if $\mathcal{E}\ a\ N = \mathsf{T}$. A similar notation is used for the satisfaction of clauses, literals and variables. $\boxed{a \models N}$

Now, we are able to define the problem **SAT**: $\boxed{\textbf{SAT}}$

Definition 3.2 (Satisfiability of CNFs) $\mathbf{SAT}\ N \coloneqq \exists a, a \models N$

In the sequel, we show that **SAT** is in NP, which we prove by giving a verifier for **SAT**. A sensible choice for a certificate of a CNF N being in **SAT** is a satisfying assignment. Thus a verifier is easy to define: given a CNF N and an assignment a, it checks whether a satisfies N. In order to prove the verifier correct, we must show that there exists an assignment of a size which is polynomial in N for every satisfiable formula N. As a list a containing the variables to which T is assigned may contain duplicates and variables which are not even used by the CNF, not every satisfying assignment has a polynomial size. Therefore, we introduce the notion of **small** assignments.

varsOfCnf

Definition 3.3 (Used Variables) *We can define a function* $\mathit{varsOfCnf} : \mathit{cnf} \to \mathcal{L}(\mathit{var})$ *calculating a list of variables contained in a CNF.*

small N α

Definition 3.4 (Small Assignments) *An assignment* α *is small with respect to a CNF* N *if it is duplicate-free and only contains variables used by the CNF:*

$$\mathit{small}\ \mathsf{N}\ \alpha \coloneqq \mathit{dupfree}\ \alpha \wedge \alpha \subseteq \mathit{varsOfCnf}\ \mathsf{N},$$

where $\mathit{dupfree} : \forall X, \mathcal{L}(X) \to \mathbb{P}$ *is a predicate capturing the absence of duplicates.*

Given an arbitrary assignment α for a CNF N, we can compute another assignment α′ which is small with respect to N by removing duplicates and variables not occuring in the CNF. To that end, we assume functions $\mathsf{dedup} : \forall(X : \mathsf{eqType}), \mathcal{L}(X) \to \mathcal{L}(X)$ and $\mathsf{intersect} : \forall(X : \mathsf{eqType}), \mathcal{L}(X) \to \mathcal{L}(X) \to \mathcal{L}(X)$ where dedup removes duplicates from a list and intersect yields a list that does exactly contain those elements that occur in both of its input lists.

compress

$$\mathsf{compress} : \mathsf{cnf} \to \mathsf{assgn} \to \mathsf{assgn}$$
$$\mathsf{compress}\ \mathsf{N}\ \alpha \coloneqq \mathsf{dedup}\ (\mathsf{intersect}\ \alpha\ (\mathsf{varsOfCnf}\ \mathsf{N}))$$

Semantically, compress changes the assignment: after removing assignments to unused variables, F is implicitly assigned to these unused variables. However, as the CNF does not refer to these variables, this does not matter. The compression of an assignment is always with respect to a particular CNF.

Lemma 3.5 (Correctness of compress**)**

1. $\mathit{small}\ \mathsf{N}\ (\mathit{compress}\ \mathsf{N}\ \alpha)$
2. $\nu \in \mathit{varsOfCnf}\ \mathsf{N} \to \mathcal{E}\ \alpha\ \nu = \mathcal{E}\ (\mathit{compress}\ \mathsf{N}\ \alpha)\ \nu$
3. $\alpha \models \mathsf{N} \leftrightarrow \mathit{compress}\ \mathsf{N}\ \alpha \models \mathsf{N}$

Lemma 3.6 (SAT is in NP**)** $\mathbf{SAT} \in \mathsf{NP}$

In order to prove Lemma 3.6, we need to show that there are valid certificates of polynomial size exactly for all satisfiable CNFs. Moreover, it has to be shown that the verifier runs in polynomial time. As these proofs are quite technical, we usually omit them on paper. However, as the analyses are an integral part of this work, the following section is an exception and treats the general procedure we use.

3.1.1 Running-time Analysis Using the Extraction Mechanism

We explain the methods we use to prove polynomial time and size bounds. As an example, we look at the evaluation function $\mathcal{E}$ for CNFs. Formally, the function is

defined as follows:

$$\begin{aligned} \mathcal{E}_N &: \mathsf{assgn} \to \mathsf{cnf} \to \mathbb{B} \\ \mathcal{E}_N\ a\ [\,] &\coloneqq \mathsf{T} \\ \mathcal{E}_N\ a\ (C :: N) &\coloneqq \mathcal{E}_C\ a\ C\ \&\ \mathcal{E}_N\ a\ N \end{aligned}$$

Here, we have annotated the evaluation function with a subscript to make clear which one we mean: $\mathcal{E}_N$ is the evaluation function for CNFs, while $\mathcal{E}_C$ is the function for clauses.

In order to extract this function to L, we first need to generate the encodings for the used datatypes, in this case for lists, $\mathbb{N}$, and $\mathbb{B}$[1]. Next, all the used functions need to be extracted first, in this case the evaluation of clauses and the Boolean conjunction. We assume this has already been done.

Now, the evaluation function itself can be extracted. During the process, the following recurrences for the running time are generated:

$$\mathsf{T}(\mathcal{E}_N)\ a\ [\,] \geqslant 9 \tag{3.1}$$

$$\mathsf{T}(\mathcal{E}_N)\ a\ (C :: N) \geqslant \mathsf{T}(\mathcal{E}_N)\ a\ N + \mathsf{T}(\mathcal{E}_C)\ a\ C + 22 \tag{3.2}$$

$\mathsf{T}(\cdot)$ can be read as "time-of". The numerical constants appearing in the inequalities are the numbers of reduction steps of the extracted function. We do not solve the recurrences directly, but instead define a recursive function $\mathsf{T}_{\mathcal{E}_N}$ implementing the equations.

Next, we aim to bound the function $\mathsf{T}_{\mathcal{E}_N}$ by a monotonic polynomial $p_{\mathcal{E}_N} : \mathbb{N} \to \mathbb{N}$ in the encoding size of the arguments such that

$$\forall a\ N, \mathsf{T}_{\mathcal{E}_N}\ a\ N \leqslant p_{\mathcal{E}_N}(\|\overline{a}\| + \left\|\overline{N}\right\|)$$

holds. In principle, we could also use a multivariate polynomial having one variable for each of the arguments, but this would only complicate things. In the case of $\mathcal{E}_N$, the polynomial $p_{\mathcal{E}_N}(n) \coloneqq n \cdot p_{\mathcal{E}_C}(n) + c_{\mathcal{E}_N} \cdot (n+1)$ works, where $p_{\mathcal{E}_C}$ is the polynomial bounding $\mathsf{T}(\mathcal{E}_C)$ and $c_{\mathcal{E}_N}$ is a constant. The bounding equation is then proved by induction on the CNF N.

Higher-order functions can be treated in a similar way, but there the running time bounds are conditional on bounds for the argument functions. In many cases, however, it seems to be easier to define a function via direct recursion instead of using a higher-order function which is more complicated to bound.

Recall that, regarding the size, we only need to bound the size of the result. The procedure of deriving a polynomial is similar in that case, although the extraction

[1] This has to be done only once and then never again.

mechanism does not derive space bounds currently; instead, we derive bounds via semantic arguments.

Remark 3.7 (Binary versus Unary Numbers) *In complexity theory, the question of the specific encoding of a problem is an important one. Especially for numbers, this is a delicate subject: a unary encoding of numbers is exponentially larger than the binary encoding. Therefore, one usually uses a binary encoding of numbers. Some number-theoretic problems, for instance, are not NP-hard anymore if a unary encoding instead of a binary encoding is used. In this thesis, however, we exclusively use a unary encoding, mainly because it is much easier to work with unary numbers in Coq, although a binary encoding would be possible. For all of the problems we study, using unary numbers is actually sound. Intuitively, this is the case since the numbers do not play a central part in the definition: in* **SAT**, numbers are only used to represent variables. If n variables are used by a CNF, only the numbers up to n need to be used by the encoding.

3.1.2 k-SAT

Finally, as a variant of **SAT**, we consider the problem k-**SAT** where each clause consists of exactly k literals.

We inductively define a predicate stating this property.

k-CNF

$$\frac{}{k\text{-CNF }[\,]} \qquad \frac{|C| = k \quad k\text{-CNF } N}{k\text{-CNF } (C :: N)}$$

The k-**SAT** problem additionally requires that $k > 0$. This is an arbitrary choice as the problem is trivially polynomial-time decidable if no clause contains a literal.[2]

k-**SAT**

Definition 3.8 (k-SAT) $k\text{-}\mathbf{SAT}\ N \coloneqq k\text{-}\mathit{CNF}\ N \wedge k > 0 \wedge \mathbf{SAT}\ N$

Remark 3.9 *Note that k acts as a parameter to the problem and is not part of an instance. More precisely, we might say that* k-**SAT** (for indeterminate k) is a **class** of problems, where we get the problem k-**SAT** for every fixed $k : \mathbb{N}$.

As k-CNF N can easily be decided in polynomial-time, we obtain a reduction from k-**SAT** to **SAT**. For the case that $k = 0$ or that N is not a k-CNF, we map to a trivial no-instance such as $x_0 \wedge \neg x_0$.

Lemma 3.10 (k-SAT reduces to SAT) $k\text{-}\mathbf{SAT} \preceq_p \mathbf{SAT}$

Proof As explained in Section 2.4, we need to verify that

- the reduction is correct, i.e. satisfies the reduction equivalence,

[2] In this case, N is a yes-instance iff $N = [\,]$.

- runs in polynomial time,
- and produces instances of a size which is polynomial in the input size.

The first condition is easy to verify, while the other two conditions use the techniques of the previous section. □

3.2 Cliques in Undirected Graphs

The second problem involved in our first reduction is the **Clique** problem on undirected graphs. We start by presenting our formalisation of undirected graphs.

UGraph

Definition 3.11 (Undirected Graphs) *An undirected graph* $\mathsf{G} = (\mathsf{V}, \mathsf{E}) : \mathit{UGraph}$ *consists of a type of vertices* $\mathsf{V} : \mathit{finType}$ *and an edge relation* $\mathsf{E} : \mathsf{V} \to \mathsf{V} \to \mathbb{P}$ *together with the following proofs:*

- E *is symmetric, i.e.* $\forall v_1\ v_2 : \mathsf{V}, \mathsf{E}\ v_1\ v_2 \leftrightarrow \mathsf{E}\ v_2\ v_1$,
- *and* E *is decidable, that is, there exists a function* $\mathsf{E}_{\mathit{dec}} : \forall v_1\ v_2 : \mathsf{V}, \mathit{dec}\ (\{v_1, v_2\} \in \mathsf{E})$, *where* $\mathit{dec}\ \mathsf{p} \coloneqq \mathsf{p} + (\neg \mathsf{p})$ *and we write* $\{v_1, v_2\} \in \mathsf{E}$ *for* $\mathsf{E}\ v_1\ v_2$.

We use the notation $\{v_1, v_2\} \in \mathsf{E}$ *instead of* $(v_1, v_2) \in \mathsf{E}$ *to explicitly express that* E *is symmetric.*

Remark 3.12 *From the perspective of proving properties about graphs, the previous definition provides an extremely convenient formalisation as all elements of type* V *are automatically a vertex of the graph and the edge relation is just a decidable proposition instead of, say, a list of pairs. However, the definition has a major problem: the propositional part of the definition, i.e. the edge relation, is not extractable to L and the extraction of the finite type is at least difficult. Moreover, even if the propositional parts would not cause any trouble, the definition makes no statements about the* ***time*** *that is needed to decide the edge relation.*

We will deal with this issue in Section 3.2.1 by introducing a "flat" first-order encoding of graphs which can be extracted. As it will be much harder to work with that definition, though, we define an equivalence between both definitions so that we can prove the correctness statements using the full propositional version while doing the running-time analysis on the flat version.

We define the metavariables $\mathsf{G} : \mathsf{UGraph}$, $v : \mathsf{V}_\mathsf{G}$ and $e : \mathsf{E}_\mathsf{G}$, where the subscript signifies that we mean the elements V and E of the graph G. If the graph G is clear from the context, we may omit the subscript.

Based on this definition of graphs, we can formalise cliques.

Definition 3.13 (Cliques and k-Cliques) *Fix a graph* $G : \mathit{UGraph}$*. A list of vertices* L *is a clique of* G *if all the edges between its elements are present in the graph.*

clique

$$\mathit{clique} : \mathcal{L}(V_G) \to \mathbb{P}$$
$$\mathit{clique}\ L \coloneqq \mathit{dupfree}\ L \land \forall v_1\ v_2 \in L, \{v_1, v_2\} \in E_G$$

k-*clique*

A k*-clique is a clique of exactly* k *elements:* $k\text{-}\mathit{clique}\ L \coloneqq |L| = k \land \mathit{clique}\ L$.

Alternatively, we can define k-cliques inductively, obtaining a definition which is more suitable for inductive proofs.

k-clique′

$$\frac{}{0\text{-clique}'\ [\,]} \qquad \frac{v \notin L \quad (\forall v' \in L, \{v, v'\} \in E_G) \quad k\text{-clique}'\ L}{(1+k)\text{-clique}'\ (v :: L)}$$

The **Clique** problem asks whether there exists a k-clique in a graph G:

Clique

Definition 3.14 (Clique) $\mathbf{Clique}\ (G, k) \coloneqq \exists L : \mathcal{L}(V_G), k\text{-clique}\ L$

Note that the number k is part of the instance, in contrast to the parameter k of the class k-**SAT**.[3]

3.2.1 First-order Encoding of Graphs

We now present a variant of graphs and the Clique problem which is extractable to L. This representation eliminates the components which cause trouble (see Remark 3.12) in the above definition: the finite type and the propositional edge relation. The finite type we replace by a natural number v giving the number of vertices. Valid vertices are natural numbers which are smaller than v. The edge relation is replaced by a list of pairs of vertices: $\mathsf{FlatUGraph} \coloneqq \mathbb{N} \times \mathcal{L}(\mathbb{N} \times \mathbb{N})$.

FlatUGraph

This type does allow instances which do not make any sense syntactically. Therefore, we enforce syntactic constraints externally:

FlatUGraph_wf

$$\mathsf{FlatUGraph\_wf}\ (V, E) \coloneqq (\forall (v_1, v_2) \in E, (v_2, v_1) \in E) \land (\forall (v_1, v_2) \in E, v_1 < V \land v_2 < V)$$

One can easily build a Boolean decider for these constraints.

Of course, we want to connect graphs of type UGraph and flat graphs of type FlatUGraph somehow. We first define some general relations for the representation of finite types by natural numbers.

Definition 3.15 (Representation of Finite Types) *Let* $T : \mathit{finType}$*,* $t : T$*, and* $k, n : \mathbb{N}$.

$n \approx T$

$$n \approx T \coloneqq n = |T|$$

$k \approx' t$

$$k \approx' t \coloneqq k = \mathit{index}\ t$$

$k \approx_{T,n} t$

$$k \approx_{T,n} t \coloneqq k = \mathit{index}\ t \land n \approx T$$

[3] One can show that for every fixed k, the **Clique** problem is decidable in polynomial time.

Using this definition, it is easy to encode type constructors like $\mathbb{O}(\cdot)$, $\cdot \times \cdot$, and $\cdot + \cdot$, as well as value constructors like $^\circ\cdot$, $(\cdot,\cdot)$, and $\mathsf{inl}\ \cdot$. More generally, we obtain the same closure properties for the flat representation of finite types as for finite types.

Example 3.16 (Encoding of Option Types) *We define* $\mathit{flatOpt}\ t \coloneqq 1+t$, $\mathit{flatSome}\ k \coloneqq 1 + k$ *and* $\mathit{flatNone} \coloneqq 0$. *Assuming that* $n \approx T$ *and* $k \approx_{T,n} t$, *we have:*

$$\mathit{flatOpt}\ n \approx \mathbb{O}(T) \qquad \mathit{flatNone} \approx_{\mathbb{O}(T),\mathit{flatOpt}\ n} \emptyset \qquad \mathit{flatSome}\ k \approx_{\mathbb{O}(T),\mathit{flatOpt}\ n}\ ^\circ t$$

Thus, we can define a representation relation for graphs as follows:

Definition 3.17 (Representation of Graphs) *Let a graph* $G = (V, E) : \mathit{UGraph}$ *and a flat graph* $g = (v, e) : \mathit{FlatUGraph}$ *be given.* g *represents* G, *written* $g \approx G$, *if:*

$g \approx G$

- $v \approx V$,
- $\forall (v_1, v_2) \in e, v_1 < v \wedge v_2 < v \wedge \exists V_1\ V_2, \{V_1, V_2\} \in E \wedge v_1 \approx' V_1 \wedge v_2 \approx' V2$
- $\forall V_1\ V_2 : V, \{V_1, V_2\} \in E \to (\mathit{index}\ V_1, \mathit{index}\ V_2) \in e$

Cliques can be defined in the expected way for flat graphs. The only difference to the definition on non-flat graphs is that we have to explicitly require the graph to be syntactically wellformed (see FlatUGraph_wf). The corresponding problem is called **FlatClique**. A proof that **FlatClique** is contained in NP is straightforward.

FlatClique

We can then derive an agreement statement for the two definitions: if $(v, e) \approx (V, E)$ and if $l : \mathcal{L}(\mathbb{N})$ and $L : \mathcal{L}(V)$, then l is a clique in (v, e) if, and only if, L is a clique in (V, E). Moreover, from a clique for one representation we can obtain a clique for the other one. This agreement allows us to reason using the propositional version, but analyse the running time on the flat version. Reductions essentially solely use the flat definition, the propositional variant is only needed to make the proof of the reduction equivalence more elegant.

3.3 Reduction from k-SAT to Clique

Now that we have defined the involved problems, we present the reduction of k-**SAT** to **Clique**. Assume that the CNF is given as

$$(l_{0,0} \vee \ldots \vee l_{0,k-1}) \wedge \ldots \wedge (l_{m-1,0} \vee \ldots \vee l_{m-1,k-1}),$$

for a number of clauses m. The basic idea of the reduction is to create a graph with vertices (i, j) for $i \in [0..m-1]$ and $j \in [0..k-1]$ such that we have a vertex (i, j) for every literal $l_{i,j}$. The edges between the vertices corresponding to two literals encode whether the literals can be satisfied at the same time and belong to different clauses. As a CNF is satisfied if and only if there exists a satisfied literal for every clause, the satisfiability problem then reduces to checking whether there exists a

m-clique in the constructed graph. In the following, we present this construction and its proof of correctness formally.

We start with the reduction to the non-flat version **Clique** using graphs of type UGraph. In order to prove polynomial-time computability, we then transfer the results to the flat version **FlatClique**.

For the rest of this section, let us fix a number k with $k > 0$ and a k-CNF N.[4]

N[i]

C[j]

N[i, j]

In order to define the graph, we need to be able to index into a CNF to obtain the clauses and literals at particular positions. We use the operations $N[i] : \mathbb{O}(\mathsf{clause})$ to obtain the clause at index i and $C[j] : \mathbb{O}(\mathsf{literal})$ to obtain the j-th literal of a clause C. From this, we derive $N[i, j]$ as a way to get the j-th literal of the i-th clause of the CNF N. The operations return option values in order to handle the case that the indices are invalid.

Now, we can set up the graph $G : \mathsf{UGraph}$ in the following way:

V_G

conflict

E_G

$$
\begin{aligned}
V_G &:= F_{|N|} \times F_k \\
\mathsf{conflict}\ (s_1, v_1)\ (s_2, v_2) &:= v_1 = v_2 \wedge s_1 \neq s_2 \\
E_G((i_1, j_1), (i_2, j_2)) &:= i_1 \neq i_2 \wedge \forall l_1\ l_2, (N[\mathsf{index}\ i_1, \mathsf{index}\ j_1] = {}^\circ l_1 \\
&\qquad \wedge N[\mathsf{index}\ i_2, \mathsf{index}\ j_2] = {}^\circ l_2) \\
&\qquad \to\ \neg \mathsf{conflict}\ l_1\ l_2
\end{aligned}
$$

Here, the type F_n is a finite type consisting of n elements. Two literals l_1, l_2 are conflicting if they refer to the same variable but have different signs. The definition of the edge relation then captures exactly the intuition described above: two vertices are connected if their corresponding literals are not conflicting and belong to different clauses. Note that the way in which we handle the case that $N[\mathsf{index}\ i, \mathsf{index}\ j] = \emptyset$ does not matter: by construction, there will always exist a literal at these positions since we assumed that N is in k-CNF.

Remark 3.18 *Labelling the graph with the syntactic literals occuring in the CNF instead of the positions would not work: a single literal can occur multiple times in different clauses (or even in the same clause) while the construction requires us to differentiate these different occurences.*

One can easily check that E_G is symmetric and decidable. For the correctness of the reduction, our goal is to show that **SAT** $N \leftrightarrow$ **Clique** $(G, |N|)$.

[4] We will later handle $k = 0$ and instances not satisfying k-CNF N as special cases.

3.3.1 From Assignments to Cliques

First of all, let us assume that $\mathfrak{a}_{\mathsf{sat}}$ is a satisfying assignment for N, i.e. $\mathfrak{a}_{\mathsf{sat}} \models \mathsf{N}$. We have to derive the existence of a $|\mathsf{N}|$-clique in the graph G. The idea is to pick an arbitrary literal which is satisfied by $\mathfrak{a}_{\mathsf{sat}}$ for every clause. The corresponding vertices in G form a clique by the construction of the edge relation.

To prove this formally, we require a few facts.

Fact 3.19 *Let* $\mathfrak{a} : \mathit{assgn}$ *and* $\mathfrak{l}_1, \mathfrak{l}_2 : \mathit{literal}$ *be arbitrary.*

$$\mathcal{E}\ \mathfrak{a}\ \mathfrak{l}_1 = \mathsf{T} \to \mathcal{E}\ \mathfrak{a}\ \mathfrak{l}_2 = \mathsf{T} \to \neg \mathit{conflict}\ \mathfrak{l}_1\ \mathfrak{l}_2$$

Fact 3.20 *Let* C *be the* $\mathfrak{i}$*-th clause of* N*, i.e.* $\mathsf{N}[\mathfrak{i}] = {}^\circ\mathsf{C}$*, and* $\mathfrak{l} \in \mathsf{C}$*. Then:*

$$\exists(\mathfrak{ci}, \mathfrak{li}) : \mathsf{V}_\mathsf{G}, \mathfrak{i} = \mathit{index}\ \mathfrak{ci} \wedge \mathsf{C}[\mathit{index}\ \mathfrak{li}] = {}^\circ\mathfrak{l}$$

That is, there exists a vertex of the graph that corresponds to a literal which is syntactically equal to $\mathfrak{l}$ *and is part of the same clause as* $\mathfrak{l}$*.*

Note that we could strengthen the preceding fact by also fixing the position of the literal inside the clause. However, for the proof it is only relevant that the literal belongs to a specific clause.

We prove the existence of a clique by induction over the number of clauses. In order for an induction to go through, the statement needs to be strengthened: as we have fixed a graph for the whole CNF, we also require that the clique provided by the inductive hypothesis only uses vertices belonging to the smaller CNF. Moreover, we need that the corresponding literals are satisfied by $\mathfrak{a}_{\mathsf{sat}}$ in order to derive the existence of the necessary edges if we add a vertex for the new clause in the inductive step. Thus, we arrive at the following statement:

Lemma 3.21 (Existence of a Clique) *Assume a sub-CNF* N' *of* N *such that* $\mathsf{N} = \mathsf{N}'' +\!\!+ \mathsf{N}'$ *for some* N''*. Then there exists* $\mathsf{L} : \mathcal{L}(\mathsf{V}_\mathsf{G})$ *with:*

1. $|\mathsf{N}'|$*-clique* L*,*
2. $\forall(\mathfrak{ci}, \mathfrak{li}) \in \mathsf{L}, \mathit{index}\ \mathfrak{ci} \geqslant |\mathsf{N}''|$*,*
3. *and* $\forall(\mathfrak{ci}, \mathfrak{li}) \in \mathsf{L}, \exists \mathfrak{l}, \mathsf{N}[\mathit{index}\ \mathfrak{ci}, \mathit{index}\ \mathfrak{li}] = {}^\circ\mathfrak{l} \wedge \mathcal{E}\ \mathfrak{a}_{\mathit{sat}}\ \mathfrak{l} = \mathsf{T}$*.*

Proof By induction on N' using Facts 3.20 and 3.19 in the cons case. □

Corollary 3.22 *There exists a* $|\mathsf{N}|$*-clique* L *in* G*.*

3.3.2 From Cliques to Assignments

Conversely, we have to show that, given a $|N|$-clique L in G, there exists a satisfying assignment for N. The proof proceeds in four steps:

1. For every clause of N, there exists one corresponding vertex in L.
2. Therefore, there exists a list of (clause, literal)-indices (i, j) where every clause is mentioned exactly once and the referenced literals are non-conflicting.
3. If we map this list of positions to a list of syntactic literals, these literals are non-conflicting and as there is at least one literal per clause, it suffices to satisfy these literals in order to satisfy the whole CNF.
4. Finally, the list of literals can be turned into a satisfying assignment.

Only the proof of the first step is interesting, the other steps are straightforward. We use an instance of the pigeonhole principle:

Lemma 3.23 (16.8.7 in [27]) *Let* $A \subseteq B$ *with* $|B| < |A|$. *Then* $\neg$*dupfree* A.

Lemma 3.24 *For every clause of the CNF, there exists a corresponding vertex in* L*:*

$$\forall i < |N|, \exists (c_i, l_i), (c_i, l_i) \in L \wedge \mathit{index}\ c_i = i$$

Proof As for every i and L the goal is decidable, we can constructively do a proof by contradiction. Thus, assume that $\forall (c_i, l_i) \in L, \mathsf{index}\ c_i \neq i$. It suffices to show that the list of clause indices of L contains duplicates, i.e. $\neg(\mathsf{dupfree}[\ c_i \mid (c_i, l_i) \in L\ ])$, as there is only an edge between two vertices (c_{i_1}, l_{i_1}) and (c_{i_2}, l_{i_2}) if $c_{i_1} \neq c_{i_2}$ by definition of E_G. We apply Lemma 3.23 for $B \coloneqq [f_0, \ldots, f_{|N|-1}] \setminus f_i$, where f_k is the k-th element of the finite type $F_{|N|}$, as i is not contained in the list of clause indices by assumption. □

We omit the formal proof of the other steps in this presentation. In the end, we obtain a satisfying assignment a_{sat} for N.

Combining both directions, we get a reduction $f : \mathsf{cnf} \to \mathsf{UGraph} \times \mathbb{N}$ if we map CNFs which are not in k-CNF to a trivial no-instance.

Theorem 3.25 (k-SAT reduces to Clique) k-**SAT** $N \leftrightarrow$ **Clique** $(f\ N)$

However, this reduction cannot be extracted to L as we are using the unextractable type of graphs UGraph. Therefore, we derive a reduction to the flat version **Flat-Clique**. The construction of the graph mirrors the definition of E_G, where we use the flat constructors for pairs to construct the vertices, see Section 3.2.1. This graph is connected to the non-flat graph G by the relation $\approx$ (Definition 3.17). We obtain the following polynomial-time reduction:

Theorem 3.26 k-**SAT** $\preceq_{\mathrm{p}}$ ***FlatClique***

Remark 3.27 *A first version of this reduction* [13] *did not use a separate non-extractable version of graphs but instead directly proved correctness on the flat version. The mechanisation was much more tedious because, among other things, we always had to work with the possibility that a vertex is invalid, i.e. not actually part of the graph. The new proof halved the lines of code of the Coq mechanisation and is also much more understandable. Therefore, we believe that defining a separate extractable problem is the right way if the natural formalisation of a problem in Coq is not directly extractable.*

In the introduction to this chapter, we mentioned that the proof of this reduction given in an introductory course on theoretical computer science merely spans half a page. Doing the proof formally is a bit more work, as this chapter has shown. Nevertheless, if one uses the right abstractions (in our case the representation of graphs using finite types), it is quite manageable. Deriving a flat version which is extractable to L and doing the running time analysis poses some additional technical effort, especially if one relates this to the usual approach on paper, which amounts to "The reduction is *obviously* polynomial-time computable". The systematic approach of representing finite types using natural numbers makes the correctness proof of the flat reduction feasible and will be useful for the more complicated reductions of the coming chapters.

Chapter 4

Informal Overview of the Proof of the Cook-Levin Theorem

We give an outline of our proof of the Cook-Levin Theorem, which is loosely based on the presentation by Sipser [24]. After introducing the problem we reduce from, a tableau construction is presented. We show how we can encode deterministic computations of a Turing machine in such a tableau and later adapt the construction to incorporate a form of nondeterminism. Finally, the structure of the formal proof is described.

We formally state the main part of the Cook-Levin Theorem:

Theorem 4.1 (Cook-Levin) **SAT** is NP-hard[1].

4.1 A Generic Problem for Turing Machines

Recall that, in order to show a problem P to be NP-hard, we have to prove that any problem Q contained in NP can be reduced to it. In our setting of complexity theory in L, we thus have to reduce the computation of L to **SAT**. In Remark 2.21, we argued why it is hard to directly reduce from L to any natural problem. Therefore, we believe that the most practical path of reducing from L to **SAT** is to use Turing machines as an intermediate problem. We start from a generic problem for Turing machines. A definition of such a problem the reader might be familiar with is the following:

Given a nondeterministic Turing machine M, an input in and a number of steps t,
does M accept in in at most t steps?

This definition explicitly employs nondeterminism, making it hard to reason formally. Therefore, we use a characterisation that relates to this one in a similar way the verifier characterisation of NP relates to the one using nondeterminism: the nondeterminism is moved into the input.

[1] From this, we directly get NP-completeness, as was shown by Lemma 3.6.

TMGenNP: Given a deterministic Turing machine M, an input in, a maximal certificate length k' and a number of steps t, does there exist a certificate $cert$ with $|cert| \leqslant k'$ such that M accepts $in \# cert$ in at most t steps?

In a theory with Turing machines as the computational model, **TMGenNP** would be almost trivially NP-hard, although it would not be a natural problem as it depends on the model of computation. Using L, however, the hardness is not trivial anymore.

4.2 Deterministic Simulation: Tableau of Configurations

We first consider how the computation of a deterministic Turing machine can be encoded for an arbitrary input of which we only know that its size is at most k. It will be covered in the next section how we can come up with such an input in the first place, i.e. how we can "guess" a certificate.

At a first glance, it might seem unintuitive that one can encode the computation of a Turing machine using a fixed-size formula as it is not known a priori how much time or space the Turing machine needs on a particular input. The key is that **TMGenNP** just asks us to simulate a Turing machine M for a bounded number of steps t on inputs of a bounded size $\leqslant k \coloneqq |in| + k'$. Thus, the formula can be made large enough to account for the worst-case: t computational steps and a space usage of $t + k$ tape cells as the Turing machine can visit at most one cell per computational step[2].

With this insight, we write down the computation of the Turing machine in a tableau (Figure 4.1). The tableau consists of $t + 1$ lines, one for each configuration that can be encountered in t computational steps. Each line encodes one configuration consisting of a state symbol and the content of the tape. We call such a line a **configuration string**. The state symbol is also annotated with the symbol currently under the head. The first line contains the initial state q_0 and the input $\sigma_1, \ldots, \sigma_l$ for $l \leqslant k$, where the head is one symbol to the left of the input (denoted by the blank the state is annotated with). Note that each line has a fixed length accounting for the maximum number of tape cells the Turing machine can use. However, the Turing machine will usually use less space than is available. The cells unused by a configuration do contain special blanks $\sqcup$. These blanks are not contained in the Turing machine tape alphabet but are part of the tableau encoding, as our Turing machines do not have blanks built-in. Each line of the tableau is delimited by #, the purpose of which will become clear in Chapter 5. In the end, each cell of the tableau will be represented by a number of Boolean variables used by the **SAT** formula.

As noted above, the state symbol also contains the current symbol under the head.

[2]The bound is $t + k$ and not t since we need the space for the input even if the Turing machine does not read it.

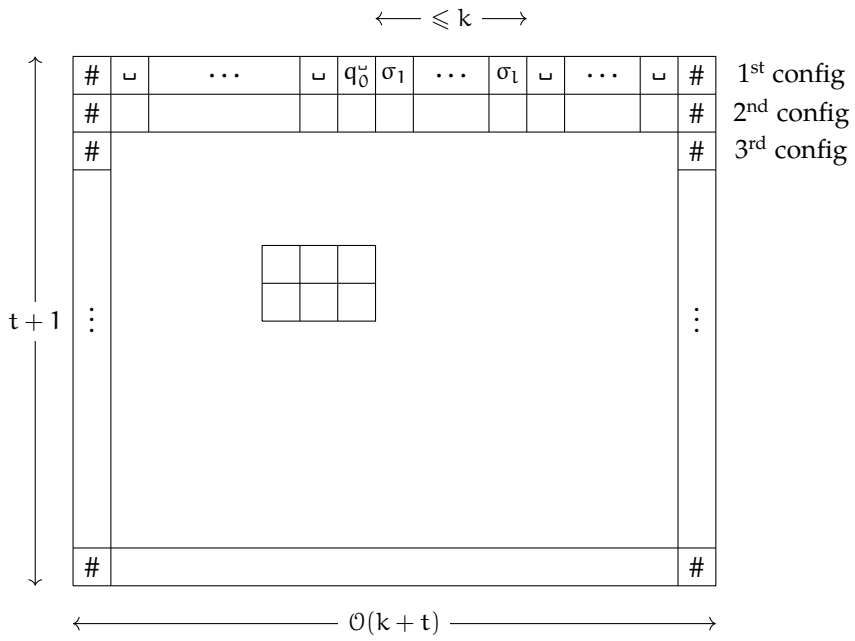


Figure 4.1: The tableau of configurations

This choice is clear as it makes the transition taken by the Turing machine only depend on one cell. However, there are multiple ways of dealing with head movements. The most straightforward way might be to move the position of the state symbol, resembling the movement of the head (**moving-head semantics**).

Example 4.2 (Moving Head) *Consider the Turing machine* M *over alphabet* $\Sigma = \{a, b, c\}$ *with* $Q = \{q_0, q_1\}$, *halt* $q_0 =$ *halt* $q_1 = F$ *and* $\delta(q_0, {}^\circ a) = (q_1, ({}^\circ b, L))$. *The successor of the tableau line*

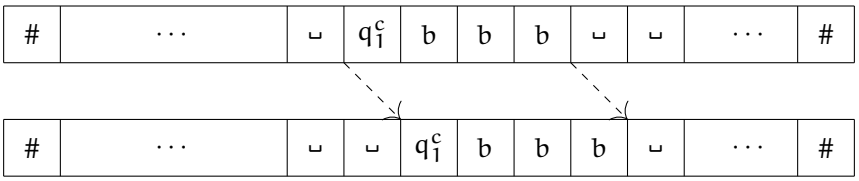
#	···	␣	c	q_0^a	b	b	␣	␣	···	#

using moving-head semantics would be

#	···	␣	q_1^c	b	b	b	␣	␣	···	#

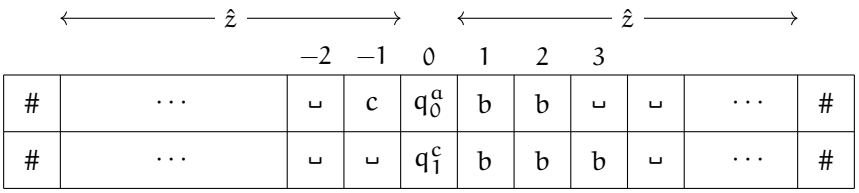
.

While this behaviour does work, it makes formal reasoning hard: Depending on **how** the Turing machine moves its head in order to write down a string, the configuration string will look differently, although it logically represents the same configuration. Thus, for any given configuration (q, tp) of the machine, there are multiple ways of encoding it as a configuration string:



Usually one wants to avoid such issues of non-uniqueness. Therefore, we instead use a **moving-tape semantics** where the position of the head (and thus the state symbol) is fixed at the center of the string. Instead of moving the head, the whole tape is shifted around the head.

Example 4.3 (Moving Tape) *Adapting example 4.2, the same transition looks like this when using moving-tape semantics:*



Note that the size of the substrings left and right of the center symbol is the same number $\hat{z}$ depending on t and k, expressing that the state symbol is fixed at the center.

With the moving-tape representation, we have arrived at a formally pleasing handling of head movements.

It remains to encode that the successive lines of the tableau follow from each other according to the Turing machine's transition function. For this, we introduce **rewrite windows** consisting of 3×2 symbols.

premise	x_1	x_2	x_3
conclusion	x_4	x_5	x_6

These rewrite windows are used to justify that one line of the tableau **validly** follows from the previous one. At each possible offset, there must exist a rewrite window whose premise matches a prefix of the previous string and whose conclusion matches a prefix of the succeding string.

Example 4.4 (Rewrite Windows) *Consider again example 4.3. For the positions labelled -2, -1, 0, 1 and 2 we need the following rewrite windows (from left to right):*

␣	c	q_0^a	c	q_0^a	b	q_0^a	b	b	b	b	␣	b	␣	␣
␣	␣	q_1^c	␣	q_1^c	b	q_1^c	b	b	b	b	b	b	b	␣

The key behind this construction is that the windows have to overlap. In this way, a global constraint (that the whole configuration string consistently follows from the previous one) is enforced using small local constraints given by the rewrite windows.

We create a set of rewrite windows that encode the valid transitions. Intuitively, these rewrite windows make the behaviours the Turing machine can have explicit, compared with the symbolic representation of the transition function. Of course, a great number of rewrite windows is needed. We mainly have three types of them: for shifts of the tape, for the transition function involving the center state symbol, and for replicating the final configuration if the Turing machine halts early in less than t steps.

Most of the windows need to exist for every possible combination of tape symbols. For instance, shifting the tape should be possible independently of its contents. Therefore, we introduce **rewrite rules** which are parameterised over variables ranging over, for instance, the tape alphabet.

Example 4.5 (Rewrite Rules for Tape Shifts) *The rewrite window*

b	b	$\sqcup$
b	b	b

is an instance of the more general rewrite rule

σ_1	σ_2	$\sqcup$
σ_3	σ_1	σ_2

,

where $\sigma_i : \Sigma$.

Note that we distinguish blanks $\sqcup$ and elements of Σ as $\sqcup$ is only a part of the representation of configurations as strings. Seeing this generalisation, one might now realise a problem regarding the tape shifts: the rewrite windows are not provided enough information for the overlapping constraints to enforce a consistent tape shift.

Example 4.6 (Inconsistent Tape Shifts) *We not only need to be able to shift the tape to the right but also leave its position unchanged. For instance, we need the following rewrite rule:*

σ_1	σ_2	$\sqcup$
σ_1	σ_2	$\sqcup$

Looking again at Example 4.3, this rule can be applied at the position labelled with 1 to obtain an alternative successor string:

		−2	−1	0	1	2	3			
#	⋯	␣	c	q_0^a	b	b	␣	␣	⋯	#
#	⋯	␣	␣	q_1^c	b	b	␣	␣	⋯	#

This change of configuration is inconsistent: the left half of the tape is shifted to the right, while the right half of the tape is left unchanged.

The reason for this inconsistency is that one cannot infer from the overlapping tape symbols alone in which direction the tape is shifted. We fix this problem by adding more information: polarities. Polarities $p : \mathsf{polarity} \coloneqq + \mid - \mid \circ$ are positive, negative, or neutral. Each symbol of the tape alphabet and each blank is annotated with a polarity indicating in which direction the tape is shifted. Polarities do not add any information to the configuration strings in their own right and are just relevant for the configuration changes. Notationally, we write $\overrightarrow{\sigma}$, $\overleftarrow{\sigma}$, $\overline{\sigma}$ for a symbol σ annotated with a positive, negative, or neutral polarity.

Example 4.7 (Polarities) *The rewrite rules mentioned above now have the following form:*

σ_1	σ_2	␣
$\overrightarrow{\sigma_3}$	$\overrightarrow{\sigma_1}$	$\overrightarrow{\sigma_2}$

σ_1	σ_2	␣
$\overline{\sigma_1}$	$\overline{\sigma_2}$	$\overline{\text{␣}}$

The inconsistent rewrite is not possible anymore as the windows overlapping from the left force a positive polarity:

		−2	−1	0	1	2	3			
#	⋯	␣	c	q_0^a	b	b	␣	␣	⋯	#
#	⋯	$\overrightarrow{\text{␣}}$	$\overrightarrow{\text{␣}}$	q_1^c	$\overrightarrow{b}$	$\overrightarrow{b}$	$\overrightarrow{b}$	$\overrightarrow{\text{␣}}$	⋯	#

Note that we omit the polarities in the premises of rules as they are irrelevant. Formally, for each of the rewrite rules depicted in Example 4.7, we do an instantiation not only for all possible combinations of variable valuations, but also for the three possible polarities in the premise (we only require that the polarities are consistent).

Remark 4.8 (Blanks and Polarities) *Formally, it is not necessary for blanks to be annotated with polarities. However, adding polarities to blanks allows us to handle symbols of the tape alphabet and blanks in a more uniform way, simplifying the proofs.*

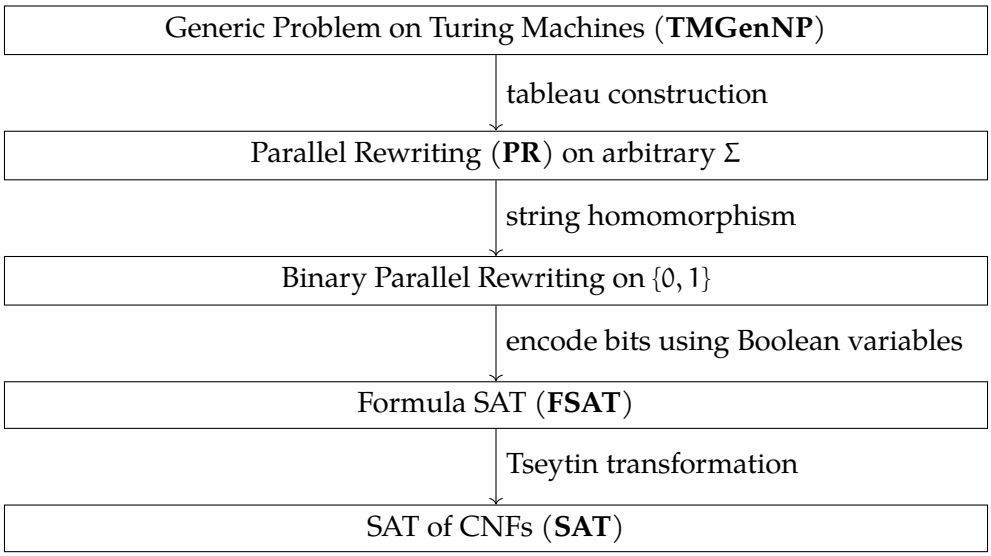
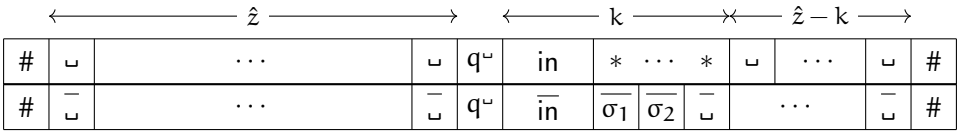


Figure 4.2: The chain of reductions from **TMGenNP** to **SAT**.

Finally, we need to enforce that the final configuration is a halting configuration. This can be encoded by requiring that an element of $[q^m \mid \mathsf{halt}\ q = \top, m : \Sigma + \{\sqcup\}]$ is contained in the last line of the tableau.

4.3 Nondeterministic Input

The construction presented in the previous section can simulate a deterministic Turing machine given an initial configuration. In the definition of **TMGenNP**, however, a part of the input, the certificate, is existentially quantified. This nondeterminism of the input can be encoded by prepending a new line to the tableau, where in is the fixed part of the input:



The idea is to add rewrite rules which can replace the $*$ (wildcard) symbols by arbitrary elements of the tape alphabet. If one wildcard is replaced by a blank, all the other wildcards to the right of it also need to be replaced by blanks in order to ensure that the used tape region is contiguous.

4.4 Intermediate Problems

In the presentation by Sipser [24], the tableau is directly encoded as a CNF. Doing this formally seems to be hard as one needs to handle the invariants of simulating the Turing machine, the representation of symbols of an arbitrary finite alphabet,

and the encoding as a CNF all at once. Instead, we factorise the proof into smaller parts by introducing three intermediate problems that deal with these concerns one-by-one. Figure 4.2 gives an overview over the reductions. The reduction of **TMGenNP** to Parallel Rewriting (**PR**), a string-based problem which will be introduced in Section 5.1, handles the main part of the Turing machine simulation. It creates an explicit representation of the Turing machine by enumerating all rewrite windows. After that, we do not have to reason about Turing machines anymore. Parallel Rewriting can be seen as a model between Turing machines and circuits. The remaining three reductions subsequently deal with the encoding of the alphabet, the encoding as a Boolean formula and the conversion to a CNF.

Chapter 5

Reducing GenNP to Parallel Rewriting

In this chapter, we introduce the string-based **Parallel Rewriting** (**PR**) problem which formally captures the idea of a string tableau constrained by rewrite windows. The rest of this chapter is then devoted to formalising the tableau construction presented in Chapter 4 and proving its correctness in a reduction from **TMGenNP** to **PR**. However, we do not encode the rewrite windows as Boolean formulas, yet, but instead stay at a string-based representation. This reduction does the main work of encoding Turing machines: one can see the construction as moving from a symbolic representation of Turing machines to an explicit one, where all possible "local" behaviours (expressed by the rewrite windows of the previous chapter) are written down explicitly.

First of all, we make the informal description of **TMGenNP** given in the previous chapter precise.

TMGenNP

Definition 5.1 (TMGenNP)

$$\mathbf{TMGenNP}(\Sigma, M, \mathtt{in}, \mathtt{k}', \mathtt{t}) \coloneqq \exists\, \mathtt{cert}, |\mathtt{cert}| \leqslant \mathtt{k}' \\ \wedge \exists \mathtt{q_f}, (\mathit{start}_M, \mathit{initTape}\ (\mathtt{in} +\!\!+ \mathtt{cert})) \rhd^{\leqslant \mathtt{t}} \mathtt{q_f},$$

where

$$\mathbf{TMGenNP} : (\mathbf{\Sigma}_\Sigma.\mathsf{mTM}\ \Sigma\ 1 \times \mathcal{L}(\Sigma) \times \mathbb{N} \times \mathbb{N}) \to \mathbb{P},$$

$\rhd$ *is the termination relation of Definition 2.4, and*

initTape

$$\mathit{initTape}\ [\,] \coloneqq \mathit{niltape}$$
$$\mathit{initTape}\ (\mathtt{x} :: \mathtt{xs}) \coloneqq \mathit{leftof}\ \mathtt{x}\ \mathtt{xs},$$

that is, the head is initially positioned left of the input.

Syntactically, it is important that the numbers $\mathtt{t}$ and $\mathtt{k}'$ are given in a unary encoding instead of a binary encoding: with a binary encoding, a verifier for **TMGenNP** could not even take time linear in the number of steps.

5.1 Parallel Rewriting

Parallel Rewriting works over a finite alphabet Σ. Given an initial string x_0, a set of rewrite windows R, a number of steps t, and a final constraint R_{final}, we are tasked with determining whether there exists a sequence of valid rewrites $x_0 \rightsquigarrow_R \dots \rightsquigarrow_R x_t$ using the rewrite windows R such that x_t contains an element of a set of strings R_{final} as a substring, written $x_t \models R_{\text{final}}$. Conceptually, the rewrite windows we consider here correspond to the rewrite windows of Chapter 4.

Definition 5.2 (Parallel Rewriting instances) $(\Sigma, o, w, x_0, R, R_{final}, t)$ *is a Parallel Rewriting instance, where* **PR** instances

- Σ : *finType is the finite alphabet,*
- $o : \mathbb{N}$ *is the rewrite offset with* $o > 0$*,*
- $\omega : \mathbb{N}$ *is the width of rewrite windows, where* $\omega > 0$ *and* $o \mid \omega$*,*
- $x_0 : \Sigma^*$ *is the initial string, where* $|x_0| \geqslant \omega$ *and* $o \mid |x_0|$*,*
- $R : \mathcal{L}(\mathit{window}_\omega(\Sigma))$ *is the set of rewrite windows,*
- $R_{\mathit{final}} : \mathcal{L}(\Sigma^*)$ *is the set of final substrings,*
- $t : \mathbb{N}$ *is the number of rewrite steps.*

Here, $\mathit{window}_\omega(\Sigma) := \Sigma^\omega \times \Sigma^\omega$ *denotes the type of windows of width* w *over* Σ *and* $n \mid m$ *means that* n *divides* m*.* $\mathit{window}_\omega(\Sigma)$

If the width is clear from the context, we write $\mathsf{window}(\Sigma)$ instead of $\mathsf{window}_\omega(\Sigma)$. Instead of directly using the projections π_1 and π_2 on a window w, we usually write $\mathsf{prem}\ w$ and $\mathsf{conc}\ w$. In the previous chapter, we considered rewrite windows of width 3. $|x_0|$ is the width of the rewrite tableau. The condition $|x_0| \geqslant \omega$ will become clear in Remark 5.6. Pictorially, o symbols are always grouped together to form one abstract symbol, explaining the divisibility constraints. Throughout this chapter, we always work with an offset of 1 and the more general case will only become relevant in Chapter 6. $\mathsf{prem}, \mathsf{conc}$

Let us fix the structural parameters Σ, o and ω satisfying the conditions stated in the definition for the rest of this section.

5.1.1 Validity

We start by defining what it means for a string a to validly rewrite to another string b, denoted $a \rightsquigarrow b$.

prefix

Definition 5.3 (Matching Windows)

$$\mathit{prefix}\ s\ t \coloneqq \exists b, t = s +\!\!+ b$$

rewHead

$$\mathit{rewHead}\ w\ a\ b \coloneqq \mathit{prefix}\ (\mathit{prem}\ w)\ a \wedge \mathit{prefix}\ (\mathit{conc}\ w)\ b,$$

that is, a window w justifies a rewrite at the head of two strings a and b if it matches the heads of a and b.

A window w justifies a rewrite at position i of strings a, b if it matches the head of the strings starting from position i: *rewAt* $w\ i\ a\ b \coloneqq$ *rewHead* $w\ a[i..]\ b[i..]$

rewAt

Intuitively, a rewrite $a \rightsquigarrow b$ is possible if we can find a rewrite window justifying a local rewrite for every possible offset. "Possible offsets" are those in the range $[0, |a| - \omega]$, as the windows have a width of ω. Additionally, we only need rewrites at multiples of the offset, again supported by the view that o symbols together form a unit.

Definition 5.4 (Validity, Explicit Characterisation) *Given a set of rewrite windows R, the validity of a rewrite from $a : \Sigma^*$ to $b : \Sigma^*$, written $a \rightsquigarrow_R^E b$, can be defined by:*

$a \rightsquigarrow_R^E b$

$$\begin{aligned} a \rightsquigarrow_R^E b \coloneqq\ & |a| = |b| \\ \wedge\ & o \mid |a| \\ \wedge\ & \forall 0 \leqslant i = j \cdot o \leqslant |a| - \omega, \exists w, w \in R \wedge \mathit{rewAt}\ w\ i\ a\ b \end{aligned}$$

Note that the dependency on o and ω is not made explicit in the notation, instead they need to be inferred from the context. If the set of rewrite windows R is clear, we may also drop that.

While this definition is fairly intuitive, it does not support easy inductive reasoning. Therefore, we use an equivalent inductive definition.

Definition 5.5 (Validity) *Given a set of rewrite windows R, the validity of a rewrite from $a : \Sigma^*$ to $b : \Sigma^*$, written $a \rightsquigarrow_R b$, is defined inductively:*

$a \rightsquigarrow_R b$

$$\frac{}{[\,] \rightsquigarrow_R [\,]}$$

$$\frac{a \rightsquigarrow_R b \quad |a| < \omega - o \quad |u| = o \quad |v| = o}{u +\!\!+ a \rightsquigarrow_R v +\!\!+ b}$$

$$\frac{a \rightsquigarrow_R b \quad |u| = o \quad |v| = o \quad w \in R \quad \mathit{rewHead}\ w\ (u +\!\!+ a)\ (v +\!\!+ b)}{u +\!\!+ a \rightsquigarrow_R v +\!\!+ b}.$$

This version prepends a chunk of o symbols in each step. The first two cases deal with proving validity of rewrites in strings of length $< \omega$. The third case is the interesting one and is defined in the intuitive way.

Remark 5.6 *It might seem peculiar that we do not require the strings to have a minimum length of ω: strings of length $< \omega$ can be rewritten vacuously to any other string as they are covered by no window and are disallowed by the definition of PR instances (see Definition 5.2). This choice enables us to only mention* rewHead *in the successor case of the inductive definition; otherwise, we would also need it in the base case. The main proofs of this chapter are simplified considerably by this choice, albeit at the cost of having nonsensical base cases.*

Proposition 5.7 (Vacuous Rewriting) *Let* $a, b : \Sigma^*$ *with* $|a| = |b| = k \cdot o < \omega$. *Then* $a \rightsquigarrow b$.

Proof By induction on k. □

Proposition 5.8 (Length Invariance) *Let* $a, b : \Sigma^*$ *with* $a \rightsquigarrow b$. *Then* $|a| = |b|$.

Lemma 5.9 (Agreement of $\rightsquigarrow^E$ and $\rightsquigarrow$) *For any set of rewrite windows* R, *it holds that*

$$a \rightsquigarrow_R^E b \leftrightarrow a \rightsquigarrow_R b.$$

Proof $\rightarrow$**:** By definition, we have k with $|a| = k \cdot o$. The proof is by induction on k with a and b quantified.

$\leftarrow$**:** By induction on $a \rightsquigarrow_R b$. □

5.1.2 The Parallel Rewriting Problem

Parallel Rewriting generates a sequence of strings according to $\rightsquigarrow_R$ for a set of rewrite windows R. The last string should contain one element of a set of strings R_{final} as a substring.

Definition 5.10 (Substring constraint) *Given a set of strings* $R_{final} : \mathcal{L}(\Sigma^*)$, *string* s *satisfies* R_{final}, *written* $s \models R_{final}$, *if:* $\boxed{s \models R_{final}}$

$$\exists subs\ k, subs \in R_{final} \wedge k \cdot o \leqslant |s| \wedge \mathit{prefix}\ subs\ s[k \cdot o..]$$

The definition requires a string contained in R_{final} to be a substring of the final string at a position which is a multiple of the offset o.

Definition 5.11 (Parallel Rewriting) $\boxed{\textbf{PR}}$

$$\textbf{PR}\ (\Sigma, o, \omega, x_0, R, R_{\text{final}}, t) \coloneqq \exists x_t, x_0 \rightsquigarrow_R^t x_t \wedge x_t \models R_{\text{final}},$$

where we implicitly require the instance to satisfy the syntactic constraints of Definition 5.2.

One can interpret **PR** to be a problem between Turing machines and circuits: of course, the definition over a finite alphabet still closely resembles Turing machines. However, in contrast to Turing machines, Parallel Rewriting can completely rewrite the string in a single step, although the power of this is limited as it operates on strings of a fixed size. Circuits, on the other hand, are similar in the sense that they also work in parallel. The fact that adjacent rewrite positions overlap and can thus enforce a global constraint in a single rewrite step is unlike circuits, though.

5.1.3 3-PR

For the rest of this chapter, we fix the width ω to 3 and the offset o to 1, which are the parameters needed for the Turing machine encoding. We call this variant 3-**PR**.

$a \rightsquigarrow_R b$

The inductive definition of validity can be simplified a bit:

$$\frac{}{[\,] \rightsquigarrow_R [\,]} \qquad \frac{a \rightsquigarrow_R b \quad |a| < 2}{x :: a \rightsquigarrow_R y :: b} \qquad \frac{a \rightsquigarrow_R b \quad w \in R \quad \mathsf{rewHead}\ w\ (x :: a)\ (y :: b)}{x :: a \rightsquigarrow_R y :: b}$$

We use the notation introduced in the previous chapter to denote the window $((x_1, x_2, x_3), (x_4, x_5, x_6))$:

x_1	x_2	x_3
x_4	x_5	x_6

Sometimes, we also use $[x_1, x_2, x_3]\ /\ [x_4, x_5, x_6]$ if we need to write down a window in-line.

5.2 Encoding Tapes and Configurations

We start with the construction of the Turing machine simulation. Our goal is to devise and verify a 3-**PR** instance simulating a given Turing machine. For the rest of the chapter, let us fix a **TMGenNP** instance consisting of a finite tape alphabet Σ, a single-tape Turing machine $M = (Q, \delta, \mathsf{start}, \mathsf{halt})$ over Σ, an input in, a number t of steps, and a maximum size k' of the certificate.

validCert

Definition 5.12 *validCert* $c \coloneqq |c| \leqslant k'$

Formalising the intuitions from Chapter 4, we define the alphabet Γ of the Parallel

Rewriting instance for the deterministic simulation[1].

$$\begin{aligned}
\mathsf{polarity} : \mathbb{T} &\coloneqq + \mid - \mid \circ \\
\mathsf{delim} : \mathbb{T} &\coloneqq \# \\
\Sigma_{state} &\coloneqq \mathbb{O}(\Sigma) \\
\mathsf{States} &\coloneqq Q \times \Sigma_{state} \\
\Sigma_{tape} &\coloneqq \mathsf{polarity} \times \Sigma_{state} \\
\Sigma_{delim} &\coloneqq \mathsf{delim} + \Sigma_{tape} \\
\Gamma &\coloneqq \mathsf{States} + \Sigma_{delim}
\end{aligned}$$

$\boxed{\Sigma_{state}}$ $\boxed{\mathsf{States}}$ $\boxed{\Sigma_{tape}}$ $\boxed{\Sigma_{delim}}$ $\boxed{\Gamma}$

The three polarities give the direction in which a tape was last shifted. Σ_{state} adds blanks to the alphabet Σ. Elements of Σ_{state} have no polarity and are used for the state symbols States, while elements of the tape alphabet Σ_{tape} are additionally annotated with a polarity. Finally, the full alphabet Γ is either a state symbol, a tape symbol or the delimiter #.

We need the subalphabets at various points. When we write down elements of Γ, we leave the injections into the sum types and option types implicit. Sometimes, we also implicitly lift functions from a smaller alphabet to a larger one. We use the metavariables $\sigma : \Sigma$, $m : \Sigma_{tape}$, $p : \mathsf{polarity}$, $\gamma : \Gamma$, $u : \Sigma^*$, and $h : \Gamma^*$. Given m, we write $\overrightarrow{m}$ for $(+, m)$, $\overleftarrow{m}$ for $(-, m)$, and $\overline{m}$ for $(\circ, m)$. For an unknown polarity p, we use m^p.

$\boxed{\overleftarrow{m}, \overrightarrow{m}, \overline{m}}$ $\boxed{m^p}$

Definition 5.13 (Polarity Reversion)

$$\begin{aligned}
\mathit{pFlip}\, p &\coloneqq \textbf{\textit{match}}\ p[\ \circ \Rightarrow \circ \mid + \Rightarrow - \mid - \Rightarrow +\ ] \\
\mathit{pRev}\, (h : \Gamma^*) : \Gamma^* &\coloneqq \mathsf{rev}[\ \mathit{pFlip}\, x \mid x \in h\ ]
\end{aligned}$$

$\boxed{\mathit{pRev}}$

Notationally, we write $\sim p$ *for* $\mathit{pFlip}\, p$. $\boxed{\sim p}$

pRev reverses a string and flips the polarities the symbols are annotated with. Note that we are already using an implicit lifting of pFlip from polarity to Γ in the definition of pRev.

Fact 5.14 (Involutions) *pFlip and pRev are involutions, that is,* $\mathit{pFlip}\, (\mathit{pFlip}\, p) = p$ *and* $\mathit{pRev}(\mathit{pRev}\, h) = h$.

Figure 5.1 shows the layout of a configuration string again. We refer to the center symbol as the **state symbol** and to the substrings left and right of it as the **left tape**

[1] This does not include the part for generating the initial configuration.

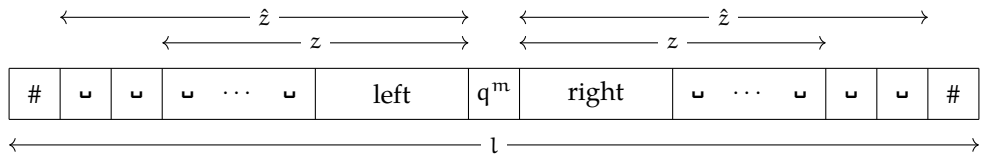


Figure 5.1: Layout of a configuration string.

half and the **right tape half**. In dependence on t, k', and in, we define the following numbers:

k
$$k \coloneqq |in| + k'$$
z
$$z \coloneqq t + k$$
$\hat{n}$
$$\forall n, \hat{n} \coloneqq n + 2$$
l
$$l \coloneqq 2 \cdot (\hat{z} + 1) + 1$$

The real maximum input size after accounting for the fixed input in and the nondeterministic certificate is k. z is the amount of space available for the Turing machine and thus z units of space need to be available to the left and to the right in the configuration string as the tape can be shifted to be completely on one side of the state symbol. For technical reasons, we want three symbols to be available to either side of the state symbol even if $t = k = 0$. One of those symbols is the delimiter #, the other two are additional blanks which will never be used by the Turing machine. Thus $\hat{z}$ is the number of symbols on each side excluding the delimiter. Finally, l is the length of the whole configuration string including the center state symbol.

In order to make reasoning about configuration strings possible, we define representation relations for tape halves and configurations.

E **Definition 5.15 (Tape Representation)** *The string* E *represents the empty tape:*

$$E\ p\ 0 \coloneqq [\#]$$
$$E\ p\ (1 + n) \coloneqq (p, ␣) :: E\ p\ n$$

$u \sim_t^{(n,p)} h$ *Tape representation is defined as:*

$$u \sim_t^{(n,p)} h \coloneqq |u| \leqslant n \wedge h = [\ (p, x) \mid x \in u\ ] +\!\!+ E\ p\ (\hat{n} - |u|)$$

$u \sim_t^{p} h$
$$u \sim_t^{p} h \coloneqq u \sim_t^{(z,p)} h,$$

$u \sim_t^{(n,p)} h$ means that h contains the elements of u annotated with polarity p where a total of n symbols are available for the simulation to use (not counting the three additional symbols). For the correctness statements, we use $u \sim_t^{p} h$, but usually generalise to $u \sim_t^{(n,p)} h$ for some n in inductive proofs.

Proposition 5.16 *We have the following basic facts related to the representation of tapes.*

1. *$\sim$ commutes with* E: $[\ \sim\gamma \mid \gamma \in \mathsf{E}\ p\ n\ ] = \mathsf{E}\ (\sim p)\ n$
2. *tape representation is invariant under $\sim$:* $u \sim_t^{(n,p)} h \to u \sim_t^{(n,\sim p)} [\ \sim\gamma \mid \gamma \in h\ ]$

A configuration $c = (q, tp)$ is represented by a string containing the state symbol at the center with strings representing the left and right tape halves to the left and right of it.

Definition 5.17 (Configuration Representation)

$$(q, tp) \sim_c (l, e, r) := \exists p, e = (q, \mathit{current}\ tp) \wedge \mathit{left}\ tp \sim_t^p l \wedge \mathit{right}\ tp \sim_t^p r$$
$$c \sim_c s := \exists l\ e\ r, s = \mathsf{rev}\ l +\!\!+ [e] +\!\!+ r \wedge (q, tp) \sim_c (l, e, r)$$

$(q, tp) \sim_c (l, e, r)$

$(q, tp) \sim_c s$

Note that the left tape half is reversed: while the Turing machine tapes have the symbol closest to the machine head at the head of the list, we need the symbol closest to the head to be next to the string's center. This will pose some difficulties later on.

Both $\sim_t$ and $\sim_c$ are computational in the sense that, given a tape half or a configuration, respectively, we can compute a string representing this tape half or configuration. Let $\mathsf{stringForTapeHalf} : \Sigma^* \to \Gamma^*$ and $\mathsf{stringForConfig} : Q \to \mathsf{tape}\ (\Sigma) \to \Gamma^*$ be such functions.

stringForTapeHalf

stringForConfig

5.3 Modifying Tapes

Recall that we need rewrite rules for shifting the tape and for encoding transitions at the center state symbol. In this section, we introduce the rewrite rules for shifting tape halves and prove the main results for manipulating the representation of a tape half. Thus, are only concerned with the representation of tape halves for now.

Tape Rules Recall that we need rules for shifting the tape to the left, to the right, or leaving its position unchanged. They use metavariables which can be instantiated with all possible values of the corresponding type. In the premise of the rules, we omit the polarity; they can be instantiated with each of the three polarities, as long as the polarity is consistent across the three symbols. The rules are annotated with the tape halves they are intended for. These annotations have no formal meaning but help with the intuition.

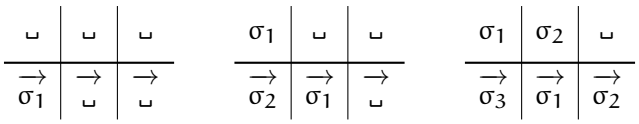
Right Shifts

σ_1	σ_2	σ_3
$\overrightarrow{\sigma_4}$	$\overrightarrow{\sigma_1}$	$\overrightarrow{\sigma_2}$

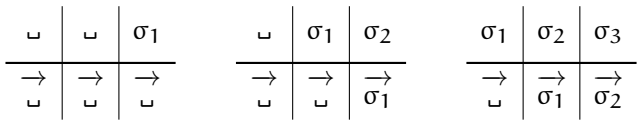
$\sqcup$	$\sqcup$	$\sqcup$
$\overrightarrow{\sqcup}$	$\overrightarrow{\sqcup}$	$\overrightarrow{\sqcup}$

(both halves)

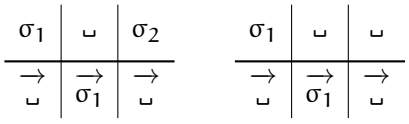
(right half)



(left half)

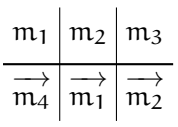


The rules implicitly encode the invariant that all symbols used by the Turing machine are placed contiguously with no blanks inbetween. For instance, we do not need the following two rules:



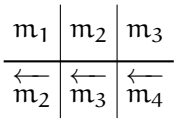
In the first case, the premise prevents the rule from ever being applicable. In the second case, the overlap of the rewrite windows and the state symbol which stands between both tape halves makes it impossible to have the substring $\sqcup\overrightarrow{\sigma_1}\sqcup$. We call such instantiations of a rewrite rule **spurious**.

Knowing this, we write down the above rules in a more succinct way as



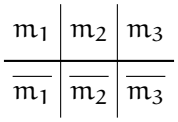
While we would be fine having the spurious rewrite windows, they do make additional reasoning necessary in some cases. Therefore, we just regard this as a notation for the expanded form above, instead of actually having the spurious instantitions. One can easily derive the rules which are actually relevant.

Left Shifts We only write down the abbreviated form:



Note that this rule exactly mirrors the rule for shifting the tape to the right.

Identity Rules



#	␣	␣
#	␣	␣

␣	␣	#
␣	␣	#

We need no windows which contain both the delimiter # and an element of Σ as the Turing machine (by construction of z) cannot use the two blanks adjacent to the delimiters within the number of steps it is given.

The collection of all windows generated by these rules is referred to as R_{tape}. $\boxed{\mathsf{R}_{\text{tape}}}$

Remark 5.18 *It now becomes clear why we need the delimiter #. Without it, symbols could just be introduced at the edge of the string. For instance, the rule*

␣	␣	␣
$\overrightarrow{\sigma_1}$	$\rightarrow$ ␣	$\rightarrow$ ␣

,

which is intended for use on the right tape half, could then also be used at the leftmost position of the left tape half as no rewrite window is overlapping from the left.

Lemma 5.19 (Symmetry of R_{tape}) *The R_{tape} rules are symmetric in the sense that they are closed under polarity-reversion of premise and conclusion.*

$$[\gamma_1,\gamma_2,\gamma_3] \,/\, [\gamma_4,\gamma_5,\gamma_6] \in \mathsf{R}_{tape} \leftrightarrow [\sim\gamma_3,\sim\gamma_2,\sim\gamma_1] \,/\, [\sim\gamma_6,\sim\gamma_5,\sim\gamma_4] \in \mathsf{R}_{tape}$$

Proof We first prove one direction and then use that pFlip is involutive (Fact 5.14). The proof is by inversion on the rule used to generate the window. The interesting case is the one for shifting the tape to the left or to the right, which follows by the fact that the rules for shifting to the left and shifting to the right are exactly symmetric. □

The following lemma will be extremely helpful in the sequel. If we want to prove a statement for the left and the right tape half, it allows us to just prove the statement for the right tape half and then get the symmetric result for the left tape half for free. This is useful in particular because we cannot do direct inductions over the reversed left tape half.

Lemma 5.20 (Symmetry of Tape Rewrites)

1. $\mathfrak{h} \rightsquigarrow_{\mathsf{R}_{tape}} \mathfrak{h}' \to \mathit{pRev}\,\mathfrak{h} \rightsquigarrow_{\mathsf{R}_{tape}} \mathit{pRev}\,\mathfrak{h}'$
2. $\mathit{pRev}\,\mathfrak{h} \rightsquigarrow_{\mathsf{R}_{tape}} \mathit{pRev}\,\mathfrak{h}' \to \mathfrak{h} \rightsquigarrow_{\mathsf{R}_{tape}} \mathfrak{h}'$
3. $\mathfrak{h} \rightsquigarrow_{\mathsf{R}_{tape}} \mathfrak{h}' \to [\ \sim x \mid x \in \mathfrak{h}\ ] \rightsquigarrow_{\mathsf{R}_{tape}} \mathfrak{h}'$

Proof

1. By induction on $\mathfrak{h} \rightsquigarrow_{R_{\text{tape}}} \mathfrak{h}'$. The first two cases are trivial. In the successor case, there is the problem that the new symbols are appended at the end of the strings, while the inductive definition of $\rightsquigarrow$ only allows to prepend new symbols. We switch to the explicit characterisation by Lemma 5.9 and do a case analysis on the position for which we need to provide a rewrite window. For all but the last position, we can apply the inductive hypothesis, while we use the polarity-reversed new rewrite window, which exists by Lemma 5.19, for the last position.
2. Apply 1. and use that pRev is an involution.
3. By induction on $\mathfrak{h} \rightsquigarrow_{R_{\text{tape}}} \mathfrak{h}'$ and inversion on the used rewrite rule. □

Manipulation of Tape Representations We prove several lemmas which are similar in flavour. They state that, starting with the representation of a tape half, we can leave the tape half unchanged, add a symbol of Σ, or remove a symbol that prefixes it. Moreover, if a tape half rewrites to a string whose first symbol is known, the rest of the string is also uniquely determined. Basically, they allow use to simulate the effect a single Turing machine transition has on the tape. The results are each proven for the right tape half and then derived for the left tape half by Lemma 5.20.

Remark 5.21 *It now becomes clear why we require at least three symbols on each side of the state symbol: this way, we can prove results about each of the tape halves individually without running into the problem of vacuous rewrites* (*Lemma 5.7*).

We start with a statement about empty tape halfs which will later be the base case of more general results.

Lemma 5.22 (Empty Tape Half: Blank Rewriting)

1. $\mathsf{E}\ p\ \hat{n} \rightsquigarrow_{R_{tape}} \mathsf{E}\ p'\ \hat{n}$ *and if* $\mathsf{E}\ p\ \hat{n} \rightsquigarrow_{R_{tape}} \sqcup^{p'} :: s$, *then* $s = \mathsf{E}\ p'\ (1 + n)$
2. $\mathsf{rev}\ (\mathsf{E}\ p\ \hat{n}) \rightsquigarrow_{R_{tape}} \mathsf{rev}\ (\mathsf{E}\ p\ \hat{n})$ *and* $\mathsf{rev}\ (\mathsf{E}\ p\ \hat{n}) \rightsquigarrow_{R_{tape}} \mathsf{rev}\ (\sqcup^{p'}) :: s) \to s = \mathsf{E}\ p'\ (1 + n)$

Proof

1. The first statement follows by induction on n. For the second one, we unfold $\hat{n} = 1 + (1 + n)$ and generalise $1 + n$ to arbitrary $n \geqslant 1$. The statement then follows by induction on n: the base case is contradictory. In the successor case, we do another case analysis on n. If $n = 0$, the statement holds by inversion on the used rewrite rule.

   Otherwise, $n = 1+n'$. We have $\mathsf{E}\ p\ (3+n') \rightsquigarrow \sqcup^{p'} :: s$ and show $s = \mathsf{E}\ p'\ (2+n)$. By inversion on the rewrite rule used at the head, we get four cases and in

each one, $s = \llcorner p' \lrcorner :: s'$ and $E\ p\ (2 + n) \rightsquigarrow \llcorner p' \lrcorner :: s'$. As $1 + n' \geqslant 1$, we apply the inductive hypothesis and get $s' = E\ p'\ (1 + n)$, closing the proof.

2. For the first statement, it suffices to show $\mathsf{rev}\ (E\ (\sim\sim p)\ \hat{n}) \rightsquigarrow_{R_{tape}} \mathsf{rev}\ (E\ (\sim\sim p)\ \hat{n})$. By Proposition 5.16.1, we can pull out one of the pFlips to turn the rev into a pRev. Applying Lemma 5.20.1, we are left with $[\ \sim\gamma \mid \gamma \in E\ p\ \hat{n}\ ] \rightsquigarrow_{R_{tape}} [\ \sim\gamma \mid \gamma \in E\ p'\ \hat{n}\ ]$. This follows by another application of Proposition 5.16.1 and part 1.

   The second part is proved in a similar way: we use that pFlip is involutive, apply Lemma 5.20.2, and use part 1. □

The important idea of Lemma 5.22 is that the rewrite is uniquely determined **once we know the first symbol** of the target string. All of the important lemmas in the remainder of this section will be similar in style. The two statements of Lemma 5.22 each have an existence and uniqueness part in addition to another property: there exists a unique s such that $E\ p\ \hat{n} \rightsquigarrow_{R_{tape}} (p', \llcorner \lrcorner) :: s$, and additionally, this s satisfies $s = E\ p'\ (1 + n)$. In the following, we write this down more succinctly using a **modified unique existence** quantifier:

$$\bar{\exists}!a, p\ a \wedge q\ a :\Leftrightarrow \exists a, p\ a \wedge (\forall b, p\ b \to b = a) \wedge q\ a. \qquad \boxed{\bar{\exists}!}$$

Thus, Lemma 5.22.1 now reads as follows:

$$\bar{\exists}!s, E\ p\ \hat{n} \rightsquigarrow_{R_{tape}} \llcorner p' \lrcorner :: s \wedge s = E\ p'\ (1 + n)$$

Remark 5.23 *Recall that we use rewrite windows of width 3. Choosing a width of 2 does in principle also work. A width of 3, however, makes some proofs easier. Consider again the uniqueness part of statement 1 of the previous lemma. The appropriate statement to prove would be:* $E\ p\ (1 + n) \rightsquigarrow \llcorner p' \lrcorner :: s \to s = E\ p'\ n$. *The corresponding strengthening would be to require* $n \geqslant 0$, *which is trivial. In the successor case, we know* $E\ p\ (2 + n) \rightsquigarrow \llcorner p' \lrcorner :: s$ *and we have to prove* $s = E\ p'\ (1 + n)$. *If we now do an inversion on the rewrite rule used at the head, we cannot, for instance, directly rule out that* $s = \overleftarrow{\sigma} :: s'$ *(by the rewrite rule* $[\llcorner, \llcorner] / [\overleftarrow{\llcorner}, \overleftarrow{\sigma}]$[2]*). In order to deal with cases like this, we would need to do a case analysis on* n *and then do more inversions on the rewrite rules at the next position.*

Rewrite windows of size 3 do not necessitate these additional inversions: they directly encode structural information such as $E\ p\ (3 + n) \not\rightsquigarrow \overleftarrow{\llcorner} :: \overleftarrow{\sigma} :: s'$. *We think, however, that using rewrite windows of size 2 would have been a valid design choice, too.*

We now prove results for arbitrary tape halves that need not be empty. Intuitively, the following three lemmas allow us to add a symbol of Σ to a string representing

[2]The modification of the rules to width 2 is straightforward.

a tape half, remove a symbol, or leave the tape half unchanged, thus enabling the tape shifts. The results are all proved by using induction on the tape half that is represented by the string and use the lemma for empty tape halves in the base case.

Lemma 5.24 (Adding a Symbol) *If* $rs \sim_t^{(n,p)} h$ *and* $|rs| < n$*, then*

1. $\exists! h', h \rightsquigarrow_{R_{tape}} \overrightarrow{\sigma} :: h' \wedge \sigma :: rs \sim_t^{(n,+)} \overrightarrow{\sigma} :: h'$
2. $\exists! h', \mathsf{rev}\, h \rightsquigarrow_{R_{tape}} \mathsf{rev}(\overleftarrow{\sigma} :: h') \wedge \sigma :: rs \sim_t^{(n,-)} \overleftarrow{\sigma} :: h'$

Proof We look at the first statement, the second statement is again derived by Lemma 5.20. The proof proceeds by induction on rs. In the base case, we choose $h' = E\,(+)\,(1{+}n)$, pick a suitable rewrite window at the head, and apply Lemma 5.22.

In the cons case, we have $\sigma_1 :: rs \sim_t^{(p,n)} h$, $1 + |rs| < n$. By inversion on $\sim_t^{(p,n)}$, we know $h = \sigma_1^p :: h_0$ and $n = 1 + n'$. We show

$$\exists! h', \sigma_1^p :: h_0 \rightsquigarrow \overrightarrow{\sigma} :: h' \wedge \sigma :: \sigma_1 :: rs \sim_t^{(p,1+n')} \overrightarrow{\sigma} :: h'$$

In order to determine a rewrite rule to apply at the head, we need to know two more symbols at the head of h_0. Thus we do a case analysis on rs: either $rs = [\,]$, $rs = [\sigma_2]$, or $rs = \sigma_2 :: \sigma_3 :: rs'$. The interesting case is the third one, for which we need the inductive hypothesis.

By inversion on $\sim_t^{(1+n',p)}$, we get $h_0 = \sigma_2^p :: \sigma_3^p :: h_1$ and $n' = 2 + n_0$, with $\sigma_1 :: \sigma_2 :: \sigma_3 :: rs' \sim_t^{(3+n_0,p)} \sigma_1^p :: \sigma_2^p :: \sigma_3^p :: h_1$. Thus, $\sigma_2 :: \sigma_3 :: rs' \sim_t^{(2+n_0,p)} \sigma_2^p :: \sigma_3^p :: h_1$, to which the inductive hypothesis can be applied for $\sigma := \sigma_1$ to obtain a unique h' with $\sigma_2 :: \sigma_3 :: h_1 \rightsquigarrow \overrightarrow{\sigma_1} :: h'$ and $\sigma_1 :: \sigma_2 :: \sigma_3 :: rs' \sim_t^{(2+n_0,+)} \overrightarrow{\sigma_1} :: h'$. By inversion on $\sim_t^{(2+n_0,+)}$, we have $h' = \overrightarrow{\sigma_2} :: \overrightarrow{\sigma_3} :: h_2$.

By using the rewrite window $[(p,\sigma_1),(p,\sigma_2),(p,\sigma_3)]$ / $[\overrightarrow{\sigma}, \overrightarrow{\sigma_1}, \overrightarrow{\sigma_2}]$, we directly get existence. Uniqueness follows by inversion on the rewrite at the head of the string and by using the uniqueness from the inductive hypothesis. □

Lemma 5.25 (Removing a Symbol) *If* $\sigma :: rs \sim_t^{(n,p)} \sigma^p :: m^p :: h$*, then*

1. $\exists! h', \sigma^p :: m^p :: h \rightsquigarrow_{R_{tape}} \overleftarrow{m} :: h' \wedge rs \sim_t^{(n,-)} \overleftarrow{m} :: h'$
2. $\exists! h', \mathsf{rev}(\sigma^p :: m^p :: h) \rightsquigarrow_{R_{tape}} \mathsf{rev}(\overrightarrow{m} :: h') \wedge rs \sim_t^{(n,+)} \overrightarrow{m} :: h'$

Lemma 5.26 (Leaving the Tape Unchanged) *If* $rs \sim_t^{(n,p)} m^p :: h$*, then*

1. $\exists! h', m^p :: h \rightsquigarrow_{R_{tape}} \overline{m} :: h' \wedge rs \sim_t^{(n,o)} \overline{m} :: h'$
2. $\exists! h', \mathsf{rev}(m^p :: h) \rightsquigarrow_{R_{tape}} \mathsf{rev}(\overline{m} :: h') \wedge rs \sim_t^{(n,o)} \overline{m} :: h'$

5.4 Encoding Transitions

In this section, we deal with the Turing machine's transition function. We introduce new rewrite rules for the different cases of the transition function and differentiate between halting and non-halting configurations. The resulting windows can be applied at the center of the configuration string at the three positions involving the state symbol. Thus, we have three rules for each transition of the Turing machine: one for each of the cases where the state symbol is in the left, center, or right cell of the rewrite window. The state symbol uniquely determines the successor string to which a configuration string can be rewritten. The flow of information for the rewrite is from the state symbol at the center to the outer regions of the tape halves: the rewrite rules involving the state symbol uniquely determine the new head of the tape halves. Then, the results of the previous section yield unique successor tape strings.

Transition Rules We first present the rules for the case that the Turing machine is not in a halting configuration, i.e. $\mathsf{halt}\ q = \mathsf{F}$. As we are working with single-tape Turing machines, we simplify the type of the transition function δ to $Q \times \mathbb{O}(\Sigma) \to Q \times \mathsf{Act}_\Sigma$ in the following presentation, omitting the singleton vector wrappers. We make the following observations regarding the Turing machine's behaviour on a transition $\delta(q, m) = (p, m', a)$:

m	state symbol	m'	written symbol
$^\circ\sigma$	q^σ	$^\circ\sigma'$	σ'
		$\emptyset$	σ
$\emptyset$	$q^\sqcup$	$^\circ\sigma'$	σ'
		$\emptyset$	/

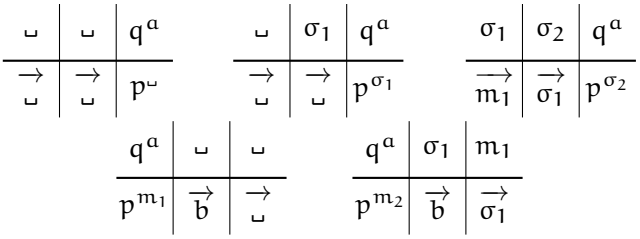
Even if $m' = \emptyset$, we can interpret that as the Turing machine just writing the current symbol again if the head is currently on a symbol ($m \neq \emptyset$). However, if the symbol currently under the head is a blank ($m = \emptyset$) and the machine does not write a new symbol ($m' = \emptyset$), no symbol is written. Thus, the rewrite windows for all of the cases except for the one where $m = \emptyset$ and $m' = \emptyset$ look very similar. This last case needs to be handled differently: if the transition function dictates to move in a direction where the next symbol is a blank again, the tape must not be shifted, which resembles the definition of head movement in Figure 2.1 on Page 10.

We start with the cases where $m, m' \neq \emptyset$.

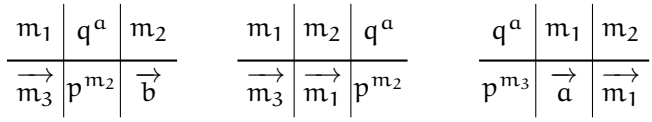
$\delta(q, {}^\circ a) = (p, {}^\circ b, L)$**:**

$\sqcup$	q^a	m_1
$\overrightarrow{\sqcup}$	$p^\sqcup$	$\overrightarrow{b}$

σ_1	q^a	m_1
$\overrightarrow{m_2}$	p^{σ_1}	$\overrightarrow{b}$

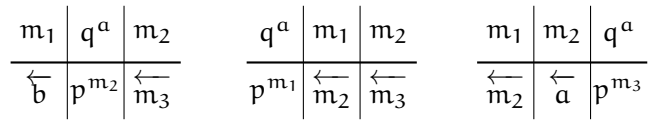


We have three main cases, one each for the three possible positions of the state symbol. Note that we again distinguish blanks and elements of Σ and thus directly encode the invariant that the used region of the configuration string is contiguous. As for the tape rules, we can write the rules down more succinctly as

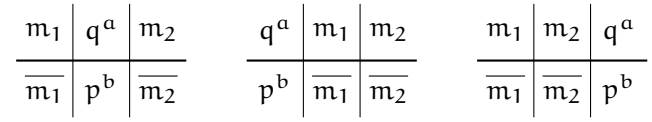


These rules do again generate spurious rewrite windows that will never be applied when starting with a valid configuration string. Since the tape rules enforce consistent tape shifts, the spurious windows do no harm; however, for simplicity, we will still work with the full definition without spurious windows in the following. From the shortened definition of the rules, one can easily restore the rules for the non-spurious windows.

$\delta(q, {}^\circ a) = (p, {}^\circ b, R)$**:**

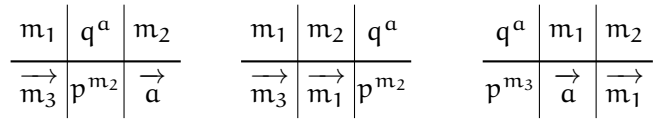


$\delta(q, {}^\circ a) = (p, {}^\circ b, N)$**:**



For the other cases, we only give the rules where the tape is shifted to the right. The other rules are derived similarly and can be found in Appendix B. The rules where $m' = \emptyset$ are very similar to the previous ones: instead of writing a new symbol b, the current symbol a is written again.

$\delta(q, {}^\circ a) = (p, \emptyset, L)$**:**



If $m = \emptyset$, the current symbol under the head is a blank:

$\delta(q, \emptyset) = (p, {}^{\circ}b, L)$:

$$\frac{m_1 \mid q^{\sqcup} \mid m_2}{\overrightarrow{m_3} \mid p^{m_2} \mid \overrightarrow{b}} \qquad \frac{m_1 \mid m_2 \mid q^{\sqcup}}{\overrightarrow{m_3} \mid \overrightarrow{m_1} \mid p^{m_2}} \qquad \frac{q^{\sqcup} \mid m_1 \mid m_2}{p^{m_3} \mid \overrightarrow{a} \mid \overrightarrow{m_1}}$$

The last case where the current symbol is a blank and the Turing machine does not write any symbol is different, as noted above. Therefore, the distinction between elements of Σ and the blank $\sqcup$ is important. We give the full rules instead of the abbreviation.

$\delta(q, \emptyset) = (p, \emptyset, L)$:

$$\frac{\sqcup \mid q^{\sqcup} \mid m_1}{\overline{\sqcup} \mid p^{\sqcup} \mid \overline{m_1}} \qquad \frac{\sigma_1 \mid q^{\sqcup} \mid \sqcup}{\overrightarrow{m_1} \mid p^{\sigma_1} \mid \sqcup}$$

$$\frac{\sqcup \mid \sqcup \mid q^{\sqcup}}{\overline{\sqcup} \mid \overline{\sqcup} \mid p^{\sqcup}} \qquad \frac{\sqcup \mid \sigma_1 \mid q^{\sqcup}}{\overrightarrow{\sqcup} \mid \overrightarrow{\sqcup} \mid p^{\sigma_1}} \qquad \frac{\sigma_2 \mid \sigma_1 \mid q^{\sqcup}}{\overrightarrow{m_1} \mid \overrightarrow{\sigma_2} \mid p^{\sigma_1}}$$

$$\frac{q^{\sqcup} \mid \sqcup \mid \sqcup}{p^{m_1} \mid \overrightarrow{\sqcup} \mid \overrightarrow{\sqcup}} \qquad \frac{q^{\sqcup} \mid \sigma_1 \mid m_1}{p^{\sqcup} \mid \overline{\sigma_1} \mid \overline{m_1}}$$

Let R_{trans} denote the collection of all windows generated by the transition rules. R_{trans}

Halting Extensions Now we deal with halting configurations, i.e. configurations with states q for which $\mathsf{halt}\ q = \mathsf{T}$. Recall that the tableau will have a fixed a number of lines and therefore, rewrites need to be possible even for halting configurations in case the Turing machine halts early. We solve this problem by adding rewrite rules that allow to rewrite halting configurations to exactly the same configuration. Again, we give a simplified definition generating spurious windows.

$$\frac{m_1 \mid q^{m_2} \mid m_3}{\overline{m_1} \mid q^{m_2} \mid \overline{m_3}} \qquad \frac{m_1 \mid m_2 \mid q^{m_3}}{\overline{m_1} \mid \overline{m_2} \mid q^{m_3}} \qquad \frac{q^{m_1} \mid m_2 \mid m_3}{q^{m_1} \mid \overline{m_2} \mid \overline{m_3}}$$

Let R_{halt} denote the collection of all windows generated by the halting rules. R_{halt}

Composition of Rewrite Windows We define the set of rewrite windows R_{sim} for the deterministic simulation of the Turing machine as $R_{sim} \coloneqq R_{tape} \# R_{trans} \# R_{halt}$. Next, we prove basic facts about the composition of the three sets of windows. R_{sim}

Definition 5.27 (State Symbols) *stateSym* $q\ \gamma := \exists m, \gamma = q^m$

The following lemma is based on the fact that a string representing a tape half cannot contain a state symbol. It helps us in lifting the results of the previous section

for tapes to $\mathsf{R}_{\mathrm{sim}}$.

Lemma 5.28 (Tape Rewriting)

1. $u \sim_t^{(n,p)} h \to (\exists w \in \mathsf{R}_{sim}, \mathit{rewHead}\ w\ h\ h') \to (\exists w \in \mathsf{R}_{tape}, \mathit{rewHead}\ w\ h\ h')$
2. $u \sim_t^{(n,p)} h \to h \rightsquigarrow_{\mathsf{R}_{sim}} h' \to h \rightsquigarrow_{\mathsf{R}_{tape}} h'$
3. $u \sim_t^{(n,p)} h \to (\mathit{pRev}\ h) \rightsquigarrow_{\mathsf{R}_{sim}} (\mathit{pRev}\ h') \to (\mathit{pRev}\ h) \rightsquigarrow_{\mathsf{R}_{tape}} (\mathit{pRev}\ h')$

Proof 1. From the simple fact that all windows contained in $\mathsf{R}_{\mathrm{tape}}$ or $\mathsf{R}_{\mathrm{halt}}$ contain a state symbol, i.e. a symbol γ satisfying stateSym $q\ \gamma$ for some q, while a string representing a tape half can never contain a state symbol.

2. By induction on $h \rightsquigarrow_{\mathsf{R}_{\mathrm{sim}}} h'$, using 1. in the successor case. In order to apply the inductive hypothesis, we invert the representation relation.

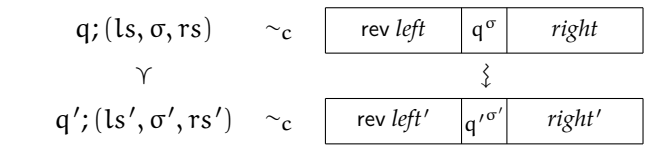
3. An induction on $\rightsquigarrow_{\mathsf{R}_{\mathrm{sim}}}$ fails: the tape representation relation does not reverse h, thus we cannot suitably invert the representation relation in order to apply the inductive hypothesis. Instead, we switch to the explicit characterisation by Lemma 5.9. □

Lemma 5.29 (State Rules) *Let* $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6 : \Gamma$ *and* $\mathit{stateSym}\ q\ \gamma_1 \vee \mathit{stateSym}\ q\ \gamma_2 \vee \mathit{stateSym}\ q\ \gamma_3$, $w := [\gamma_1, \gamma_2, \gamma_3]\ /\ [\gamma_4, \gamma_5, \gamma_6]$. *Then:*

1. $\mathit{halt}\ q = \mathsf{F} \to w \in \mathsf{R}_{sim} \to w \in \mathsf{R}_{trans}$
2. $\mathit{halt}\ q = \mathsf{T} \to w \in \mathsf{R}_{sim} \to w \in \mathsf{R}_{halt}$

5.4.1 Single Simulation Steps

We prove that, if we start out with a configuration string, a single rewrite step using the rewrite windows $\mathsf{R}_{\mathrm{sim}}$ exactly corresponds to a step of the Turing machine if the configuration is not halting and otherwise, the rewrites just replicate the current halting configuration. Pictorially, we prove the following diagram for the case of non-halting configurations:



where $ls \sim_t^p \mathit{left}, rs \sim_t^p \mathit{right}$ and $ls' \sim_t^{p'} \mathit{left}', rs' \sim_t^{p'} \mathit{right}'$. From the existence of a $\succ$ or $\rightsquigarrow$ transition, we get the unique existence of the other transition.

We start off with an important lemma that allows us to split up a rewrite $h \rightsquigarrow h'$ into three rewrites involving the center state symbol and a rewrite each for the two tape halves.

Proposition 5.30

$$\begin{aligned}
&(A +\!\!+ [c, d, e, f, g] +\!\!+ B) \rightsquigarrow_R (A' +\!\!+ [c', d', e', f', g'] +\!\!+ B') \wedge |A| = |A'| \wedge |B| = |B'| \\
\leftrightarrow \quad &(A +\!\!+ [c, d] \rightsquigarrow_R A' +\!\!+ [c', d']) \\
&\wedge([f, g] +\!\!+ B) \rightsquigarrow_R ([f', g'] +\!\!+ B') \\
&\wedge [c, d, e] \,/\, [c', d', e'] \in R \wedge [d, e, f] \,/\, [d', e', f'] \in R \wedge [e, f, g] \,/\, [e', f', g'] \in R
\end{aligned}$$

Conversely, if we start out with a string of the form given in the previous proposition and rewrite to another string, the new string can also be split up accordingly.

Proposition 5.31

$$\begin{aligned}
&(A +\!\!+ [a, b, c, d, e] +\!\!+ B) \rightsquigarrow s' \\
\rightarrow\ &\exists A'\ B'\ a'\ b'\ c'\ d'\ e', s = A' +\!\!+ [a', b', c', d', e'] +\!\!+ B' \wedge |A| = |A'| \wedge |B| = |B'|
\end{aligned}$$

The following two results are the main correctness statements for the Turing machine simulation.

Lemma 5.32 (Step Simulation)
If $(q, tp) \sim_c s$, $(q, tp) \succ (q', tp')$ *and* $|tp| < z$, *then* $\exists! s', s \rightsquigarrow_{R_{sim}} s' \wedge (q', tp') \sim_c s'$.

Proof The proof follows the following idea which was already outlined previously:

left	h_l	q^m	h_r	*right*
$\exists!$*left*$'$	h_l'	$q'^{m'}$	h_r'	$\exists!$*right*$'$

Given a configuration string, we split it into the state symbol q^m, the left tape half, and the right tape half. The state symbol uniquely determines the transition the Turing machine takes. By analysing the heads h_l and h_r of the tape halves, we can find out the rewrite windows to use at the center. These windows then uniquely determine the heads h_l' and h_r' of the successor tape halves. Using Lemmas 5.24-5.26, the successor tape halves are therefore fully determined. The rewrites can be justified seperately by Lemma 5.30.

We have to do several case analyses on the transition that is taken and the shape of the left and right tape halves. As an example, we look at the case where $m = {}^\circ\sigma$ and $\delta(q, {}^\circ\sigma) = (q', {}^\circ\sigma', L)$, $\mathsf{left}\ tp = \mathsf{rev}(\sigma_1 :: \sigma_2 :: \mathtt{left}')$, $\mathsf{right}\ tp = \sigma_3 :: \mathtt{right}'$. By assumption, there are $h_1 = \sigma_1^p :: \sigma_2^p :: h_1'$, $h_2 = \sigma_3^p :: h_2'$ with $\sigma_1 :: \sigma_2 :: \mathtt{left}' \sim_t^{(z,p)} h_1$ and $\sigma_3 :: \mathtt{right}' \sim_t^{(z,p)} h_2$. Pictorially, the tape looks as follows:

left′	σ_2^p	σ_1^p	q^σ	σ_3^p	right′

Thus, we apply the window

σ_1^p	q^σ	σ_3^p
$\overrightarrow{\sigma_2}$	q'^{σ_1}	$\overrightarrow{\sigma'}$

at the center. Doing one more case analysis for the left and right tape halves, we can determine the other two rewrite windows to use at the center.

Next, we transform the two halves of the tape. The element σ_1 needs to be removed from the left tape half, while σ' needs to be added to the right tape half. By Lemma 5.25, there is a unique h_1'' such that

$$\mathsf{rev}(\sigma_1^p :: \sigma_2^p :: h_1) \rightsquigarrow \mathsf{rev}(\overleftarrow{\sigma_2} :: h_1'') \text{ and } \sigma_2 :: \mathtt{left}' \sim_t^{(z,+)} \overrightarrow{\sigma_2} :: h_1''.$$

Similarly, by Lemma 5.24, there is a unique h_2'' such that

$$\sigma_3^p :: h_2' \rightsquigarrow \overrightarrow{\sigma'} :: \overrightarrow{\sigma_3} :: h_2'' \text{ and } \sigma' :: \sigma_3 :: \mathtt{right}' \sim_t^{(z,+)} \overrightarrow{\sigma'} :: \overrightarrow{\sigma_3} :: h_2''.$$

These two results can be used to justify the rewrite for the two tape halves. Moreover, we have to show that

$$(q', tp') \sim_c \mathsf{rev}(\overrightarrow{\sigma_2} :: h_1'') +\!\!+ [q'^{\sigma_1}] +\!\!+ (\overrightarrow{\sigma'} :: \overrightarrow{\sigma_3} :: h_2''),$$

which is straightforward, as tp' results from tp by writing σ' and moving the head to the left, i.e. $tp' = \mathsf{midtape}\ (\sigma_2 :: \mathtt{left}')\ \sigma_1\ (\sigma' :: \sigma_3 :: \mathtt{right}')$.

Finally, we prove that the rewrite is unique. Assuming that the current configuration string rewrites to s', we can also split up s' along the state symbol by Proposition 5.31. We do an inversion on the rewrite windows used at the center, applying Lemmas 5.28 and 5.29 in the process, and then use the uniqueness for both tape halves. □

Lemma 5.33 (Halting Simulation)
If $(q, tp) \sim_c s$, *halt* $q = \mathsf{T}$, *then* $\exists! s', s \rightsquigarrow_{R_{sim}} s' \wedge (q, tp) \sim_c s'$.

Proof Using a similar style of arguments as for Lemma 5.32, but simpler, as we do not have to handle all the cases of the transition function. □

5.5 Deterministic Simulation

We extend the results of the previous section to cover multiple simulation steps and define a set of final substrings. In the end, we obtain that, starting from any fixed input, the resulting **PR** instance does exactly simulate the Turing machine.

5.5.1 Multi-step Simulation

If there is enough space left on the tape, the rewrite windows R_{sim} can exactly act like the Turing machine.

Lemma 5.34 (Multi-step Completeness) *Let* $(q, tp) \sim_c s$, $(q, tp) \succ^i (q', tp')$ *for some* $z \geqslant i \geqslant 0$, *and* $|tp| \leqslant z - i$, *then* $\exists! s', s \rightsquigarrow^i_{R_{sim}} s' \wedge (q', tp') \sim_c s'$.

Proof By induction on $(q, tp) \succ^i (q', tp')$, using Lemma 5.32 in the successor case. □

Remark 5.35 *At this point, our choice to have a fixed state symbol (head) position finally pays off. If we had chosen to use a moving-head semantics, we would now need an additional invariant stating that the state symbol is at least* i *symbols away from the border of the string. This would need to be built-in into the representation relations, requiring an additional parameter for* $\sim_t$ *and* $\sim_c$ *determining the amount of available space. The requirement* $|tp| \leqslant z - i$ *which only talks about the tape that is represented is much simpler.*

Lemma 5.36 (Multi-step Halting) *If* $(q, tp) \sim_c s$, *halt* $q = \mathsf{T}$, *then* $\exists! s', s \rightsquigarrow^i_{R_{sim}} s' \wedge (q, tp) \sim_c s'$.

Lemma 5.37 (Multi-step Soundness) *If* $(q, tp) \sim_c s, i \leqslant z, |tp| \leqslant z - i$, *and* $s \rightsquigarrow^i_{R_{sim}} s'$, *then there exist* q', tp' *and* j *with*

- $(q', tp') \sim_c s'$,
- $j \leqslant i$,
- $(q, tp) \succ^j (q', tp')$,
- *and* $|tp'| \leqslant |tp| + j$.

Proof Logically, a new line is appended to the tableau of configurations with each rewrite step. Thus, a direct induction over $s \rightsquigarrow^i_{R_{sim}} s'$ fails: the definition of $\rightsquigarrow^i_{R_{sim}}$ prepends a new line with each step. By Proposition 2.1, we switch to ${}^i\!\rightsquigarrow_{R_{sim}}$. Now, the induction goes through. In the successor case, we make a case analysis on halt q (which is also the reason why the naive induction fails). □

5.5.2 Soundness and Completeness

Next, we deal with the final constraint of ending in a halting state and fix the full **PR** instance. We do not directly define the set of final substrings according to the definition of **PR**, but instead work with a more abstract notion for now.

Definition 5.38 (Halting String) *haltingString* $s \coloneqq \exists p\ m, p^m \in s \wedge$ *halt* $p = \mathsf{T}$

Fact 5.39 *If* $(q, tp) \sim_c s$, *then* *halt* $q = \mathsf{T}$ *if, and only if,* *haltingString* s.

We now obtain soundness and completeness of the full simulation.

Theorem 5.40 (Completeness) *If* $|tp| \leqslant k$, $(q, tp) \sim_c s$, *and* $(q, tp) \rhd^{\leqslant t} (q', tp')$, *then there is* s' *with* $s \rightsquigarrow^{t} s'$, $(q', tp') \sim_c s'$ *and* *haltingString* s'.

Proof By Lemmas 5.34, 5.36 and Fact 5.39. □

Theorem 5.41 (Soundness) *If* $(q, tp) \sim_c s$, $|tp| \leqslant k$, $s \rightsquigarrow^{t} s'$, *and* *haltingString* s', *then there are* q', tp' *with* $(q', tp') \sim_c s'$, $(q, tp) \rhd^{\leqslant t} (q', tp')$ *and* $|tp'| \leqslant z$.

Proof By Lemmas 5.37 and Fact 5.39. □

Next, we define valid initial strings and concretise the final substrings. For the definition of initial strings, we use the notion of valid initial tapes introduced in Definition 5.1 on Page 38.

Definition 5.42 (Initial Strings)

initialString
isInitialString

$$\textit{initialString}\ c \coloneqq \textit{stringForConfig}\ q_0\ (\textit{initTape}\ (in \mathbin{+\!\!+} c))$$
$$\textit{isInitialString}\ s \coloneqq \exists s', s = \textit{initialString}\ s' \wedge \textit{validCert}\ s'$$

$R_{\textit{final}}$

Definition 5.43 (Final Substrings) $R_{\textit{final}} \coloneqq [\ q^{m} \mid \textit{halt}\ q = \top\ ]$

Proposition 5.44 $s \models R_{\textit{final}} \leftrightarrow \textit{haltingString}\ s$

We have now reduced **TMGenNP** to the following question:

Problem 5.45
Does there exist a string s *with* *isInitialString* s *such that* **PR** $(\Gamma, 1, 3, s, R_{\text{sim}}, R_{\text{final}}, t)$ *holds?*

5.6 Interlude: Nondeterministic Preludes

In this section, we give a recipe to reduce existential questions as in Problem 5.45, where the initial string of a **PR** instance is unknown, to full **PR** instances with a fixed initial string. While all rewrite windows seen so far only allowed for deterministic rewriting on configuration strings, the key is to make the rewrite windows nondeterministic. The results presented here can also be seen as providing a (very limited) form of compositionality for **PR**.

The construction works by adding new rewrite windows and a new initial string, which together form a **prelude** to the given **PR** instance. The prelude generates an initial string to the original instance. Of course, we have to make sure that the new windows do not interfere with the "old" ones. To a large part, this can be ensured syntactically by expanding the alphabet. Additionally, we require the rewrite windows to produce a string of the old alphabet in exactly t' steps, which we call the number of **prelude steps**.

We present the results for the special case of 3-**PR**, albeit they transfer directly to the more general setting.

The following definition generalises the question stated by Problem 5.45.

Definition 5.46 (Existential 3-PR (Ex3PR)) *Given a 3-***PR** instance $(\Gamma, \mathfrak{l}, R, R_{\text{final}}, t)$ missing an input, where $\mathfrak{l}$ denotes the desired input length, and a predicate $p : \mathcal{L}(\Gamma) \to \mathbb{P}$, **Ex3PR** is defined as follows:

Ex3PR

$$\textbf{Ex3PR}\ (\Gamma, \mathfrak{l}, R, R_{\text{final}}, t)\ p \coloneqq \exists s, |s| = \mathfrak{l} \wedge p\ \mathfrak{l} \wedge \textbf{PR}\ (\Gamma, 1, 3, s, R, R_{\text{final}}, t)$$

Let us fix an **Ex3PR** instance $S = (\Gamma, \mathfrak{l}, R, R_{\text{final}}, t)$ over input predicate p. Moreover, let a **prelude alphabet** Δ : finType, a list of prelude windows $R' : \mathcal{L}((\text{window}\ \Delta\ (\Gamma + \Delta)))$, a number of prelude steps t', and an initial string $x_0 : \mathcal{L}(\Delta)$ be given. Collectively, we refer to (Δ, R', t', x_0) as a **prelude**. Here, the new notation window Δ $(\Gamma+\Delta)$ means that the premises of the windows are over type Δ, while the conclusions are over type $\Gamma + \Delta$. The latter is necessary because the prelude should eventually generate an input for S, while the new windows should not be applicable to strings of the original instance after the prelude has finished.

The alphabet of the new **PR** instance will be $\mathcal{A} \coloneqq \Gamma + \Delta$.

$\mathcal{A}$

Throughout this section, we implicitly lift elements of Γ and Δ to $\mathcal{A}$, the same applies to strings and windows over these alphabets. Where it is needed, we use the predicates origString, preludeString : $\mathcal{L}(\mathcal{A}) \to \mathbb{P}$ to distinguish strings over Γ and Δ.

origString

preludeString

Assumption 5.47 (Structural Assumptions on the Prelude) *We place the following assumptions on* (Δ, R', t', x_0)*:*

(A_0) $\mathfrak{l} \geqslant 3$

(A_1) $\forall x_0', x_0 \rightsquigarrow_{R'}^{t'} x_0' \to \mathit{origString}\ x_0'$

(A_2) $\forall k\ x, k < t' \to x_0 \rightsquigarrow_{R'}^{k} x \to \mathit{preludeString}\ x$

(A_3)**, Completeness** $\forall x_0', |x_0'| = \mathfrak{l} \wedge p\ x_0' \to x_0 \rightsquigarrow_{R'}^{t'} x_0'$

(A_4)**, Soundness** $\forall x_0', x_0 \rightsquigarrow_{R'}^{t'} x_0' \to p\ x_0'$

(A_5)**, Compatibility** $|x_0| = \mathfrak{l}$

Assumption (A_0) is required to avoid vacuous rewriting. Assumptions (A_1) and (A_2) together express that the initial string x_0 rewrites in exactly t' steps to a string to which the original windows R can be applied. Moreover, assumptions (A_3) and (A_4) ensure that the prelude can generate exactly those strings described by the predicate p as an input to S.

Now, we construct the new **PR** instance S' over alphabet $\mathcal{A}$. We define:

R_{comb}

$$S' \coloneqq (\mathcal{A}, 1, 3, x_0, R_{comb}, R_{final}, t + t') \text{ with } R_{comb} \coloneqq R \uplus R'$$

S' witnesses the reduction of **Ex3PR** to **PR**: the main goal of this section is to show that **Ex3PR** $p\ S \leftrightarrow$ **PR** S'. From an abstract perspective, the proof is quite simple. Most of the formal work is due to the handling of the different alphabets and the injections into the combined alphabet.

We therefore only state the most important intermediate result. Intuitively, this allows us to split a sequence of rewrites in the combined instance S' into t' rewrites due to the prelude and t rewrites due to the original instance S.

Lemma 5.48 *If* $x_0 \rightsquigarrow^{t'+t}_{R_{comb}} s$, *then there exists* x'_0 *with* $x_0 \rightsquigarrow^{t'}_{R'} x'_0$, $x'_0 \rightsquigarrow^{t}_{R} s$, *and* origString s.

Proof For the proof, we use the following simple facts:

1). If $n \leqslant t'$ and $x_0 \rightsquigarrow^{n}_{R_{comb}} b$, then $x_0 \rightsquigarrow^{n}_{R'} b$.

2). If $|s| \geqslant 3$, origString s, and $s \rightsquigarrow^{n}_{R_{comb}} b$, then $s \rightsquigarrow^{n}_{R} b$ and origString b.

We apply additivity of $\rightsquigarrow^{t+t'}$ (Proposition 2.1) to the assumption and choose the resulting x'_0. It is known that $x_0 \rightsquigarrow^{t'}_{R_{comb}} x'_0$ and $x'_0 \rightsquigarrow^{t}_{R_{comb}} s$. By 1), $x_0 \rightsquigarrow^{t'}_{R'} x'_0$, which shows the first part of the goal. Applying (A_1), we get origString x'_0. The proof is closed using 2). □

The main result is now a straightforward consequence.

Theorem 5.49 ***Ex3PR*** $S\ p \leftrightarrow$ **PR** S'

Proof Using Lemma 5.48 and the assumptions. □

Remark 5.50 *We can generalise the above setup a bit by replacing the fixed initial prelude string* x_0 *with another predicate stating which prelude strings are valid. This way, one can reduce an* ***Ex3PR*** *instance to another* (*possibly simpler*) ***Ex3PR*** *instance by adding a prelude. This may be interpreted as providing a* (*very limited*) *form of compositionality for* **PR**. In the mechanisation, we prove this more general statement, but we do not need it for the Turing machine simulation.

5.7 Guessing the Certificate

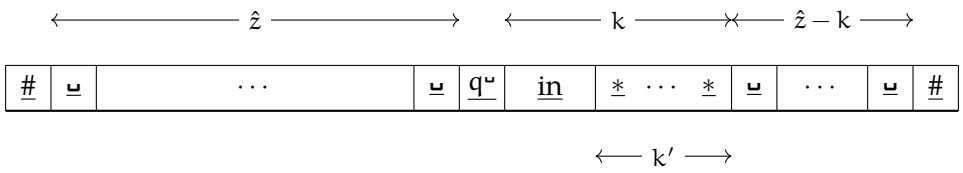
In this section, we apply the technique presented in Section 5.6 to "guess" an accepted certificate and thus generate a full input for the Turing machine. To that end, we design an initial string and nondeterministic rewrite rules that generate an input in a single rewrite step.

The alphabet Δ of the prelude has $4 + |\Sigma|$ elements, where $q_0 \coloneqq \mathsf{start}$ is the initial state. $\boxed{\Delta}$

$$\Delta \coloneqq \underline{\sqcup} \mid \underline{*} \mid \underline{q_0^{\sqcup}} \mid \underline{\#} \mid \underline{\sigma} \qquad \sigma : \Sigma$$

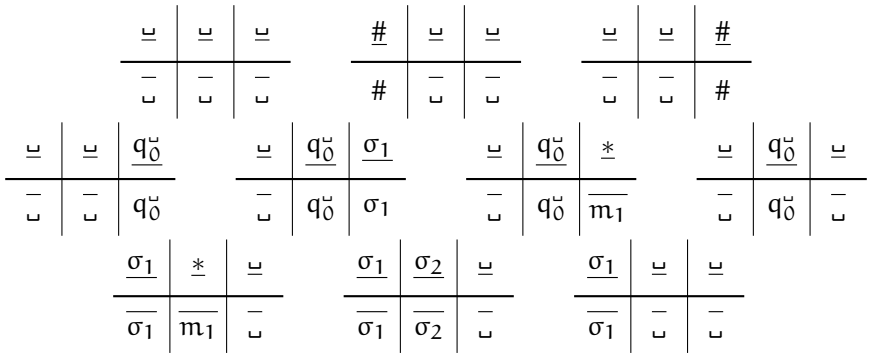
The symbols $\underline{\sqcup}$ and $\underline{\#}$ just mirror the corresponding symbols of Γ. $\underline{q_0^{\sqcup}}$ is the fixed initial state symbol, the $\underline{*}$ symbols are wildcards that will be replaced by the certificate and the $\underline{\sigma}$ are used for the fixed input. As in the previous section, we implicitly lift Δ and Γ to the combined alphabet $\mathcal{A} \coloneqq \Gamma + \Delta$. This requires some additional work for the formal proofs which we omit on paper. $\boxed{\mathcal{A}}$

As in Section 4.3, the initial string has the following form, where $\underline{in} = [\ \underline{\sigma} \mid \sigma \in in\ ]$:



Let $\mathsf{initStr} : \mathcal{L}(\Delta)$ be this initial string. $\boxed{\mathsf{initStr}}$

For the rewrite windows, we have to ensure that all elements of Σ are placed contiguously and that certificates of length less than k' are allowed, too. Exploiting the fact that the rewrite windows overlap, this is easy to achieve. The following rules are a subset of the needed ones, with the full set of rules being given in Appendix B.



The windows enforce that, if we decide to replace a $\underline{*}$ with a blank instead of an element of Σ, all $\underline{*}$s to the right of it are also replaced by a blank. Let $\mathsf{R}_{\mathsf{prelude}}$ denote the set of prelude rewrite windows. $\boxed{\mathsf{R}_{\mathsf{prelude}}}$

We prove that these rewrite windows exactly produce valid initial strings in a single rewrite step. It is not necessary to split up the string along the state symbol as before since it does not take any special role in the prelude. Instead, the individual components of the initial string can just be stacked upon each other from right to left, proving soundness and completeness in each step. We need several rather

unspectacular inductions for that. This gives us soundness (A_4) and completeness (A_3) of the prelude.

Lemma 5.51 (Soundness and Completeness)

1. $\mathit{isInitialString}\ s \wedge |s| = l \to \mathit{initStr} \rightsquigarrow_{R_{\mathit{prelude}}} s$
2. $\mathit{initStr} \rightsquigarrow_{R_{\mathit{prelude}}} s \to \mathit{isInitialString}\ s$

Applying Theorem 5.49, we have now arrived at the full reduction of **TMGenNP** to 3-**PR**.

Theorem 5.52 (TMGenNP reduces to PR)

$$\textbf{TMGenNP}\ (\Sigma, M, k, t) \leftrightarrow \textbf{PR}\ (\mathcal{A}, 1, 3, \mathsf{initStr}, R_{\mathsf{prelude}} \uplus R_{\mathsf{sim}}, R_{\mathsf{final}}, 1 + t)$$

Proof We have previously reduced **TMGenNP** to Problem 5.45. Theorem 5.49 for $t' = 1$ allows us to use the prelude we just constructed to reduce to the **PR** instance. It remains to show assumptions $(A_0) - (A_5)$. The structural requirements (A_0) and (A_5) on the initial prelude string hold by construction. Completeness (A_3) and Soundness (A_4) follow from Lemma 5.51. (A_2) holds trivially as $k < 1$ implies $k = 0$. Finally, (A_1), i.e. $\mathsf{initStr} \rightsquigarrow_{R_{\mathsf{prelude}}} x_0' \to \mathsf{origString}\ x_0'$, follows by an easy induction on $\rightsquigarrow_{R_{\mathsf{prelude}}}$ due to the construction of the rewrite windows. □

5.8 Mechanisation

To close this chapter, we comment on the design choices for the Coq mechanisation and in particular the differences to the presentation on paper.

5.8.1 PR

First of all, we impose the syntactic constraints (like $\omega > 0$) on **PR** instances externally, using a predicate PR_wellformed. Similarly, we do not model windows as elements of type $\Sigma^\omega \times \Sigma^\omega$ using vectors[3], but instead use lists: $\mathcal{L}(\Sigma) \times \mathcal{L}(\Sigma)$. The constraint that the lists have length ω is separate.

In order to ease the mechanisation of the reduction, the variant 3-**PR** is defined explicitly as a new problem using a separate inductive predicate for validity (cf. Section 5.1.3) and an inductive datatype for storing the three symbols of the premise or conclusion of a window instead of lists. Then, we do another reduction from 3-**PR** to **PR**, which is just a trivial embedding.

Our definition of validity for 3-**PR** is parameterised over an abstract rewritesHead : $\mathcal{L}(\Sigma) \to \mathcal{L}(\Sigma) \to \mathbb{P}$ predicate which determines whether a prefix of the first list rewrites to a prefix of the second list. The rewHead predicate of Definition 5.3 is

[3]This is mainly because Coq's built-in vectors are unpleasant to use.

an instance of that. However, for the proof of the reduction, we use another instantiation: sets of rewrite windows can be alternatively formalised as inductive predicates of type

$$\Sigma \to \Sigma \to \Sigma \to \Sigma \to \Sigma \to \Sigma \to \mathbb{P},$$

with arguments corresponding to the six elements determining a rewrite window. Using inductive predicates instead of lists has the huge advantage of easier automation in Coq: inversions are straightforward and proofs that a rewrite is possible can be done by `eauto` with suitable hints.

As the finite types for the alphabet of **PR** are not directly extractable to L, we define flat versions **FlatPR** and **Flat-3-PR** similar to the definition of **FlatClique** in Chapter 3. The agreement with the non-flat variants is straightforward to derive.

FlatPR

Flat-3-PR

5.8.2 Organisation of Rewrite Rules

A debatable decision is the organisation of rewrite rules. Regarding R_{tape}, we have opted for two inductive predicates shiftRightRules and identityRules, which are then combined into tapeRules. The definition of identityRules is abbreviated as presented in Section 5.3 and thus includes spurious windows, while we explicitly make the distinction between $\sigma : \Sigma$ and blanks for shiftRightRules: the spurious windows for shifting make inversions much harder. The rules for shifting the tape to the left are **defined** to be the polarity reversion of the right-shifting rules, which makes Lemma 5.19 almost trivial.

For the transitions, the rules are much more complicated. We have a hierarchy of inductive predicates that organise them (Figure 5.2). First, we differentiate according to the syntactic form of the read and written symbols. For instance, the inductive predicate for (Some, None) corresponds to transitions of the form $\delta(q, {}^\circ\sigma) = (q', \emptyset, a)$. Next is a case analysis on the location of the state symbol, and finally, we distinguish between the three types of movement. It might seem peculiar that we analyse the state symbol location before the movement as the transition function determines the movement but not the location. This makes the inversions in the proofs of the Lemmas 5.32 and 5.33 more efficient. Based on the location of the state symbol, one arrives at a contradiction earlier. Moving to this organisation provided a significant reduction in RAM usage when running the proof.

The bottom level of the hierarchy can be shared among all four cases except for (None, None), where we need the special handling for the edge of the used tape region. Moreover, for all cases but the (None, None) case, we implement the simplified rules. The spurious windows do not result in any complication in the proofs.

As noted in the proof outline of Lemma 5.32, the number of cases for this lemma is huge: in the end, there are 100 non-contradictory cases left after analysing the possible transitions the Turing machine takes and accounting for the possible symbols

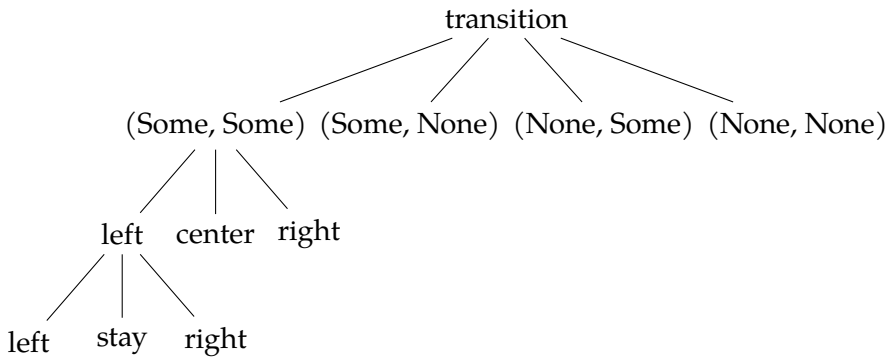


Figure 5.2: The hierarchy of inductive predicates for transitions.

at the heads of the two tape halves. The proof thus heavily relies on automation. There is `LTac` code that analyses the transition the Turing machine can take, destructs the two tape halves far enough, and then puts together the needed lemmas. Custom tactics to invert the tape representation relation $\sim_t$ are quite important. Running the proof takes about 35 minutes[4] and 4 gigabytes of RAM, despite handwritten inversion lemmas for the second and third level of Figure 5.2.

5.8.3 List-based Rules

Of course, one still needs to derive a list-based version of the rewrite rules (i.e. a variant from which we can directly compute the list of rewrite windows) in order to do the extraction to L and the accompanying running time analysis. Moreover, we use the representation of finite types using bounded subsets of $\mathbb{N}$ as in Section 3.2.1.

The proof of agreement of the inductive rules with the flat list-based rules proceeds in two steps: first, we define list-based rules still using finite types and show their agreement with the inductive rules. As the hierarchy of Figure 5.2 does not admit an easy computation of the windows (it is not at all natural to do a case analysis on the position of the state symbol when *computing* the windows), we additionally define the inductive predicates using an alternative hierarchy that closely matches the way the windows can be computed (but is slower to invert). The advantage of this approach is that the agreement between the two hierarchies can be proved fully automatically at the level of inductive predicates. In the second step, the flat list-based rules are defined and it is shown that they agree with the finite-type list-based rules, modulo the representation defined by $\approx$.

We desire that the finite-type and flat rules are defined in one go, with a minimal effort required to show their agreement. Moreover, in the end, we also need to instantiate the rules with all concrete elements of Σ and with the possible polarities

[4]Measured on an Intel Ivy Bridge Core i7-3740QM @ 3.5GHz with 24GB RAM.

to obtain the list of rewrite windows. In order to fulfill these two requirements, we define an abstract version fAlphabet of the alphabet $\mathcal{A}$ that contains "holes" where an element of Σ or Σ_{state}, a state of Q, or a polarity has to be placed. More specifically, these holes are formalised as variables which can be of type Σ, Σ_{state}, Q, or polarity. Given an environment which provides values for the variables, we have two reification procedures reifyFin and reifyFlat taking an element of fAlphabet either to a representation using the finite type $\mathcal{A}$ or a flat representation using $\mathbb{N}$. It is shown that, for environments related by $\approx$, the corresponding outputs of reifyFin and reifyFlat are again related by $\approx$.

fAlphabet

reifyFin

reifyFlat

We then lift the reification procedures to windows and lists of windows. For the instantiation of the rules, we generate all possible environments for the given number of variables a rule uses and then instantiate the rule with each of those environments to obtain a rewrite window. In order to generate the windows for the transitions, this needs to be combined with an iteration over all states and elements of Σ_{state}, doing a case analysis on the transition function for every valuation.

The proof of agreement with the inductive rules can mostly be automated and only requires the manual choice of an assignment to the variables of a rule.

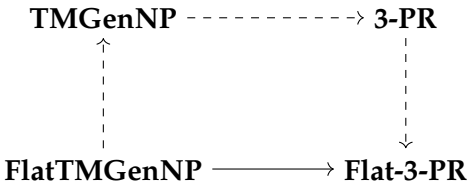
The full reduction produces a **Flat-3-PR** instance, where the flat reification of the rewrite windows is used. As the Turing machines use finite types and vectors, which are both not directly extractable, we also have a flat encoding of these together with a flat variant **FlatTMGenNP** of **TMGenNP**. The definition of flat Turing machines TM also represents the finite type for the tape alphabet by natural numbers and uses a list-based representation of the transition function; the formalisation was already available in the Coq library of undecidable problems [12].

FlatTMGenNP

TM

Theorem 5.53 ***FlatTMGenNP*** $\preceq_{\text{p}}$ ***Flat-3-PR***

Logically, the reduction for the flat problems takes the following shape:



Computationally, we do a direct reduction from **FlatTMGenNP** to **Flat-3-PR**, but for proving the equivalence, we take the route via the non-flat variants of the problems.

Together with a trivial proof of **Flat-3-PR** $\preceq_{\text{p}}$ **FlatPR**, we obtain a reduction to **FlatPR**.

Chapter 6

Reducing PR to Binary PR

In this chapter, we reduce **PR** instances over an arbitrary finite alphabet $\Sigma : \mathsf{finType}$ to instances over a binary alphabet represented by the type $\mathbb{B}$. The idea of this reduction is to replace every symbol $\sigma : \Sigma$ with a unique bitstring of length $|\Sigma|$. More specifically, the i-th element σ_i (counting from zero) of the finite type Σ is replaced by the string $0^i 1 0^{|\Sigma|-i-1}$, where we write 0 for F and 1 for T for convenience. Thus, initial strings, rewrite windows, and final substrings need to be adapted. This operation is incorporated using special string homomorphisms which we call uniform.

For the reduction, we modify the offset o of the **PR** instance by multiplying it with $|\Sigma|$[1]. This is motivated by the fact that, semantically, a bitstring of length $|\Sigma|$ represents a single symbol of the original alphabet.

The variant of **PR** in which the alphabet is fixed to $\mathbb{B}$ is called **BinaryPR**. All definitions of Section 5.1 carry over.

6.1 Uniform Homomorphisms

We introduce string homomorphisms and a special subclass of them, uniform homomorphisms. While one usually uses string homomorphisms over a finite alphabet, we do not require finiteness for most of the definitions and results in this section.

homomorphism

Definition 6.1 (String Homomorphisms) *Given types Σ_1, Σ_2, $h : \Sigma_1^* \to \Sigma_2^*$ is a homomorphism, written* homomorphism h, *if $h(a_1 +\!\!+ a_2) = h\ a_1 +\!\!+ h\ a_2$ for all $a_1, a_2 : \Sigma_1^*$.*

Fact 6.2 *Fix a homomorphism $h : \Sigma_1^* \to \Sigma_2^*$.*

1. $h\ [\,] = [\,]$
2. $h(a :: b) = h[a] +\!\!+ h\ b$
3. $h(\mathit{concat}\ l) = \mathit{concat}\ [\ h\ x \mid x \in l\ ]$

[1] In fact, this reduction is the motivation for having the offset in the first place.

Given a function $f : \Sigma_1 \to \Sigma_2^*$, we can define a canonical homomorphism

$$\mathsf{canonicalHom}\ f \coloneqq \lambda l.\mathsf{concat}\ [\ f\ x \mid x \in l\ ] \qquad \boxed{\mathsf{canonicalHom}}$$

lifting f to lists.

Proposition 6.3 *Let* $f : \Sigma_1 \to \Sigma_2^*$. *Then* (1) *canonicalHom* f *is a homomorphism and* (2) *it is the unique homomorphism satisfying the property* $\forall x, h[x] = f\ x$.

A uniform homomorphism is a special kind of homomorphism. First of all, it is ε**-free**, meaning that it only maps $[\,]$ to $[\,]$. Secondly, it maps all pairs of strings a, b with $|a| = |b|$ to strings of the same length. Put differently, the length of every string is multiplied by a constant k when passing through the homomorphism.

Definition 6.4 (Uniform Homomorphisms) $\boxed{\mathit{uniformHom}}$

$$\mathit{uniformHom}\ h \coloneqq \mathit{homomorphism}\ h \wedge (\forall x, |h\ [x]| \geqslant 1) \wedge (\forall x_1\ x_2, |h\ [x_1]| = |h\ [x_2]|)$$

Note that the definition is in terms of singleton lists. The more general property follows directly by using the properties of a homomorphism:

Proposition 6.5 *Let* $h : \Sigma_1^* \to \Sigma_2^*$ *with uniformHom* h.

1. $|l_1| = |l_2| \to |h\ l_1| = |h\ l_2|$
2. $h\ l = [\,] \to l = [\,]$

Again, we can define a canonical uniform homomorphism starting from a function $f : \Sigma_1 \to \Sigma_2^*$ satisfying $|f\ x| = k$ for all $x : \Sigma_1$ and some $k > 0$.

Proposition 6.6 (Canonical Uniform Homomorphisms)
Let $f : \Sigma_1 \to \Sigma_2^*$ *with* $\forall x, |f\ x| = k$ *for some* $k > 0$. *Then uniformHom* (*canonicalHom* f).

Conversely, we can formalise the above intuition that the length of a string is multiplied by a constant k when a uniform homomorphism is applied to it.

Lemma 6.7 *If* Σ : *finType and uniformHom* h, *then* $\mathbf{\Sigma}_k.\forall x, |h\ x| = k \cdot |x|$.

Proof As Σ is a finite type, inhabitation is decidable. If $|\Sigma| = 0$, then we pick 42. Otherwise, let $\sigma : \Sigma$ and pick $|h\ [\sigma]|$. For arbitrary $x : \Sigma^*$, the statement follows by induction on x and using the defining properties of a uniform homomorphism. □

6.2 Applying Uniform Homomorphisms to PR

We can show that **PR** is invariant under injective uniform homomorphisms. That is, given an injective uniform homomorphism h and a **PR** instance S, we can define another **PR** instance S' where we have applied h to all strings of S. S' is a yes-instance if, and only if, S is a yes-instance.

Fix a **PR** instance $S = (\Sigma_1, o, \omega, x_0, R, R_{\text{final}}, t)$. Assume that $|\Sigma_1| > 0$. Instances of **PR** not satisfying this property are trivial no-instances.

Without loss of generality[2], we assume that the homomorphism is given as $h' : \Sigma_1 \to \Sigma_2^*$ with

$$(A_1) \quad |h'\, x| = k \text{ for a fixed } k > 0$$
$$(A_2) \quad \text{injective } h'$$

Then, we define the full homomorphism as $h \coloneqq \mathsf{canonicalHom}\ h'$. Properties (A_1) and (A_2) carry over to h:

Proposition 6.8

1. *uniformHom* h
2. $|h\, x| = k \cdot |x|$
3. *injective* h

The following lemma allows us to split the image of h along multiples of k and follows from the properties of uniform homomorphisms in a straightforward way.

Lemma 6.9 (Inversion of h)

1. $h\, a = u +\!\!+ v \to |u| = k \to \exists a_1\ a_2, a = a_1 :: a_2 \wedge h\, [a_1] = u \wedge h\, a_2 = v$
2. $h\, a = u +\!\!+ v \to |u| = c \cdot k \to \exists a_1\ a_2, a = a_1 +\!\!+ a_2 \wedge h\, a_1 = u \wedge h\, a_2 = v$

Homomorphisms can, of course, be lifted to rewrite windows.

h_{window}

Definition 6.10 $h_{window}\ w \coloneqq (h\ (\mathit{prem}\ w), h\ (\mathit{conc}\ w))$

The transformed **PR** instance is defined as

R_h
S'

$$R_h \coloneqq [\ h_{\text{window}}\ w \mid w \in R\ ]$$
$$S' \coloneqq (\Sigma_2, k \cdot o, k \cdot \omega, h\, x_0, R_h, [\ h\, l \mid l \in R_{\text{final}}\ ], t).$$

[2]This follows from the results of the previous section.

Note that the offset and the width both need to be multiplied by the multiplicative factor k of h. It is easy to see that this definition is well-formed, that is, S' does again satisfy the syntactic constraints of **PR** (Definition 5.2). The goal for the rest of this section is to show that **PR** $S \leftrightarrow$ **PR** S'.

We start with the agreement of rewHead. The first statement of the following lemma is the equivalence one would expect. However, it is rather weak, as the backwards direction requires that we know the premise and conclusion to be in the image of h. For the proof of equivalence of S and S', we need a stronger statement: if we have a string $h\ a$ that rewrites to a string s, we need to be able to derive that s is in the image of h.

Lemma 6.11 (Agreement for rewHead**)**

1. *rewHead* $w\ a\ b \leftrightarrow$ *rewHead* $(h_{window}\ w)\ (h\ a)\ (h\ b)$
2. *If* $w \in R_h$, $|a_1| = o$, $|u| = k \cdot o$, *and* rewHead $w\ (h\ a_1 +\!\!+ h\ a_2)\ (u +\!\!+ v)$, *then there exists* b_1 *with* $u = h\ b_1$ *and* $|b_1| = o$.

Proof Using Lemma 6.9 and the results of Proposition 6.8. □

This can be lifted to $\rightsquigarrow$. For the proof, yet another characterisation of validity will be quite convenient. The base case allows one to prove a rewrite using a single rewrite window[3]. In the successor case, we can add o symbols, where o is the offset, by providing a rewrite window for the head.

$$\frac{w \in R}{\mathsf{prem}\ w \rightsquigarrow^D \mathsf{conc}\ w}$$

$$\frac{a \rightsquigarrow^D b \quad |u| = o \quad |v| = o \quad w \in R \quad \mathsf{rewHead}\ w\ (u +\!\!+ a)\ (v +\!\!+ b)}{(u +\!\!+ a) \rightsquigarrow^D (v +\!\!+ b)}$$

$\boxed{\rightsquigarrow^D}$

To some extent, this is the most intuitive characterisation of validity: it does not allow vacuous rewrites (i.e. allows only rewrites of length $\geqslant \omega$). Accordingly, it is not fully equivalent to $\rightsquigarrow$:

Lemma 6.12 (Agreement of $\rightsquigarrow$ and $\rightsquigarrow^D$) $a \rightsquigarrow_R b \wedge |a| \geqslant \omega \leftrightarrow a \rightsquigarrow^D b$

The restriction to strings of length at least ω does not pose a problem as Definition 5.2 imposes that as a syntactic constraint. Using the new characterisation, we can now prove the equivalence for $\rightsquigarrow$.

Lemma 6.13 (Agreement for $\rightsquigarrow$) *Let* a *with* $|a| \geqslant \omega$ *be given.*

[3] In the mechanisation, this first rule is formulated differently so that the proof of equivalence works without assuming the rewrite windows to have width ω, see the comments at the end of Chapter 5.

1. $(h\ a) \rightsquigarrow_{R_h} b' \to \exists b, b' = h\ b \wedge a \rightsquigarrow_R b$.
2. $a \rightsquigarrow_R b \leftrightarrow (h\ a) \rightsquigarrow_{R_h} (h\ b)$

Proof 1. We switch to $\rightsquigarrow^D$ and do an induction on $\rightsquigarrow^D$. In the base case, we show that the whole string is covered by the rewrite window and use the definition of R_h and Lemma 6.11.1, as well as the facts provided by Proposition 6.8 and Lemma 6.9. In the successor case, we use Lemma 6.11.2.

2. Direction $\to$ follows by an easy induction. The other direction follows from 1. □

The last lemma can trivially be transferred to $\rightsquigarrow^n$. After having show that the final constraints agree in the expected way, we can obtain the main agreement result.

Theorem 6.14 **PR** $S \leftrightarrow$ **PR** S'

6.3 Reduction to BinaryPR

The previous section's results can be applied to an injective uniform homomorphism into $\mathbb{B}$ to directly obtain the reduction of **PR** to **BinaryPR**. We use a homomorphism which just computes a unary encoding of the alphabet's elements over a binary alphabet. The i-th element σ_i (counting from zero) of the finite type Σ is mapped to the string $F^i T F^{|\Sigma|-i-1}$.

Formally, let us fix a **PR**-instance $S = (\Sigma, o, \omega, x_0, R, R_{\text{final}}, t)$. In order to apply the results of the previous section, we rely on $|\Sigma| > 0$. We handle the case $|\Sigma| = 0$ separately in the end.

The following function defines the homomorphism for single elements.

Definition 6.15

$h_{\mathbb{N}}$ $h_\mathbb{N}\ sig\ n \coloneqq \textit{if}\ n \overset{?}{<} sig\ \textit{then}\ F^n + [T] + F^{sig-n-1}\ \textit{else}\ F^{sig}$

h_Σ $h_\Sigma\ \sigma \coloneqq h_\mathbb{N}\ (|\Sigma|)\ (\textit{index}\ \sigma)$

Here, $\overset{?}{<}$ denotes a Boolean decider for the relation $<$ and index *gives the position of an element of the finite type.*

The corresponding homomorphism can be obtained by using canonicalHom. However, the technique of Section 6.2 just asks us to provide a function operating on single symbols.

Properties (A_1) and (A_2) are straightforward to verify.

Fact 6.16 (Uniformity)

1. $|h_{\mathbb{N}}\ sig\ n| = sig$
2. $|h_{\Sigma}\ \sigma| = |\Sigma|$

Fact 6.17 (Injectivity)

1. $h_{\mathbb{N}}\ sig\ m = F^n + [T] + F^{sig-n-1} \to m = n$
2. *injective* h_{Σ}

The reduction now directly follows by Theorem 6.14. In the special case of $|\Sigma| = 0$, it maps to a trivial no-instance of **BinaryPR**.

Lemma 6.18 (PR reduces to BinaryPR) **PR** $S \leftrightarrow$ **BinaryPR** S', *where* S' *is defined as in Section 6.2.*

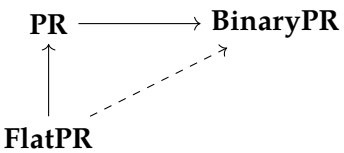
6.4 Mechanisation

We briefly comment on the differences of the Coq mechanisation and the proof on paper as well as on the proof of computability.

First of all, **BinaryPR** is defined to be a syntactically different problem from **PR** in the Coq mechanisation. The only change is that the alphabet is fixed to $\mathbb{B}$. Nearly all definitions (for instance the definition of $\rightsquigarrow$) can be re-used and do not need to be stated explicitly again.

Remember that we enforce the syntactic constraints on valid **PR** instances (see Definition 5.2) externally. Therefore, we explicitly need to assume wellformedness of the **PR** instance for the results of Section 6.2. As wellformedness can be decided, the reduction maps non-wellformed instances to a trivial no-instance.

BinaryPR has the nice property of being directly extractable as we do not work over a general finite type anymore. Thus, it is not required to define a flat version of the problem again. However, as **PR** still needs a separate flat version, our reduction structure looks as follows:



The main verification is done for the reduction of **PR** to **BinaryPR**, as described in this chapter. As this reduction cannot be extracted, we define a separate reduction of **FlatPR** to **BinaryPR**. This reduction basically uses the $h_{\mathbb{N}}$ function of Definition 6.15 to transform the finite subset of $\mathbb{N}$ to $\mathbb{B}$. For the verification of this

reduction, we show that for instances of **FlatPR** and **PR** that are equal up to representation of finite types $\approx$, the output of the respective reductions to **BinaryPR** is the same up to reordering of elements in lists. Logically, the reduction thus consists of two parts: a reduction from **FlatPR** to **PR**, converting the finite subset $\{0, \ldots, n-1\}$ of $\mathbb{N}$ into the finite type F_n, and the reduction from **PR** to **BinaryPR**. Computationally, however, this is shortcut to a single direct reduction which runs in polynomial time.

Theorem 6.19 ***FlatPR*** $\preceq_p$ **BinaryPR**

Chapter 7

Reducing Binary PR to Formula Satisfiability

In this chapter, we finally encode **PR** over a binary alphabet as a Boolean formula. This formula will not be in CNF, yet. The main idea of the encoding is to explicitly unfold the tableau induced by the **BinaryPR** instance. As we are working over a binary alphabet, each character of the involved strings can be accomodated by one Boolean variable. A satisfying assignment to the formula will therefore resemble a sequence of valid rewrites, starting with the initial string and ending up with a string satisfying the final substring constraint. The formula ϕ we generate is a conjunction of three gadgets: a formula ϕ_{init} enforcing that the first line of the tableau matches the initial string, a formula ϕ_{trans} encoding that the individual lines follow from each other according to the rewrite windows, and a formula ϕ_{final} which makes the final substring constraint hold.

7.1 Formula Satisfiability (FSAT)

We start by introducing a generalised variant of **SAT** that does not require the formula to be in CNF.

Formulas are defined inductively: $\boxed{\mathcal{F}}$

$$f : \mathcal{F} \coloneqq \mathsf{T} \mid v \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid \neg f \qquad (v : \mathsf{var})$$

One can directly derive the operators $\wedge$ and $\vee$ for more than two operands, which we will denote by $\bigwedge_{f \in l} f$ and $\bigvee_{f \in l} f$ for $l : \mathcal{L}(\mathcal{F})$.

Assignments $a : \mathsf{assgn}$ are defined in the same way as for CNFs, see Section 3.1. An evaluation function $\mathcal{E} : \mathsf{assgn} \to \mathcal{F} \to \mathbb{B}$ can be derived in the canonical way. We say that an assignment a satisfies f, written $a \models f$, if $\mathcal{E}\ a\ f = \mathsf{T}$.

Definition 7.1 (Formula Satisfiability) **FSAT** $f \coloneqq \exists a, a \models f$ **FSAT**

We could prove that **FSAT** is in NP using the same notion of small assignments as for CNF satisfiability. This will, however, already follow from the reduction of **FSAT** to **SAT** in Chapter 8.

Explicit Assignments The assignments we use, where we just store a list of variables to which the value T is assigned and to all other variables the value F is assigned implicitly, are quite indirect and unstructured. For the rest of this chapter a more explicit characterisation bearing resemblance to the Boolean strings used by **BinaryPR** will be quite convenient. One possible characterisation is a list of Booleans $e : \mathcal{L}(\mathbb{B})$ together with a start index s, where the value $e[i]$ at position i of this list is the value assigned to variable $s + i$. To variables outside the range $[s, s + |e|)$, the value F is assigned.

$[s.. + l)$

As we will be working with ranges of variables $[\mathsf{start}, \mathsf{start} + \mathsf{len})$ a lot throughout this chapter, we introduce the notation $[s.. + l)$ instead of $[s, s + l)$, which makes the notation less redundant. In the following, we use the letter I to denote variable ranges of this form.

We can naturally convert back and forth between such explicit assignments and assignments $a : \mathsf{assgn}$. First, we present a function that generates an explicit assignments for the variables in range $[\mathsf{start}, \mathsf{start} + \mathsf{len})$.

explicitA

$$\mathsf{explicitAssgn} : \mathsf{assgn} \to \mathbb{N} \to \mathbb{N} \to \mathcal{L}(\mathbb{N})$$
$$\mathsf{explicitAssgn}\ a\ \mathsf{start}\ 0 \coloneqq [\,]$$
$$\mathsf{explicitAssgn}\ a\ \mathsf{start}\ (1 + \mathsf{len}) \coloneqq \mathsf{explicitAssgn}\ a\ \mathsf{start}\ \mathsf{len} +\!\!+ [(\mathsf{start} + \mathsf{len}) \mathbin{\hat{\in}} a]$$

$a[I]$

We use the notation $a[s.. + l) \coloneqq \mathsf{explicitAssgn}\ a\ s\ l$ and may also write $a[I]$ for a variable range I.

Lemma 7.2 (Properties of explicitAssgn**)**

1. $|\mathit{explicitAssgn}\ a\ \mathit{start}\ \mathit{len}| = \mathit{len}$
2. $k < \mathit{len} \to (\mathit{explicitAssgn}\ a\ \mathit{start}\ \mathit{len})[k] = {}^{\circ}(\mathcal{E}\ a\ (\mathit{start} + k))$
3. $\mathit{explicitAssgn}\ a\ s\ (l_1 + l_2) = \mathit{explicitAssgn}\ a\ s\ l_1 +\!\!+ \mathit{explicitAssgn}\ a\ (s + l_1)\ l_2$

$e[I]$

Given an explicit assignment e, we can project out the assignment to a particular range of variables $I = [\mathsf{start}.. + \mathsf{len})$:

$$e[I] = e[\mathsf{start}.. + \mathsf{len}) \coloneqq (e[\mathsf{len}..])[..\mathsf{start}).$$

The subscripting notation thus is overloaded for full assignments and for explicit assignments, but this will not pose a problem as it will always be clear from the context what we mean.

7.2 Encoding Properties using Boolean Formulas

Based on explicit assignments, we now present ways to encode predicates using formulas $f : \mathcal{F}$ and show that this representation is closed under conjunction and disjunction, the main composition operations we need for the reduction.

Definition 7.3 (Encoding of Predicates) *A formula* f *encodes a predicate* $\mathrm{p} : \mathcal{L}(\mathbb{B}) \to \mathbb{P}$ *on a range of variables* $\mathrm{I} = [\mathit{start}.. + \mathit{len})$ *if, for every assignment* $\mathfrak{a}$, $\mathfrak{a} \models \mathrm{f}$ *is equivalent to satisfaction of* p *under the partial assignment given by* I*:* encodesPredAt

$$\mathit{encodesPredAt}\ \mathit{start}\ \mathrm{l}\ \mathrm{f}\ \mathrm{p} \coloneqq \forall \mathfrak{a}, \mathfrak{a} \models \mathrm{f} \leftrightarrow \mathrm{p}\ (\mathfrak{a}[\mathit{start}.. + \mathrm{l}])$$

This encoding scheme is rather simple: we can only prove statements about contiguous ranges of variables. An elaborate definition allowing a more fine-grained control would be possible, but over-complicated for our simple use case.

To stay in line with the variable range notation, we also write encodesPredAt $\mathrm{I}\ \mathrm{f}\ \mathrm{p}$ for a range of variables I.

A rather trivial but important fact is that encodesPredAt is extensional with respect to the encoded predicate. This will be quite convenient for proving correctness of encodings by switching to an equivalent characterisation of the predicate we want to encode.

Fact 7.4 (Extensionality) *Let* $\forall e, |e| = \mathrm{l} \to \mathrm{p}_1\ e \leftrightarrow \mathrm{p}_2\ e$*. Then*

$$\mathit{encodesPredAt}\ \mathrm{s}\ \mathrm{l}\ \mathrm{f}\ \mathrm{p}_1 \leftrightarrow \mathit{encodesPredAt}\ \mathrm{s}\ \mathrm{l}\ \mathrm{f}\ \mathrm{p}_2.$$

As a first example, we define the encoding of a literal:

$$\mathsf{encodeLiteral}\ \nu\ \mathsf{sign} : \mathcal{F} \coloneqq \textbf{if}\ \mathsf{sign}\ \textbf{then}\ \nu\ \textbf{else}\ \neg\nu$$ encodeLiteral

As one would expect, the formula encodeLiteral ν s forces variable ν to take the value s:

Proposition 7.5 (Correctness of encodeLiteral**)**

$$\mathit{encodesPredAt}\ \nu\ 1\ (\mathit{encodeLiteral}\ \nu\ \mathrm{s})(\lambda \mathrm{l}.\mathrm{l} = [\mathrm{s}])$$

Next, we show that conjunction and disjunction of formulas correspond to conjunction and disjunction of represented predicates. One problem with this is that the two predicates may talk about different variable ranges I_1 and I_2. Our simple representation model does not allow to accurately depict this, hence the new predicate will talk about the smallest range of variables I that contains I_1 and I_2. We write this as $\mathrm{I} = \mathrm{I}_1 \sqcup \mathrm{I}_2$.

$$\mathsf{combStart}\ \mathrm{s}_1\ \mathrm{s}_2 \coloneqq \min\ \mathrm{s}_1\ \mathrm{s}_2$$ combStart

$$\mathsf{combLength}\ \mathrm{s}_1\ \mathrm{s}_2\ \mathrm{l}_1\ \mathrm{l}_2 \coloneqq \max\ (\mathrm{s}_1 + \mathrm{l}_1)(\mathrm{s}_2 + \mathrm{l}_2) - \min\ \mathrm{s}_1\ \mathrm{s}_2$$ combLength

$$[\mathrm{s}_1.. + \mathrm{l}_1) \sqcup [\mathrm{s}_2.. + \mathrm{l}_2) \coloneqq [\mathsf{combStart}\ \mathrm{s}_1\ \mathrm{s}_2.. + \mathsf{combLength}\ \mathrm{s}_1\ \mathrm{s}_2\ \mathrm{l}_1\ \mathrm{l}_2)$$ $\mathrm{I}_1 \sqcup \mathrm{I}_2$

Using $e[\mathrm{s}.. + \mathrm{l}]$, we can restore the assignments to I_1 and I_2 given an assignment to $\mathrm{I}_1 \sqcup \mathrm{I}_2$:

Proposition 7.6 *Let* $I_1 = [s_1.. + l_1)$ *and* $I_2 = [s_2.. + l_2)$.

1. $a[I_1] = (a[I_1 \sqcup I_2])[(s_1 - \mathit{combStart}\ s_1\ s_2).. + l_1]$
2. $a[I_2] = (a[I_1 \sqcup I_2])[(s_2 - \mathit{combStart}\ s_1\ s_2).. + l_2]$

Now, we have all the tools necessary to combine the encoding of two predicates:

Lemma 7.7 (Encoding of $\wedge$ and $\vee$) *Let* $I_1 = [s_1.. + l_2)$ *and* $I_2 = [s_2.. + l_2)$. *Assume that* *encodesPredAt* $s_1\ l_1\ f_1\ p_1$ *and* *encodesPredAt* $s_2\ l_2\ f_2\ p_2$.

1. *encodesPredAt* $(I_1 \sqcup I_2)(f_1 \wedge f_2)$
   $(\lambda e.p_1(e[s_1 - \mathit{combStart}\ s_1\ s_2.. + l_1]) \wedge p_2(e[s_2 - \mathit{combStart}\ s_1\ s_2.. + l_2]))$
2. *encodesPredAt* $(I_1 \sqcup I_2)(f_1 \vee f_2)$
   $(\lambda e.p_1(e[s_1 - \mathit{combStart}\ s_1\ s_2.. + l_1]) \vee p_2(e[s_2 - \mathit{combStart}\ s_1\ s_2.. + l_2]))$

Proof Straightforward using Proposition 7.6. □

By conjoining formulas encoding single literals, a list $L : \mathcal{L}(\mathbb{B})$ can be encoded by forcing a range of variables to be equal to the list.

encodeList

$$\begin{aligned}\mathsf{encodeList}\ s\ [\,] &\coloneqq \mathsf{T}\\ \mathsf{encodeList}\ s\ (l :: L) &\coloneqq \mathsf{encodeLiteral}\ s\ l \wedge \mathsf{encodeList}\ (1 + s)\ L\end{aligned}$$

Proposition 7.8 (Correctness of encodeList**)**

$$\mathit{encodesPredAt}\ s\ (|L|)\ (\mathit{encodeList}\ s\ L)\ (\lambda e.L = e)$$

encodeList will be the basic ingredient of encoding **PR**.

7.3 Reduction to FSAT

In this section, we show how to encode the three subformulas $\phi_{\mathsf{init}}, \phi_{\mathsf{trans}}, \phi_{\mathsf{final}}$. As most of the encodings are straightforward to see from the definition of **PR**, we only give an informal description for some of them.

From now on, fix a **BinaryPR** instance $(o, \omega, x_0, R, R_{\mathsf{final}}, t)$ that satisfies the syntactic constraints of Definition 5.2. For our convenience, we define $m \coloneqq |x_0|$ to be the length of each of the tableau's lines.

m

The variable layout of the formula we produce is demonstrated by Figure 7.1

We start with the encoding of ϕ_{trans}. This is most intuitive when looking at the explicit characterisation of validity (see Definition 5.4): we have a big conjunction over all pairs of subsequent lines for which we have to enforce validity. Validity can be encoded as a conjunction over the offsets at which a window needs to hold, and

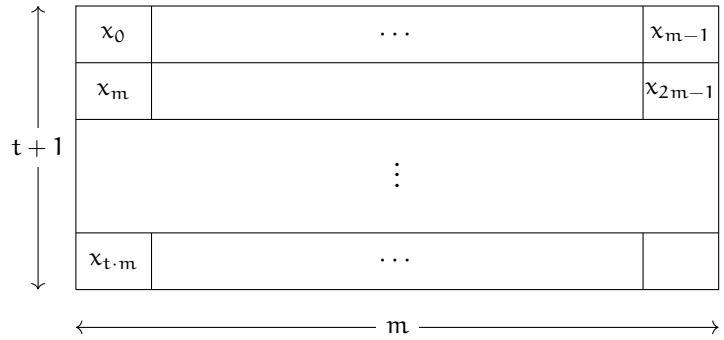


Figure 7.1: The variable layout of the constructed **FSAT** instance does directly mirror the layout of the **PR** strings in a tableau.

at each position we have a disjunction over all possible rewrite windows. First of all, we need to be able to encode a single rewrite window. As a rewrite windows just encodes two lists, this is straightforward to derive as a conjunction for premise and conclusion. Since the premise of the window needs to match in one line of the tableau while the conclusion needs to hold for the next line, the encoding is parameterised over two indices s_1 and s_2 giving the respective start of the encoding.

$$\mathsf{encodeWindow}\ s_1\ s_2\ w \coloneqq \mathsf{encodeList}\ s_1\ (\mathsf{prem}\ \ w) \wedge \mathsf{encodeList}\ s_2\ (\mathsf{conc}\ \ w) \qquad \boxed{\mathsf{encodeWindow}}$$

The correctness statement of course also needs to range over all the variables between the two starting indices.

Proposition 7.9 (Correctness of encodeWindow**)** *Let* $s, l : \mathbb{N}$ *and* $win : \mathit{window}(\mathbb{B})$.

$$\begin{aligned}&\mathit{encodesPredAt}\ s\ (l+\omega)\ (\mathit{encodeWindow}\ s\ (s+l)\ win)\\ &\qquad(\lambda e.e[0..+\omega) = \mathit{prem}\ \ win \wedge e[l..+\omega) = \mathit{conc}\ \ win)\end{aligned}$$

This can be lifted to express that one of the windows of R holds at a given position by computing a disjunction over the list of windows. We assume a function $\mathsf{encodeWindows} : \mathbb{N} \to \mathbb{N} \to \mathcal{F}$ taking the two start indices for that purpose, without going into more detail. $\boxed{\mathsf{encodeWindows}}$

Proposition 7.10 (Correctness of encodeWindows**)** *Let* $s, l : \mathbb{N}$.

$$\begin{aligned}&\mathit{encodesPredAt}\ s\ (l+\omega)\ (\mathit{encodeWindows}\ s\ (s+l))\\ &\qquad(\lambda e.\exists w, w \in R \wedge e[0..+\omega) = \mathit{prem}\ \ w \wedge e[l..+\omega) = \mathit{conc}\ \ w)\end{aligned}$$

A bit more interesting is the function needed to encode all windows at every offset of a string. Technically, this is a conjunction over all possible offsets. The fact

that the windows only need to hold at multiples of o is a minor inconvenience as this means that we cannot define the function computing the formula using direct structural recursion over the position at which we encode the windows next[1]. Instead, we employ a step index which is initialised to the length $m = |x_0|$ of the lines of the tableau to make the function structurally recursive.

encodeWindowsLine

$$\begin{aligned}&\mathsf{encodeWindowsLine}'\ \mathsf{step}\ l\ s_1\ s_2 \coloneqq \textbf{if}\ l \overset{?}{<} \omega\ \textbf{then}\ \mathsf{T}\\ &\textbf{else match}\ \mathsf{step}[\ 0 \Rightarrow \mathsf{T} \mid 1 + \mathsf{step} \Rightarrow \mathsf{encodeWindows}\ s_1\ s_2\\ &\qquad\qquad \wedge\ \mathsf{encodeWindowsLine}'\ \mathsf{step}\ (l - o)\ (s_1 + o)\ (s_2 + o)\ ]\end{aligned}$$

$$\mathsf{encodeWindowsLine}\ \mathsf{step}\ s \coloneqq \mathsf{encodeWindowsLine}'\ m\ m\ s\ (s + m)$$

Intuitively, the parameter s_1 of $\mathsf{encodeWindowsLine}'$ determines the first index starting at which the premises of the windows are placed, while s_2 is the starting index of the conclusions.

Proposition 7.11 (Correctness of $\mathsf{encodeWindowsLine}$**)** *Let* $l \leqslant m$.

$$\begin{aligned}&\mathit{encodesPredAt}\ s\ (l + m)\ (\mathit{encodeWindowsLine}'\ l\ l\ s\ (s + m))\\ &\qquad (\lambda e.(e[0.. + l)) \rightsquigarrow_R (e[m.. + l))\end{aligned}$$

Moreover, we have

$$\mathit{encodesPredAt}\ s\ (2 \cdot m)(\mathit{encodeWindowsLine}\ s)(\lambda e.(e[0.. + m)) \rightsquigarrow_R (e[m.. + m))$$

Proof The second statement is the first one instantiated with $l \coloneqq |x_0|$. For the first statement, we use strong induction on l. □

ϕ_{trans}

Now we can obtain the formula ϕ_{trans} by forming a conjunction over all lines of the tableau.

Lemma 7.12 (Correctness of ϕ_{trans}**)**

$$\begin{aligned}&\mathit{encodesPredAt}\ 0\ ((1 + t) \cdot m)\ \phi_{\mathit{trans}}\\ &\qquad (\lambda e.\forall 0 \leqslant i < t, ((e[i \cdot m.. + m)) \rightsquigarrow_R (e[(1 + i) \cdot m.. + m)))\end{aligned}$$

The final constraint can be encoded similarly: we have a nested disjunction over the strings in R_{final} and the possible positions at which they can match. For the latter, remember that the final substrings need to match at positions which are multiples of the offset o. Therefore, we employ a step-indexing technique similar to the one for $\mathsf{encodeWindowsLine}$.

[1] Coq requires functions to be structurally recursive.

Lemma 7.13 (Correctness of ϕ_{final}) $\boxed{\phi_{final}}$

$$\mathit{encodesPredAt}\ (t \cdot m)\ m\ \phi_{\mathit{final}}\ (\lambda e.e \models R_{\mathit{final}})$$

Finally, the initial constraint is simply an instance of encodeList: the first $|x_0|$ variables have to match x_0. Therefore, in the end the whole formula is

$$\phi \coloneqq \text{encodeList}\ 0\ x_0 \wedge \phi_{\text{trans}} \wedge \phi_{\text{final}}. \qquad \boxed{\phi}$$

Lemma 7.14 (Correctness of ϕ)

$$\begin{aligned}\mathit{encodesPredAt}\ 0\ ((1+t)\cdot m)\ \phi\\ (\lambda e.\ \ & e[0..+m) = x_0\\ & \wedge(\forall 0 \leqslant i < t, ((e[i\cdot m..+m)) \rightsquigarrow_R (e[(1+i)\cdot m..+m)))\\ & \wedge e[t\cdot m..+m) \models R_{\mathit{final}})\end{aligned}$$

Theorem 7.15 (BinaryPR reduces to FSAT)

$$\textbf{FSAT}\ \phi \leftrightarrow \exists x_t, x_0 \rightsquigarrow_R^t x_t \wedge x_t \models R_{\text{final}}$$

Proof

$\rightarrow$: As **FSAT** ϕ holds, we have a satisfying assignment a. By Lemma 7.14, the corresponding explicit assignment to the first $(1+t)\cdot|x_0|$ variables witnesses a valid rewrite sequence satisfying the final constraint. We prove that $e[0..m) \rightsquigarrow_R^t e[t\cdot m..+m)$ by switching to ${}^t\!\rightsquigarrow_R$ and doing an induction on t.

$\leftarrow$: From the proof that the initial string rewrites in t steps to a final string we have to generate a sequence of intermediate strings that together form a satisfying assignment. For this, we generalise to arbitrary strings x, y with $|x| = |x_0| = m$ and show that, if $x \rightsquigarrow_R^n y$, then there exists e with $|e| = (1+n)\cdot m$ and

$$\begin{aligned}& e[0..+m) = x \wedge e[n\cdot m..+m) = y\\ \wedge(\forall 0 \leqslant i < n, & (e[i\cdot m..+m)) \rightsquigarrow_R e[(1+i)\cdot m..+m))).\end{aligned}$$

This statement follows by an induction on $x \rightsquigarrow_R y$. □

7.4 Mechanisation

We again comment on some aspects of the mechanisation.

As in the previous chapter, we have to treat non-wellformed instances separately by first deciding wellformedness and mapping violating instances to a trivial no-instance.

A major inconvenience is that many of the representation proofs are somewhat repetitive and mainly rely on inductively forming conjunctions and disjunctions of formulas. Except for the two functions which need step-indexing, these can be taken care of by a few lemmas that allow to form the disjunction or conjunction of a list of formulas at a fixed position or by replicating a formula parameterised by the index of the first variable it uses at various positions for a given offset between the replications. The authors of [20] face a similar problem in encoding certain predicates using Diophantine equations. They come up with Coq tactics that can largely automate the process of finding an encoding and proving its correctness if the predicate has a shape that is already known, using the Coq unification mechanism. If the predicate is not in a known shape, extensionality can be used to bring it into an equivalent shape, similar to our Fact 7.4. The key idea of their approach is to use Sigma types that informatively bundle the encoding and its proof of correctness. Sadly, a similar trick can not be applied in our setting: Sigma types (or even their computational part) currently are not extractable to L. Thus, it seems unavoidable to explicitly define the formula encoding a particular predicate.

As both **BinaryPR** and **FSAT** are directly extractable, we need not define separate flat versions. Thus, we obtain the following polynomial-time reduction:

Theorem 7.16 **BinaryPR** $\preceq_{\mathrm{p}}$ **FSAT**

Chapter 8

Reducing Formula Satisfiability To SAT

In this chapter, we finally show how to convert an arbitrary logical formula $f : \mathcal{F}$ into CNF. If one does the conversion in a naive way, this incurs an exponential blowup. Consider, for instance, the formula $f_1 \vee f_2$ for two arbitrary formulas f_1, f_2. If we were to first recursively convert f_1, f_2 to CNFs N_1, N_2 and then apply distributivity to end up with a new CNF N equivalent to $N_1 \vee N_2$, the number of clauses in N were in $\mathcal{O}(|N_1| \cdot |N_2|)$.

Example 8.1 *Assume that* $N_1 = C_1 \wedge C_2$ *and* $N_2 = C_3 \wedge C_4$. *Then*

$$\begin{aligned} N \leftrightarrow & N_1 \vee N_2 \\ \leftrightarrow & (C_1 \vee (C_3 \wedge C_4)) \wedge (C_2 \vee (C_3 \wedge C_4)) \\ \leftrightarrow & (C_1 \vee C_3) \wedge (C_1 \vee C_4) \wedge (C_2 \vee C_3) \wedge (C_2 \vee C_4) \end{aligned}$$

Clearly, this does not work if we want to obtain a polynomial-time reduction. Instead, we use the Tseytin transformation [31]. The key insight behind the transformation is that the exponential blowup is caused by having to duplicate subformulas, as can be seen in Example 8.1. This can be prevented by introducing new variables which represent the subformulas, so that we may use the variable instead of duplicating the whole formula. Introducing new variables is fine since we only require the resulting CNF N to be *equisatisfiable* to the original formula f, i.e. **SAT** $N \leftrightarrow$ **FSAT** f, but not equivalent.

In the Tseytin transformation, each subformula f' of the formula f is represented by a variable v' together with a CNF N', such that **FSAT** $f' \leftrightarrow$ **SAT** $(v' \wedge N')$. Intuitively, the CNF N' forces the variable v' to be equivalent to f' for any assignment.

Example 8.2 *As an example, we consider the transformation for the disjunctive formula* $f = f_1 \vee f_2$. *First of all, we recursively apply the transformation to* f_1 *and* f_2 *in order to obtain variables* v_1 *and* v_2 *and CNFs* N_1 *and* N_2 *that represent the respective formulas. Of course, we have to take care that the new variables introduced for the transformations of* f_1

and f_2 *do not overlap. We now add a fresh variable* v *for the subformula* $f_1 \vee f_2$ *and build a CNF* N *such that* $v \wedge N$ *is equisatisfiable to* $f_1 \vee f_2$. N *should enforce that* $v \leftrightarrow (v_1 \vee v_2)$ *and that* v_1 *and* v_2 *represent* f_1 *and* f_2*, respectively. Therefore, set* $N = N_1 \wedge N_2 \wedge N'$*, where* N' *is the CNF*

$$N' \coloneqq (\neg v \vee v_1 \vee v_2) \wedge (\neg v_1 \vee v) \wedge (\neg v_2 \vee v),$$

which is equivalent to $v \leftrightarrow (v_1 \vee v_2)$.

8.1 Preliminaries: Composing Partial Assignments

As mentioned in the introduction, the Tseytin transformation adds new variables. In order to prove the transformation correct, we need some tools to be able to introduce fresh variables and compose assignments to different ranges of variables.

First of all, we define functions computing the maximum variable used in a CNF or formula (or zero if no variable is used). We refer to both of these functions by maxVar; it will be clear from the context whether we mean CNFs or formulas.

maxVar

Moreover, we need predicates that allow us to express that all variables of a formula or CNF are contained in a non-contiguous range of variables such as $[0, b_1) \cup [b_2, b_3]$. Variable ranges are formalised as predicates $r : \mathbb{N} \to \mathbb{P}$. For instance, the range $[0, b_1) \cup [b_2, b_3]$ is represented as $\lambda n.n < b_1 \vee (n \geqslant b_2 \wedge n \leqslant b_3)$. Nevertheless, we use the more intuitive mathematical notation on paper and write $v \in r$ for $r\ v$ and $r_1 \subseteq r_2$ for $\forall v, r_1\ v \to r_2\ v$.

$v \in r$

$r_1 \subseteq r_2$

$v \in_{\mathsf{cnf}} N$

We use the predicates $v \in_{\mathsf{cnf}} N$ and $v \in_{\mathcal{F}} f$ to denote that a variable is contained in a CNF or formula.

$v \in_{\mathcal{F}} f$

Definition 8.3 (Variable Containment) *Let a range* $r : \mathbb{N} \to \mathbb{P}$*, a formula* f*, and a CNF* N *be given.* f *only uses variables of* r*, written* $f \subseteq r$*, if* $\forall v \in_{\mathcal{F}} N, v \in r$.

$f \subseteq r$

N *only uses variables of* r*, written* $N \subseteq r$*, if* $\forall v \in_{cnf} N, v \in r$.

$N \subseteq r$

Of course, variable containment is monotonous.

Proposition 8.4

1. $r_1 \subseteq r_2 \to N \subseteq r_1 \to N \subseteq r_2$
2. $(N_1 +\!\!+ N_2) \subseteq r \leftrightarrow N_1 \subseteq r \wedge N_2 \subseteq r$

We now turn to the composition of assignments. Recall that assignments are represented by a list of variables to which T is assigned and to all other variables F

is assigned implicitly. In principle, this form of assignments is completely non-compositional as all assignments are total. However, we may still get a form of composition by overriding implicit F values. If we only compose assignments which explicitly assign values to disjoint variable ranges, we may still obtain the properties one would intuitively expect.

We define variable containment $a \subseteq r$ for assignments in the same way as for formulas and CNFs: $a \subseteq r$

$$a \subseteq r \coloneqq \forall v \in a, v \in r$$

Note that this definition only talks about variables to which T is assigned. We obtain the same properties as in Proposition 8.4.

Definition 8.5 (Composition and Restriction) *For two assignments a_1, a_2, the composition $a_1 \cup a_2$ is given as $a_1 \cup a_2 \coloneqq a_1 +\!\!+ a_2$. That is, we assign T to the variables to which either a_1 or a_2 assigns T, and assign F only to those values to which neither a_1 nor a_2 assigns T.* $a_1 \cup a_2$

The restriction of a to the range $[0, b)$ is defined as $a|_b \coloneqq [\ v \mid v \in a \wedge v \overset{?}{<} b\ ]$.[1] $a|_b$

Regarding composition, we can show that adding an assignment to variables which are disjoint from the variables used by a CNF or formula does not change the result of evaluation.

Proposition 8.6 *Let ranges $r, r' : \mathbb{N} \to \mathbb{P}$ and assignments a, a' be given with $a' \subseteq r'$ and $\forall n, \neg(n \in r \wedge n \in r')$, i.e. r and r' are disjoint. Then:*

1. $v \in r \to \mathcal{E}\ a\ v = \mathcal{E}\ (a \cup a')\ v$
2. $N \subseteq r \to (\mathcal{E}\ a\ N = \mathsf{T} \leftrightarrow \mathcal{E}\ (a \cup a')\ N = \mathsf{T})$
3. $f \subseteq r \to (\mathcal{E}\ a\ f = \mathsf{T} \leftrightarrow \mathcal{E}\ (a \cup a')\ f = \mathsf{T})$

Similarly, we can restrict an assignment. We only need this for formulas.

Proposition 8.7 *If $f \subseteq [0, n)$, then $a \models f \leftrightarrow (a|_n) \models f$.*

Remark 8.8 *During the development of the proof, we also considered various other forms of assignments which directly allow for partial assignments and thus admit a more intuitive form of composition (for instance the explicit assignments of Chapter 7). However, we need to directly reason about evaluation for much of the proof of correctness. Partial assignments require us to define the result of evaluation as an option value, which entails additional reasoning. We thus think that the added expressivity of partial assignments is not worth it.*

[1] We restrict this to ranges $[0, b)$ as we would have to reason about Boolean deciders for arbitrary ranges r otherwise.

8.2 Correctness of the Tseytin Transformation

In this section, we formally define the Tseytin transformation and verify its correctness. The main effort will be to justify the individual transformations for the operators $\wedge, \vee$ and $\neg$ due to the need to compose the new variables introduced by the subformulas. With this in mind, our first step is to eliminate $\vee$ in the formula so that we can omit the proof of correctness for this operator without loss of generality. Then, the main verification step is to define a relation between formulas and CNFs that is strong enough for an inductive proof to go through.

eliminateOr

The operator $\vee$ can directly be eliminated by applying De Morgan's law $\mathsf{f}_1 \vee \mathsf{f}_2 \leftrightarrow \neg(\neg \mathsf{f}_1 \wedge \neg \mathsf{f}_2)$. We assume a function $\mathsf{eliminateOr} : \mathcal{F} \to \mathcal{F}$ applying this law recursively. Moreover, we use an inductive predicate $\mathsf{orFree} : \mathcal{F} \to \mathbb{P}$ that holds exactly for all formulas not containing an application of $\vee$. We have the following properties:

orFree

Proposition 8.9 (Properties of eliminateOr**)**

1. *orFree* (*eliminateOr* f)
2. $\mathcal{E}\ \mathfrak{a}\ \mathsf{f} = \mathcal{E}\ \mathfrak{a}$ (*eliminateOr* f)

8.2.1 The Tseytin Transformation

Now, we define the transformation function. The main challenge is to keep track of the used variables, such that we can easily introduce fresh variables as needed. To that end, we define an auxiliary function $\mathsf{tseytin}' : \mathsf{var} \to \mathcal{F} \to \mathsf{var} \times \mathsf{cnf} \times \mathsf{var}$ (Figure 8.1). Its first argument is the next unused variable $\mathfrak{nf}$ (short for "next free"), followed by the formula f we would like to transform. It returns a triple $(\mathfrak{rv}, \mathsf{N}, \mathfrak{nf}')$, where $\mathfrak{rv}$ is the variable representing the formula, N is the accompanying CNF, and $\mathfrak{nf}'$ is the next free variable after transforming f. The function's different cases make use of primitive functions introducing the CNF constraints. They can be found in Figure 8.2. Note that we add a fresh variable for the variable base case, too, and force this variable to be equivalent to the original one. This is a design choice we make to simplify the invariant for the proof of correctness a bit. Alternatively, this case could read $\mathsf{tseytin}'\ \mathfrak{nf}\ \nu \coloneqq (\nu, [\,], \mathfrak{nf})$, returning just the original variable with an empty CNF. The transformation function tseytin (Figure 8.1) can be derived by first computing the maximum variable used by the formula and then using $\mathsf{tseytin}'$.

8.2.2 Proof of Correctness

We now show tseytin to be correct, in the sense that the resulting pair (ν, N) of representing variable ν and CNF N is equisatisfiable to the original formula. Formally, we define the following relation:

Definition 8.10 (Representation of Formulas by CNFs)

$\mathsf{f} \models\!\models (\nu, \mathsf{N})$

$$\mathsf{f} \models\!\models (\nu, \mathsf{N}) \coloneqq \mathbf{FSAT}\ \mathsf{f} \leftrightarrow \mathbf{SAT}\ ([(\mathsf{T}, \nu)] :: \mathsf{N})$$

$$
\begin{aligned}
&\mathsf{tseytin}' : \mathsf{var} \to \mathcal{F} \to \mathsf{var} \times \mathsf{cnf} \times \mathsf{var} && \boxed{\mathsf{tseytin}'}\\
&\mathsf{tseytin}'\ \mathfrak{nf}\ \mathsf{T} \coloneqq (\mathfrak{nf}, \mathsf{tseytinTrue}\ \mathfrak{nf}, 1 + \mathfrak{nf})\\
&\mathsf{tseytin}'\ \mathfrak{nf}\ \nu \coloneqq (\mathfrak{nf}, \mathsf{tseytinEquiv}\ \nu\ \mathfrak{nf}, 1 + \mathfrak{nf})\\
&\mathsf{tseytin}'\ \mathfrak{nf}\ (\mathsf{f}_1 \wedge \mathsf{f}_2) \coloneqq \mathbf{let}\\
&\qquad\qquad (r\nu_1, \mathsf{N}_1, \mathfrak{nf}_1) = \mathsf{tseytin}'\ \mathfrak{nf}\ \mathsf{f}_1\\
&\qquad\qquad (r\nu_2, \mathsf{N}_2, \mathfrak{nf}_2) = \mathsf{tseytin}'\ \mathfrak{nf}_1\ \mathsf{f}_2\\
&\qquad\qquad \mathbf{in}\ (\mathfrak{nf}_2, \mathsf{N}_1 +\!\!+ \mathsf{N}_2 +\!\!+ \mathsf{tseytinAnd}\ \mathfrak{nf}_2\ r\nu_1\ r\nu_2, 1 + \mathfrak{nf}_2)\\
&\mathsf{tseytin}'\ \mathfrak{nf}\ (\neg \mathsf{f}) \coloneqq \mathbf{let}\ (r\nu, \mathsf{N}, \mathfrak{nf}') = \mathsf{tseytin}'\ \mathfrak{nf}\ \mathsf{f}\\
&\qquad\qquad \mathbf{in}\ (\mathfrak{nf}', \mathsf{N} +\!\!+ \mathsf{tseytinNot}\ \mathfrak{nf}'\ r\nu, 1 + \mathfrak{nf}')\\
&\mathsf{tseytin}'\ \mathfrak{nf}\ (\mathsf{f}_1 \vee \mathsf{f}_2) \coloneqq \ldots
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{tseytin} : \mathcal{F} \to \mathsf{var} \times \mathsf{cnf} && \boxed{\mathsf{tseytin}}\\
&\mathsf{tseytin}\ \mathsf{f} \coloneqq \mathbf{let}\ (\nu, \mathsf{N}, \_) = \mathsf{tseytin}'\ (1 + \mathsf{maxVar}\ \mathsf{f})\ \mathbf{in}\ (\nu, \mathsf{N})
\end{aligned}
$$

Figure 8.1: The Tseytin transformation is defined via an auxiliary function keeping track of the next free variable which is initialised with the successor variable of the maximum variable used by the formula. The omitted case for $\vee$ is defined analogously to $\wedge$, replacing tseytinAnd with tseytinOr.

Function	CNF
$\mathsf{tseytinAnd}\ \nu\ \nu_1\ \nu_2$	$(\overline{\nu} \vee \nu_1) \wedge (\overline{\nu} \vee \nu_2) \wedge (\nu \vee \overline{\nu_1} \vee \overline{\nu_2})$
$\mathsf{tseytinOr}\ \nu\ \nu_1\ \nu_2$	$(\overline{\nu} \vee \nu_1 \vee \nu_2) \wedge (\nu \vee \overline{\nu_1}) \wedge (\nu \vee \overline{\nu_2})$
$\mathsf{tseytinNot}\ \nu\ \nu'$	$(\overline{\nu} \vee \overline{\nu'}) \wedge (\nu \vee \nu')$
$\mathsf{tseytinEquiv}\ \nu\ \nu'$	$(\nu \vee \overline{\nu'}) \wedge (\overline{\nu} \vee \nu')$
$\mathsf{tseytinTrue}\ \nu$	ν

Figure 8.2: The primitive CNFs for the Tseytin transformation. We use the notation $\overline{\nu}$ for $\neg\nu$.

The statement we would like to prove is that $\mathsf{tseytin}\ f = (v, N) \to f \models\!\!\models (v, N)$. As tseytin is defined in terms of $\mathsf{tseytin}'$, we first prove a more general statement for this function. The first direct generalisation one might come up with is

$$f \subseteq [0, b) \to nf \geqslant b \to \mathsf{tseytin}'\ nf\ f = (v, N, nf') \to f \models\!\!\models (v, N),$$

where we have generalised the initial value $1 + \mathsf{maxVar}\ f$ for nf. However, an inductive proof still does not go through: in the cases where an operator is used, we do not know enough to do any reasoning about the assignments as the relation does not hold any information on the structure of the CNFs and assignments. We therefore define a stronger relation parameterised over b, nf, and nf'. Intuitively, the variable range $[0, b)$ is used by the original formula and $[nf, nf')$ is the range of new variables introduced by the transformation.

Definition 8.11 (Strengthened Representation Relation)

$f \models\!\!\models^{nf,nf'}_{b} (v, N)$

$f \models\!\!\models^{nf,nf'}_{b} (v, N)$, *if the following conditions hold:*

1. $N \subseteq ([0, b) \cup [nf, nf'))$,
2. $v \in [nf, nf')$,
3. *for all* $a \subseteq [0, b)$, *there exists* $a' \subseteq [nf, nf')$ *such that* $(a \cup a') \models N$,
4. *and for all* a *with* $a \models N$, *the equivalence* $a \models v \leftrightarrow a \models f$ *holds.*

The first two conditions make the expected statements about the range of used variables. Note that the second condition would be too strong if we had not chosen to introduce a new variable in the variable case of the transformation.

The third condition states that for any assignment to the original variables $[0, b)$, we can extend this assignment to the new variables in range $[nf, nf')$ such that the CNF N is satisfied. Intuitively, the condition does hold since the CNF produced by the Tseytin transformation only contains constraints which relate the original variables to the new variables; therefore, for any assignment to the original variables, we should be able to find a consistent assignment for the new variables. Only if we force the representing variable to be T is N actually equisatisfiable to f.

Finally, the fourth condition says that an assignment satisfying N assigns the value T to the representing variable v if and only if it satisfies f.

Based on this relation, we can prove the following statement by induction on f:

$$f \subseteq [0, b) \to nf \geqslant b \to \mathsf{tseytin}'\ nf\ f = (v, N, nf') \to f \models\!\!\models^{nf,nf'}_{b} (v, N)$$

We factor out the inductive steps into compatibility lemmas for the operators.

Lemma 8.12 (Compatibility with $\wedge$) *Assume that* $\mathsf{f}_1 \subseteq [0, \mathsf{b})$, $\mathsf{f}_2 \subseteq [0, \mathsf{b})$, *and*

- $\forall \mathsf{nf}\ \mathsf{nf}'\ \nu\ \mathsf{N}, \mathsf{nf} \geqslant \mathsf{b} \to \mathit{tseytin}'\ \mathsf{nf}\ \mathsf{f}_1 = (\nu, \mathsf{N}, \mathsf{nf}') \to \mathsf{f}_1 \left|\!\!\left|\!\!\frac{\mathsf{nf},\mathsf{nf}'}{\mathsf{b}}\right|\!\!\right| (\nu, \mathsf{N})$,
- $\forall \mathsf{nf}\ \mathsf{nf}'\ \nu\ \mathsf{N}, \mathsf{nf} \geqslant \mathsf{b} \to \mathit{tseytin}'\ \mathsf{nf}\ \mathsf{f}_2 = (\nu, \mathsf{N}, \mathsf{nf}') \to \mathsf{f}_2 \left|\!\!\left|\!\!\frac{\mathsf{nf},\mathsf{nf}'}{\mathsf{b}}\right|\!\!\right| (\nu, \mathsf{N})$.

Then $\forall \mathsf{nf}\ \mathsf{nf}'\ \nu\ \mathsf{N}, \mathsf{nf} \geqslant \mathsf{b} \to \mathit{tseytin}'\ \mathsf{nf}\ (\mathsf{f}_1 \wedge \mathsf{f}_2) = (\nu, \mathsf{N}, \mathsf{nf}') \to (\mathsf{f}_1 \wedge \mathsf{f}_2) \left|\!\!\left|\!\!\frac{\mathsf{nf},\mathsf{nf}'}{\mathsf{b}}\right|\!\!\right| (\nu, \mathsf{N})$.

Proof By computation, we have $\mathsf{tseytin}'\ \mathsf{nf}\ \mathsf{f}_1 = (\mathsf{r}\nu_1, \mathsf{N}_1, \mathsf{nf}_1)$ and $\mathsf{tseytin}'\ \mathsf{nf}_1\ \mathsf{f}_2 = (\mathsf{r}\nu_2, \mathsf{N}_2, \mathsf{nf}_2)$ and have to show $(\mathsf{f}_1 \wedge \mathsf{f}_2) \left|\!\!\left|\!\!\frac{\mathsf{nf},1+\mathsf{nf}_2}{\mathsf{b}}\right|\!\!\right| (\mathsf{nf}_2, \mathsf{N}_1 \# \mathsf{N}_2 \# \mathsf{tseytinAnd}\ \mathsf{nf}_2\ \mathsf{r}\nu_1\ \mathsf{r}\nu_2)$. We instantiate the inductive hypotheses accordingly to obtain

- $\mathsf{f}_1 \left|\!\!\left|\!\!\frac{\mathsf{nf},\mathsf{nf}_1}{\mathsf{b}}\right|\!\!\right| (\mathsf{r}\nu_1, \mathsf{N}_1)$,
- $\mathsf{f}_2 \left|\!\!\left|\!\!\frac{\mathsf{nf}_1,\mathsf{nf}_2}{\mathsf{b}}\right|\!\!\right| (\mathsf{r}\nu_2, \mathsf{N}_2)$.

The first two goals are trivial using monotonicity of variable containment (Proposition 8.4).

For the fourth goal, we have to prove $\mathcal{E}\ \mathfrak{a}\ \mathsf{nf}_2 = \mathsf{T} \leftrightarrow \mathfrak{a} \models (\mathsf{f}_1 \wedge \mathsf{f}_2)$ under the assumption that $\mathfrak{a} \models \mathsf{N}_1 \# \mathsf{N}_2 \# \mathsf{tseytinAnd}\ \mathsf{nf}_2\ \mathsf{r}\nu_1\ \mathsf{r}\nu_2$. From the assumption, we get $\mathfrak{a} \models \mathsf{N}_1$ and $\mathfrak{a} \models \mathsf{N}_2$; therefore, the proof is straightforward from the fourth conditions of the inductive hypotheses and the definition of tseytinAnd.

The third goal is the most complicated one since we have to provide a satisfying assignment to the new variables. Assume an assignment $\mathfrak{a}$ with $\mathfrak{a} \subseteq [0, \mathsf{b})$. First of all, we apply the third conditions of the inductive hypotheses to this assignment to obtain disjoint assignments $\mathfrak{a}'_1, \mathfrak{a}'_2$ to the new variables in ranges $[\mathsf{nf}, \mathsf{nf}_1)$ and $[\mathsf{nf}_1, \mathsf{nf}_2)$ introduced by the transformation of the subformulas f_1 and f_2, and have that (1) $(\mathfrak{a} \cup \mathfrak{a}'_1) \models \mathsf{N}_1$ and (2) $(\mathfrak{a} \cup \mathfrak{a}'_2) \models \mathsf{N}_2$. In order to determine the value we should assign to the new representing variable, we make a case analysis on $\mathcal{E}\ \mathfrak{a}\ (\mathsf{f}_1 \wedge \mathsf{f}_2)$. If $\mathcal{E}\ \mathfrak{a}\ (\mathsf{f}_1 \wedge \mathsf{f}_2) = \mathsf{T}$, we choose $\mathfrak{a}' \coloneqq [\mathsf{nf}_2] \# \mathfrak{a}'_2 \# \mathfrak{a}'_1$. In the other case, we pick $\mathfrak{a}' \coloneqq \mathfrak{a}'_2 \# \mathfrak{a}'_1$. The rest of the proof is a bit technical, using the third and fourth conditions of the inductive hypotheses and the various results of Section 8.1, mainly monotonicity (Proposition 8.4) and the results on evaluation under extended assignments (Proposition 8.6). □

Lemma 8.13 (Compatiblity with $\neg$)
Assume $\mathsf{f} \subseteq [0, \mathsf{b})$ *and*

$$\forall \mathsf{nf}\ \mathsf{nf}'\ \nu\ \mathsf{N}, \mathsf{nf} \geqslant \mathsf{b} \to \mathit{tseytin}'\ \mathsf{nf}\ \mathsf{f} = (\nu, \mathsf{N}, \mathsf{nf}') \to \mathsf{f} \left|\!\!\left|\!\!\frac{\mathsf{nf},\mathsf{nf}'}{\mathsf{b}}\right|\!\!\right| (\nu, \mathsf{N}).$$

Then $\forall \mathsf{nf}\ \mathsf{nf}'\ \nu\ \mathsf{N}, \mathsf{nf} \geqslant \mathsf{b} \to \mathit{tseytin}'\ \mathsf{nf}\ (\neg \mathsf{f}) = (\nu, \mathsf{N}, \mathsf{nf}') \to (\neg \mathsf{f}) \left|\!\!\left|\!\!\frac{\mathsf{nf},\mathsf{nf}'}{\mathsf{b}}\right|\!\!\right| (\nu, \mathsf{N})$.

Proof Similar in style to the compatibility lemma for $\wedge$ (Lemma 8.12). □

Assuming that the formula does not contain $\vee$'s, we can now prove the desired statement.

Theorem 8.14 (Correctness of tseytin$'$**)**

$$\mathit{orFree}\ f \to f \subseteq [0, b) \to nf \geqslant b \to \mathit{tseytin}'\ nf\ f = (v, N, nf') \to f \left|\frac{nf, nf'}{b}\right| (v, N)$$

Proof By induction on f.

$f = \mathsf{T}$**:** The conditions of the representation relation are straightforward to verify, using monotonicity (Proposition 8.4). For the third condition, we pick the assignment $[nf]$, setting the newly introduced variable to T.

$f = v$**:** Again an easy proof. For the third condition, given an assignment $a \subseteq [0, b)$ to the original variables, we pick the same value that a assigns to v for the assignment to the newly added variable nf (as v should be equivalent to the new variable nf).

$f = f_1 \wedge f_2$**:** By Lemma 8.12.

$f = f_1 \vee f_2$**:** Contradictory.

$f = \neg f'$**:** By Lemma 8.13. □

We get the correctness statement for tseytin as a straightforward corollary, after having shown that $\left|\frac{nf, nf'}{b}\right|$ is indeed stronger than $\left|\models\right|$.

Proposition 8.15 *Let* $f \subseteq [0, b)$ *and* $nf \geqslant b$*. Then* $f \left|\frac{nf, nf'}{b}\right| (v, N) \to f \left|\models\right| (v, N)$.

Proof We have to show two directions.

- Assume that $a \models f$. By the hypothesis' third condition instantiated with the assignment $a|_b$ to the original variables, we get an assignment $a' \subseteq [nf, nf')$ to the new variables with $(a|_b \cup a') \models N$. We have to show that $(a|_b \cup a') \models (v \wedge N)$. Satisfaction of N follows by assumption. By the fourth condition of the assumption, it now suffices to show $a|_b \cup a' \models f$. We have shown that it is fine to extend an assignment by a' and still get the same evaluation if $f \subseteq [0, b)$ (Propositon 8.6). Moreover, Proposition 8.7 justifies the restriction of a to $a|_b$. Therefore the goal follows by the assumption $a \models f$.
- We have $\mathcal{E}\ a\ ([(\mathsf{T}, v)] :: N)$ for an assignment a and show that $a \models f$. This is straightforward using the fourth condition of the strengthened relation. □

Corollary 8.16 (Correctness of tseytin**)**
Assume that orFree f*. Then tseytin* $\mathsf{f} = (\nu, \mathsf{N}) \to \mathsf{f} \models\!\models (\nu, \mathsf{N})$.

The full reduction first eliminates disjunctions and then applies the Tseytin transformation.

$$\mathsf{reduction}\ \mathsf{f} \coloneqq \textbf{let}\ (\nu, \mathsf{N}) = \mathsf{tseytin}\ (\mathsf{eliminateOr}\ \mathsf{f})\textbf{in}\ [(\mathsf{T}, \nu)] +\!\!+ \mathsf{N}$$

reduction

Lemma 8.17 (FSAT reduces to **SAT)** **FSAT** $\mathsf{f} \leftrightarrow$ **SAT** (reduction f)

Proof By combining Proposition 8.9 and Corollary 8.16. □

One quickly notes that the CNFs produced by this reduction have a clause size of at most 3. We can easily adapt the reduction to yield 3-CNFs by making the clause size exactly 3. For that, we duplicate some of the literals in the clauses computed by the functions in Figure 8.2 and the function reduction. Thus, we directly obtain a reduction to 3-**SAT**, too.

Lemma 8.18 (FSAT reduces to 3-**SAT)**

$$\textbf{FSAT}\ \mathsf{f} \leftrightarrow 3\text{-}\textbf{SAT}\ (\mathsf{reduction}'\ \mathsf{f}),$$

where reduction′ is the reduction obtained by the modifications described above.

$\mathsf{size}_{\mathcal{F}}$

We close this chapter by noting that the Tseytin transformation does indeed only incur a linear increase in size, where we define the size $\mathsf{size}_{\mathcal{F}}$ of a formula and the size $\mathsf{size}_{\mathsf{cnf}}$ of a CNF (up to constant factors) as the number of nodes in the AST induced by it.

$\mathsf{size}_{\mathsf{cnf}}$

Proposition 8.19 *There are constants* $\mathsf{c}_{\mathit{eliminateOr}}$ *and* $\mathsf{c}_{\mathit{tseytin}}$ *such that:*

1. $\mathit{size}_{\mathcal{F}}\ (\mathit{eliminateOr}\ \mathsf{f}) \leqslant \mathsf{c}_{\mathit{eliminateOr}} \cdot \mathit{size}_{\mathcal{F}}\ \mathsf{f}$
2. *If* $\mathit{tseytin}'\ \mathfrak{nf}\ \mathsf{f} = (\nu, \mathsf{N}, \mathfrak{nf}')$*, then* $\mathit{size}_{\mathit{cnf}}\ \mathsf{N} \leqslant \mathsf{c}_{\mathit{tseytin}} \cdot \mathit{size}_{\mathcal{F}}\ \mathsf{f}$

Moreover, if orFree f*, then*

$$\mathit{tseytin}\ \mathsf{f} = (\nu, \mathsf{N}) \to \nu < 1 + \mathit{maxVar}\ \mathsf{f} + \mathit{size}_{\mathcal{F}}\ \mathsf{f} \wedge \mathsf{N} \subseteq [0, 1 + \mathit{maxVar}\ \mathsf{f} + \mathit{size}_{\mathcal{F}}\ \mathsf{f}).$$

The mechanisation in Coq closely follows the description on paper. The running-time analysis crucially relies on the size bounds of the previous proposition.

Theorem 8.20

1. **FSAT** $\preceq_{\mathrm{p}}$ 3-**SAT**
2. **FSAT** $\preceq_{\mathrm{p}}$ **SAT**

Chapter 9

Conclusion

In this final chapter we evaluate our results, give an overview on related work, and comment on potential future work, both from the perspective of formalisation and mechanisation.

In the preceding chapters, we have proved the following chain of reductions[1]:

$$\textbf{TMGenNP} \preceq_{\mathrm{p}} \textbf{3-PR} \preceq_{\mathrm{p}} \textbf{PR} \preceq_{\mathrm{p}} \textbf{BinaryPR} \preceq_{\mathrm{p}} \textbf{FSAT} \preceq_{\mathrm{p}} \textbf{SAT}$$

Combining these results, we obtain our following main mechanised result:

Theorem 9.1 **TMGenNP** is in NP. *Moreover, if* **TMGenNP** is NP-hard, then **SAT** is NP-complete.

We do not see this as a full result, yet. Recall that we proposed to use Turing machines as an intermediate problem for the reduction from L to **SAT** as Turing machines are structurally simpler and seem to be a good reduction target for eliminating L's size explosion problem (see Remark 2.21). Thus, a mechanised proof of the step from L to Turing machines is still missing for a proof of the Cook-Levin Theorem in L, but we think that this work contributes a significant part towards a fully mechanised proof.

As this thesis shows, formal complexity theory including running time analyses with reference to a concrete model of computation is to some extent feasible. The overhead for producing computable reductions and analysing their running time is quite noticeable, however, see Appendix A. We think that more abstract results like the Hierarchy Theorems might be mechanisable with a smaller overhead and are thus more promising candidates for future mechanisations. Most running-time bounds we have established in this thesis are not very precise as we only required polynomial bounds. In order to be able to comfortably establish precise bounds,

[1] Formally, we have shown the statements for the corresponding "flat" encodings in L.

some work needs to be done to be able to properly work with asymptotic complexities, like a suitable abstraction for constants.

9.1 Related Work

Proofs of the Cook-Levin Theorem There are a number of different proofs of the Cook-Levin Theorem available. In this thesis, we have formalised the tableau-style proof by Sipser [24, Chapter 7.4], which is in the spirit of Cook's original construction [6]. However, Cook considers the problem of determining tautologihood of DNFs, which he does by first building a CNF and then negating it. The construction is limited to Turing machines with one-sided infinite tapes.

A class of more modern proofs uses circuits. First of all, a family of circuits is designed which encodes the computation of a Turing machine for different input lengths. Then, it is shown how a circuit can be encoded as a Boolean formula. This proof is available in different flavours, for example the construction of the circuit family can proceed in a tableau-like style (e.g. in Chapter 9.3 of [24]) or by first restricting the Turing machine to be *oblivious*, such that it shows the same sequence of head movements on every input of a certain length [5, p. 199ff]. Although the circuit-based proofs might appear more elegant on paper and their ideas can be used for other proofs in circuit complexity, their mechanisation seems to be a lot more tedious as one would first have to formalise the standard notions of complexity and constructibility of circuits.

Levin [21] treats so-called *search problems* where not only the existence of a solution has to be determined, but also a certificate needs to be given. He shows a problem equivalent to **SAT** to be *universal*, but does not give an account of his proof.

Existing Mechanisation As mentioned in the introduction, there is an existing mechanisation of the translation of Turing machines to Boolean formulas in the theorem prover ACL2 [14]. They also use a tableau-style construction. However, there are a number of key differences to our mechanisation. First of all, they restrict their proof to Turing machines with one-sided infinite tapes. By this restriction, they also circumvent the problem of non-uniqueness of the representation of a tape which we explained in Chapter 4 and addressed by using a moving-tape semantics instead of the standard moving-head semantics. Their proof can use a simpler moving-head semantics. Moreover, they employ nondeterministic Turing machines and have a fixed input, which we did not choose to do because our definition of NP makes use of the verifier characterisation. Their proof ends at the problem which we have called **FSAT**, omitting the additional step to **SAT**.

From a high-level perspective, the most striking difference is that they do a *direct* reduction instead of factorising the proof as we did. This is quite interesting, as we originally thought it would not be feasible to do a direct reduction and deal with

the encoding of transitions and arbitrary finite alphabets as well as the accounting of variables in the resulting formula all at once[2]. Motivated by this thought process, we developed our factorisation and only after finishing the mechanisation became aware of the existing implementation. However, we think that our factorisation leads to the proof being much more elegant and understandable. While it seems hard to get more than a high-level intuitive grasp of the proof [14], it is our belief that our proof is quite satisfactory even on paper. Finally, the authors remark that ACL2 in its then-current state is not suitable for a running-time analysis. Due to a lack of alternatives, they do their analysis by defining a second version of their translation functions that counts the number of steps the translation takes. It is not clear at all that the resulting cost model is reasonable in the sense explained in Section 2.3.3. Because of the missing connection to a real machine model, we are of the opinion that their result does not fully include the Cook-Levin Theorem, which clearly requires to derive the NP-hardness of a (natural) problem. We see the full running-time analysis with respect to a reasonable computational model as the major advantage of our proof.

Formalising Complexity Theory There have been other approaches to formalising complexity-theoretic results. As mentioned in the introduction, Asperti has studied results like the Hierarchy Theorems and Borodin's Gap Theorem from the perspective of reverse complexity theory, examining what properties a computational model needs to satisfy to prove the mentioned theorems [3, 2]. Her results have been mechanised in the proof assistant Matita [29]. However, despite developing a formalisation of Turing machines [4], the abstract results have not been connected to a concrete computational model, yet.

Forster et al. [10] have developed a framework for the verified programming of multi-tape Turing machines in Coq, which we have been using for the definition of Turing machines. They formalise time and space measures and continue with a formally verified universal Turing machine and a multi-tape to single-tape compiler, both with a verified polynomial time and constant-factor space overhead. Despite these successes, they come to the conclusion that larger formalisations do not seem to be feasible and suggest the approach of using L we have been pursuing in this thesis.

Heiter formalises the undecidability of the Post correspondence problem in [16] by reducing from Turing machines. The reduction uses string-based rewriting systems as an intermediate problem. However, this rewriting problem does not bear much resemblance to our **PR**, with the key difference being that her construction is not suitable for resource-limited computations as needed for our reduction.

[2] We still think that this is the case if one uses two-sided infinite tapes.

9.2 Future Work

In this thesis, we have formally verified a reduction from Turing machines to **SAT**. However, as we are working in the setting of complexity theory in L, we would eventually like to obtain a mechanised reduction from L to **SAT**. The missing reduction from L to Turing machines is certainly challenging as it requires the implementation of a heap-based evaluation strategy for the lambda calculus using Turing machines[3]. The framework for the verified programming of Turing machines [10], whose formalisation of Turing machines is used in this thesis, makes this much easier as it allows to do relatively high-level programming using control-flow primitives and the possibility to encode inductive datatypes on individual tapes. An unpublished implementation of a heap-based simulation is already available in the Coq library of undecidable problems [12]. Then, a compiler from multi-tape to single-tape Turing machines is necessary, which has already been verified in [10]. Note that one can fix a constant Turing machine which takes the L-term to simulate as an input so that the running time of the multi-tape to single-tape construction need not be analysed. Currently, this reduction is work-in-progress.

Another relatively isolated line of work is to define a version of L including nondeterminism and to prove that the resulting definition of NP agrees with the verifier characterisation we use here. Such extensions to λ-calculi have been studied previously from the perspective of modelling concurrent systems, for example in [19], but never from the perspective of obtaining a reasonable computational model. This might also open the door to formalise other results of complexity theory which rely on nondeterminism.

For the proofs in this thesis, we were mostly only concerned with the time usage of a reduction and not its space usage (we only made sure that it does not exhibit size explosion). It would be interesting to formalise space classes like LogSpace and PSpace and prove results like Savitch's Theorem which relates nondeterministic and deterministic space. For a mechanisation to be possible, first the extraction framework would need to be expanded to also derive recurrences for the space usage of a term.

On a more general note, the expansion of the reasonability result of [9] beyond decision problems would be interesting in order to facilitate the mechanisation of results outside of the mere complexity theory of decision problems. It seems like one might need to investigate the matter of size explosion in more detail in order for this to work, maybe resulting in general conditions to rule out size explosion or a direct scheme to compress size-exploding terms.

[3] Note that the more complicated interleaving strategy of [9] is not necessary as we do not care about the space overhead.

Finally, on the more technical side there is the issue of binary encodings. Throughout this thesis we used a unary encoding for numbers. For the problems we considered this is reasonable (or even necessary, as with the encoding of the number of steps in **TMGenNP**), but many number-theoretic problems are not NP-hard anymore if one employs a unary encoding instead of a binary encoding[4]. An example for this is the **SubsetSum** problem, asking whether there is a subset $A \subseteq B$ of a given set of numbers B such that A sums exactly to a number c. However, almost all Coq standard library functions are defined with respect to the Peano numbers $\mathbb{N}$, whose Scott encoding is unary. Having to manually redefine them for binary numbers would be rather unpleasant, especially as writing recursive functions on binary numbers is more difficult[5]. A translation can usually be done mechanically, which is why we feel like it might be feasible to implement an automatic translation plugin using the MetaCoq project [30] for meta progamming in Coq. This plugin could also automatically prove agreement with the original unary definition. Alternatively, the translation could happen directly at the level of the extraction plugin so that the Peano numbers are extracted to a binary encoding instead of their Scott encoding.

[4] or any other c-nary encoding for $c > 1$

[5] The Peano recursor for binary numbers seems like the best way, but clutters up the function's definition. Arithmetical operations like addition and multiplication should, of course, still be defined via direct recursion on binary numbers.

Appendix A

Structure of the Mechanisation

We give an overview of the structure of our mechanisation. Our development is based on the Coq library of undecidable problems [12], which contains among other things a formalisation of L and Turing machines as well as the extraction framework. The files we contribute mainly live in the `/L/Complexity` directory of the project. The project is hosted on GitHub.

Basic Definitions For completeness, we start with an overview over the basic definitions for complexity theory which were developed by Fabian Kunze.

Component	Spec	Proof
$\mathcal{O}$-notation and monotonicity	70	167
decidability & computability in time	118	182
P, NP and reductions	102	157

The basic definitions we need are quite compact. The files for $\mathcal{O}$ notation mostly contain automation, for instance for proving that a particular function is polynomial or monotonic. The file for NP includes basic facts like properties of polynomial-time reductions and the inclusion $\mathsf{P} \subseteq \mathsf{NP}$.

Throughout the thesis, we use a small library of additional facts about lists and other things that are not yet present in the Coq standard library or the undecidability library. Moreover, we have a shared file providing running time and size bounds for common functions.

Component	Spec	Proof
preliminaries	169	381
polynomial bounds	112	253

Problem Definitions The definitions of the problems we use are contained in the subfolder `Problems`.

Component	Spec	Proof
SAT	201	446
FSAT	93	87
k-**SAT**	26	42
Clique & **FlatClique**	194	263
variants of **Parallel Rewriting**	551	1022
TMGenNP	28	50

The files for extractable problems include the respective extraction to L. For example, the extraction of **SAT** makes up 84 lines of specification and 247 lines of proof of the total number of lines. The definitions for flat problems include predicates connecting them to the corresponding non-flat version. In total, we have five variants of Parallel Rewriting: **PR**, **FlatPR**, **3-PR**, **Flat-3-PR** and **BinaryPR**. As the **3-PR** variant fixes all definitions to the special case of the width being 3 and the offset being 1, most definitions and lemmas have to be stated again for the variant. The file on generic problems for Turing machines explores different definitions and proves their equivalence.

Reduction to Clique

Component	Spec	Proof
pigeonhole principle	64	150
k-**SAT** to **Clique**	140	249
k-**SAT** to **FlatClique**	176	333
k-**SAT** to **SAT**	6	41

While the (not extractable) proof using the abstract representation of graphs is fairly compact, with a major part component being a proof of the pigeonhole principle adapted from the ICL 2019 course [17], defining a computable variant and analysing its running time takes up some space. For completeness, we include a reduction from k-**SAT** to **SAT** proving that it is contained in NP.

Proof of the Cook-Levin-Theorem Recall that our proof of the Cook-Levin Theorem consists of four main reductions and a minor embedding of **3-PR** into **PR**.

Component		Spec	Proof
TMGenNP to **3-PR**	reduction	1843	2481
	time analysis	838	1706
3-PR to **PR**		37	174
PR to **BinaryPR**		222	719
BinaryPR to **FSAT**		312	1078
FSAT to **SAT**		213	605

The reduction of **TMGenNP** to **3-PR** includes the formalisation of preludes for

PR needed for guessing the certificate. 735 lines of specification of 909 lines of the reduction's correctness proof are for generating the list-based windows and proving their agreement with the inductive predicates. The running-time analysis makes up a very significant part of the whole reduction.

The remaining reductions are much simpler, with the reduction to **BinaryPR** including a formalisation of uniform homomorphisms and their action on **PR** instances.

In total, we have contributed 5230 lines of specification and 10038 lines of proof, of which 1636 and 3181 lines are for the extraction/running-time analysis (not including the definition of the flat problems or the flat reductions). If we additionally exclude the flat problems, flat reductions and any proofs of computability (i.e. also exclude the definition of the list-based rules), we are left with 2339 lines of specification and 4524 lines of proof. These numbers thus only capture the correctness of the construction, but do not even include what is needed for a reduction in synthetic computability theory (as it is not at all clear how one would, in general, translate inductive predicates to something which is computable).

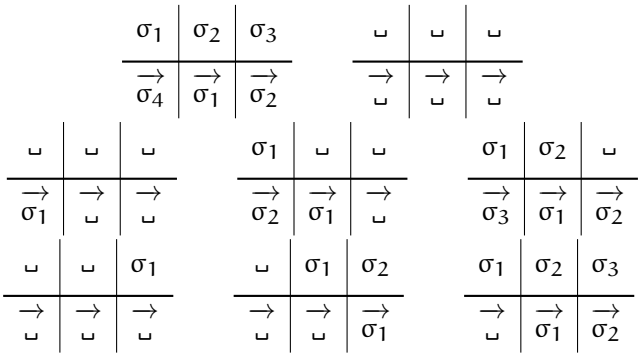
Appendix B

Full Rewrite Rules

The following list contains the full set of rewrite rules as implemented by the Coq mechanisation.

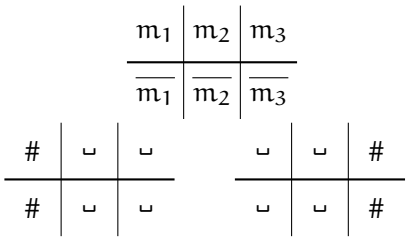
B.1 Tape Rules

Right Shifts



Left Shifts The rules for left shifts are defined to be the polarity-reversion of the rules for shifting to the right, i.e. $[\gamma_1, \gamma_2, \gamma_3] \,/\, [\gamma_4, \gamma_5, \gamma_6]$ is a rule for shifting to the left if, and only if, $[{\sim}\gamma_3, {\sim}\gamma_2, {\sim}\gamma_1] \,/\, [{\sim}\gamma_6, {\sim}\gamma_5, {\sim}\gamma_4]$ is a rule for shifting to the right.

Identity Rules



B.2 Transition Rules

$\delta(q, {}^{\circ}a) = (p, {}^{\circ}b, L)$:

m_1	q^a	m_2
$\overrightarrow{m_3}$	p^{m_2}	$\overrightarrow{b}$

m_1	m_2	q^a
$\overrightarrow{m_3}$	$\overrightarrow{m_1}$	p^{m_2}

q^a	m_1	m_2
p^{m_3}	$\overrightarrow{a}$	$\overrightarrow{m_1}$

.

$\delta(q, {}^\circ a) = (p, {}^\circ b, R)$:

m_1	q^a	m_2
$\overleftarrow{b}$	p^{m_2}	$\overleftarrow{m_3}$

q^a	m_1	m_2
p^{m_1}	$\overleftarrow{m_2}$	$\overleftarrow{m_3}$

m_1	m_2	q^a
$\overleftarrow{m_2}$	$\overleftarrow{a}$	p^{m_3}

$\delta(q, {}^\circ a) = (p, {}^\circ b, N)$:

m_1	q^a	m_2
$\overline{m_1}$	p^b	$\overline{m_2}$

q^a	m_1	m_2
p^b	$\overline{m_1}$	$\overline{m_2}$

m_1	m_2	q^a
$\overline{m_1}$	$\overline{m_2}$	p^b

$\delta(q, {}^\circ a) = (p, \emptyset, L)$:

m_1	q^a	m_2
$\overrightarrow{m_3}$	p^{m_2}	$\overrightarrow{a}$

m_1	m_2	q^a
$\overrightarrow{m_3}$	$\overrightarrow{m_1}$	p^{m_2}

q^a	m_1	m_2
p^{m_3}	$\overrightarrow{a}$	$\overrightarrow{m_1}$

$\delta(q, {}^\circ a) = (p, \emptyset, R)$:

m_1	q^a	m_2
$\overleftarrow{a}$	p^{m_2}	$\overleftarrow{m_3}$

q^a	m_1	m_2
p^{m_1}	$\overleftarrow{m_2}$	$\overleftarrow{m_3}$

m_1	m_2	q^a
$\overleftarrow{m_2}$	$\overleftarrow{a}$	p^{m_3}

$\delta(q, {}^\circ a) = (p, \emptyset, N)$:

m_1	q^a	m_2
$\overline{m_1}$	p^a	$\overline{m_2}$

q^a	m_1	m_2
p^a	$\overline{m_1}$	$\overline{m_2}$

m_1	m_2	q^a
$\overline{m_1}$	$\overline{m_2}$	p^a

$\delta(q, \emptyset) = (p, {}^\circ b, L)$:

m_1	$q^\sqcup$	m_2
$\overrightarrow{m_3}$	p^{m_2}	$\overrightarrow{b}$

m_1	m_2	$q^\sqcup$
$\overrightarrow{m_3}$	$\overrightarrow{m_1}$	p^{m_2}

$q^\sqcup$	m_1	m_2
p^{m_3}	$\overrightarrow{a}$	$\overrightarrow{m_1}$

$\delta(q, \emptyset) = (p, {}^\circ b, R)$:

m_1	$q^\sqcup$	m_2
$\overleftarrow{b}$	p^{m_2}	$\overleftarrow{m_3}$

$q^\sqcup$	m_1	m_2
p^{m_1}	$\overleftarrow{m_2}$	$\overleftarrow{m_3}$

m_1	m_2	$q^\sqcup$
$\overleftarrow{m_2}$	$\overleftarrow{a}$	p^{m_3}

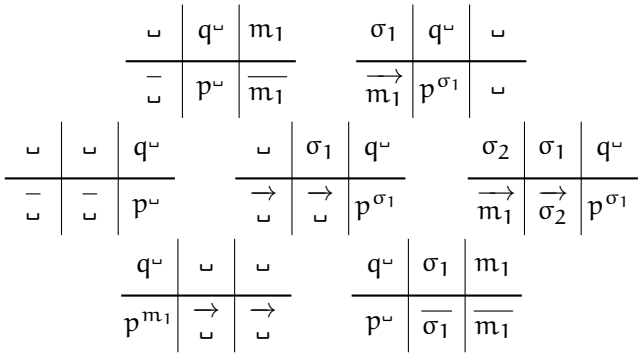
$\delta(q, \emptyset) = (p, {}^\circ b, N)$:

m_1	$q^\sqcup$	m_2
$\overline{m_1}$	p^b	$\overline{m_2}$

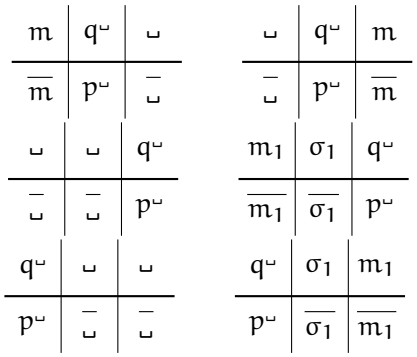
$q^\sqcup$	m_1	m_2
p^b	$\overline{m_1}$	$\overline{m_2}$

m_1	m_2	$q^\sqcup$
$\overline{m_1}$	$\overline{m_2}$	p^b

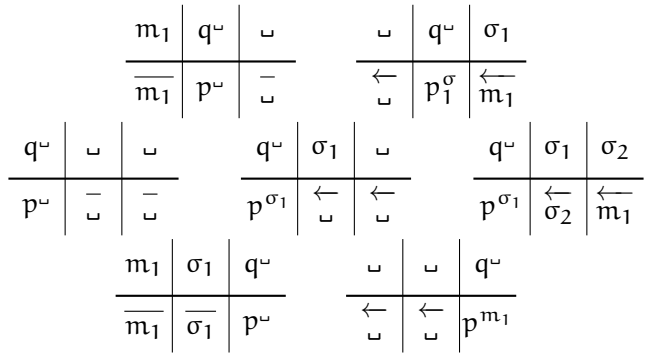
$\delta(q, \emptyset) = (p, \emptyset, L)$:



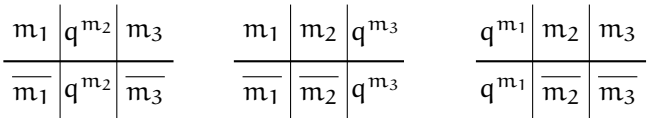
$\delta(q, \emptyset) = (p, \emptyset, N)$:



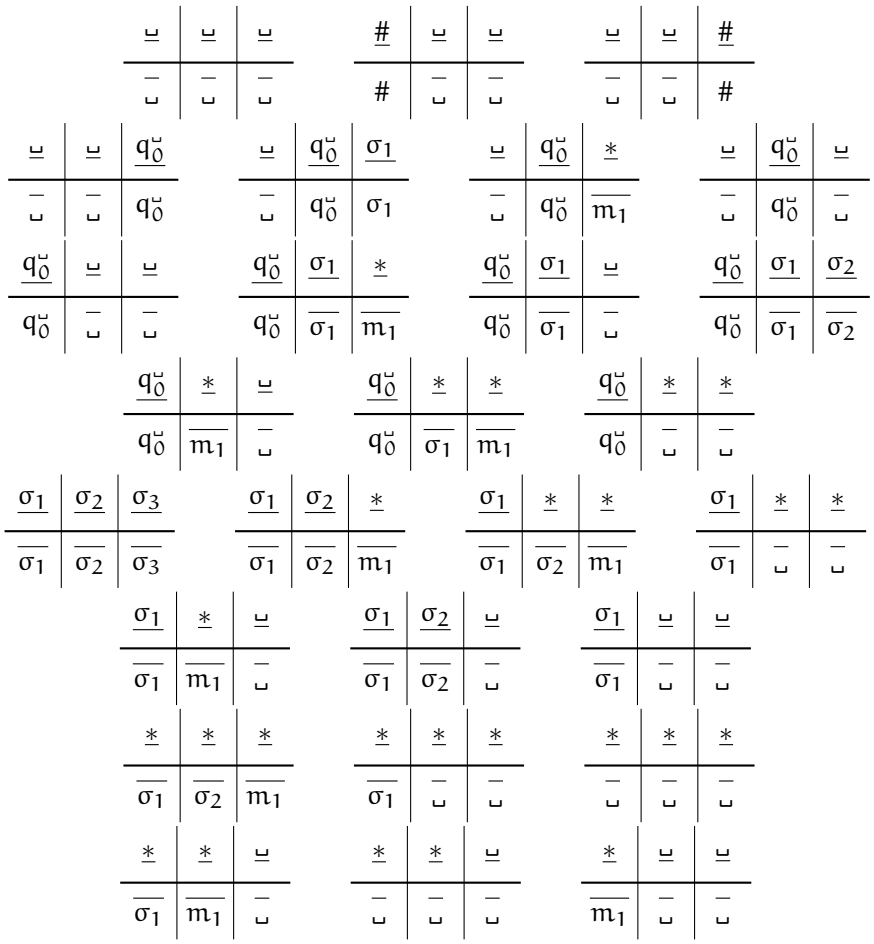
$\delta(q, \emptyset) = (p, \emptyset, R)$:



B.3 Halting Rules



B.4 Prelude Rules



Appendix C

Semantics of Turing Machines

In this chapter, we present the parts of the semantics of Turing machines that were left out in Section 2.2. Let us fix a Turing machine tape alphabet Σ : finType and a Turing machine $M = (Q, \delta, \mathsf{start}, \mathsf{halt}) : \mathsf{TM}\ \Sigma\ n$.

We start with the functions left, right : $\mathsf{tape}_\Sigma \to \mathcal{L}(\Sigma)$ and current : $\mathsf{tape}_\Sigma \to \mathbb{O}(\Sigma)$ giving the parts of the tape left, right, or under the head.

$$\begin{aligned} \mathsf{left}\ \ \mathsf{niltape} &\coloneqq [\,] & \mathsf{right}\ \ \mathsf{niltape} &\coloneqq [\,] \\ \mathsf{left}\ \ (\mathsf{leftof}\ r\ rs) &\coloneqq [\,] & \mathsf{right}\ \ (\mathsf{leftof}\ r\ rs) &\coloneqq r :: rs \\ \mathsf{left}\ \ (\mathsf{rightof}\ l\ ls) &\coloneqq l :: ls & \mathsf{right}\ \ (\mathsf{rightof}\ l\ ls) &\coloneqq [\,] \\ \mathsf{left}\ \ (\mathsf{midtape}\ ls\ c\ rs) &\coloneqq ls & \mathsf{right}\ \ (\mathsf{midtape}\ ls\ c\ rs) &\coloneqq rs \end{aligned}$$

$$\begin{aligned} \mathsf{current}\ \ (\mathsf{midtape}\ ls\ c\ rs) &\coloneqq {}^\circ c \\ \mathsf{current}\ \ \_ &\coloneqq \emptyset \end{aligned}$$

The function tape_write writes a symbol to the current position of a tape.

$$\begin{aligned} \mathsf{tape\_write} &: \mathsf{tape}_\Sigma \to \mathbb{O}(\Sigma) \to \mathsf{tape}\ \Sigma \\ \mathsf{tape\_write}\ t\ \emptyset &\coloneqq t \\ \mathsf{tape\_write}\ t\ ({}^\circ s) &\coloneqq \mathsf{midtape}\ (\mathsf{left}\ \ t)\ s\ (\mathsf{right}\ t) \end{aligned}$$

Recall the function tape_move : $\mathsf{tape}_\Sigma \to \mathsf{move} \to \mathsf{tape}_\Sigma$ we have defined on Page 10 that moves a tape in a given direction. The function doAct : $\mathsf{tape}_\Sigma \to \mathsf{Act}_\Sigma \to \mathsf{tape}_\Sigma$ performs an action on a tape by first writing a symbol and then moving the tape.

$$\mathsf{doAct}\ t\ (s, m) \coloneqq \mathsf{tape\_move}\ (\mathsf{tape\_write}\ t\ s)\ m$$

In order to perform a computational step, we start by gathering the symbols under the heads and then lookup the value of the transition function to perform the actions given by the transition function on all tapes.

$$\mathsf{step} : \mathsf{conf}_M \to \mathsf{conf}_M$$
$$\mathsf{step}\ (q, ts) \coloneqq \textbf{let}\ (q', as) \coloneqq \delta(q, [\ \mathsf{current}\ \ t \mid t \in ts\ ])\textbf{in}\ (q', [\ \mathsf{doAct}\ t\ a \mid t \in ts, a \in as\ ])$$

The execution is defined computationally. We first define an abstract loop function.

$$\mathsf{loop} : \forall A, (A \to A) \to (A \to \mathbb{B}) \to A \to \mathbb{N} \to \mathbb{O}(A)$$
$$\mathsf{loop}\ f\ p\ a\ k \coloneqq \textbf{if}\ p\ a\ \textbf{then}\ {}^\circ a\ \textbf{else match}\ \ k\ [\ 0 \Rightarrow \emptyset \mid S\ k' \Rightarrow \mathsf{loop}\ f\ p\ (f\ a)\ k'\ ]$$

Intuitively, f is a step function, p is a predicate that is T when the loop should break, a is the initial value, and k is the maximum number of steps. If the loop terminates within at most k steps, the final value is returned. Before we give the loop function for Turing machines, we fix initial configurations and halting configurations.

$$\mathsf{initConf}\ t \coloneqq (\mathsf{start}, t)$$
$$\mathsf{haltingConf}\ (q, t) \coloneqq \mathsf{halt}\ q$$

Now, we can instantiate loop suitably to obtain a function executing the Turing machine M.

$$\mathsf{loopTM} : \mathsf{Conf}_M \to \mathbb{N} \to \mathbb{O}(\mathsf{Conf}_M)$$
$$\mathsf{loopTM}\ c\ \mathsf{steps} \coloneqq \mathsf{loop}\ \mathsf{step}\ \mathsf{haltingConf}\ c\ \mathsf{steps}$$
$$\mathsf{execTM}\ t \coloneqq \mathsf{loopTM}\ (\mathsf{initConf}\ t)$$

C.1 Single-tape Turing Machines

For the whole of Chapter 5, we have used simpler definitions for single-tape Turing machines. In the following, we present the derived definitions.

$$\mathsf{sconf}_M \coloneqq Q \times \mathsf{tape}_\Sigma$$
$$\mathsf{trans}_M\ ((q, t) : \mathsf{sconf}_M) \coloneqq \textbf{let}\ (q', [a]) \coloneqq \delta(q, [t])\textbf{in}\ (q', a)$$
$$\mathsf{tepsstep}\ ((q, t) : \mathsf{sconf}_M) \coloneqq \textbf{let}\ (q', a) \coloneqq \mathsf{trans}_M(q, \mathsf{current}\ \ t)\textbf{in}\ (q', \mathsf{doAct}\ tp\ a)$$

Here we use the same notations for vectors as for lists, i.e. $[a]$ is a singleton vector in the definition of trans_M, for instance. We define the following relations:

$$(q, t) \succ (q', t') \coloneqq \mathsf{halt}\ q = F \wedge \mathsf{sstep}\ s = s'$$
$$(q, tp) \rhd^k (q', tp') \coloneqq (q, tp) \succ^k (q', tp') \wedge \mathsf{halt}\ q' = T$$
$$(q, tp) \rhd^{\leqslant k} (q', tp') \coloneqq \exists l \leqslant k, (q, tp) \rhd^l (q', tp')$$

One can show that these relational definitions agree with loopTM in the expected way.

Bibliography

[1] Beniamino Accattoli. "(In)Efficiency and Reasonable Cost Models". In: *Electronic Notes in Theoretical Computer Science* 338 (Oct. 2018), pp. 23–43. DOI: 10.1016/j.entcs.2018.10.003.

[2] Andrea Asperti. "A Formal Proof of Borodin-Trakhtenbrot's Gap Theorem". In: *Certified Programs and Proofs*. Ed. by Georges Gonthier and Michael Norrish. Cham: Springer International Publishing, 2013, pp. 163–177. ISBN: 978-3-319-03545-1.

[3] Andrea Asperti. "Reverse Complexity". In: *Journal of Automated Reasoning* 55 (2015), pp. 373–388.

[4] Andrea Asperti and Wilmer Ricciotti. "A formalization of multi-tape Turing machines". In: *Theoretical Computer Science* 603 (July 2015). DOI: 10.1016/j.tcs.2015.07.013.

[5] Markus Bläser. "Theoretical Computer Science: An Introduction". 2020.

[6] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[7] Yannick Forster, Dominik Kirst, and Gert Smolka. "On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem". In: *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. New York, NY, USA: ACM, Jan. 2019.

[8] Yannick Forster and Fabian Kunze. "A certifying extraction with time bounds from Coq to call-by-value Lambda Calculus". In: *Interactive Theorem Proving - 10th International Conference, ITP 2019, Portland, USA*. Also available as arXiv:1904.11818. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Apr. 2019, 17:1–17:19.

[9] Yannick Forster, Fabian Kunze, and Marc Roth. "The Weak Call-By-Value Lambda Calculus is Reasonable for Both Time and Space". In: *47th ACM SIG-*

PLAN Symposium on Principles of Programming Languages (POPL 2020), New Orleans, USA. ACM, Jan. 2020.

[10] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. "Verified Programming of Turing Machines in Coq". In: *9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020*. New York, NY, USA: ACM, 2020.

[11] Yannick Forster and Gert Smolka. "Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq". In: *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasilia, Brazil, September 26-29, 2017*. Apr. 2017.

[12] Yannick Forster et al. "A Coq Library of Undecidable Problems". In: CoqPL '20.

[13] Lennard Gäher. *Memo: Polynomial-time reduction from 3SAT to Clique*. Accessed: 20.03.2020. URL: https://www.ps.uni-saarland.de/~gaeher/files/3SATClique.pdf.

[14] Ruben Gamboa and John Cowles. "A Mechanical Proof of the Cook-Levin Theorem". In: *Theorem Proving in Higher Order Logics*. Ed. by Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 99–116. ISBN: 978-3-540-30142-4.

[15] Juris Hartmanis and R. Stearns. "On the Computational Complexity of Algorithms". In: *Transactions of The American Mathematical Society - TRANS AMER MATH SOC* 117 (May 1965), pp. 285–285. DOI: 10.2307/1994208.

[16] Edith Heiter. "Undecidability of the Post Correspondence Problem in Coq". Bachelor's thesis. 2017.

[17] *ICL 2019 Coq Files*. Accessed: 30.03.2020. URL: https://courses.ps.uni-saarland.de/icl_19/2/Resources.

[18] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[19] Arne Kutzner and Manfred Schmidt-Schauß. "A Non-Deterministic Call-by-Need Lambda Calculus". In: *SIGPLAN Not.* 34.1 (Sept. 1998), pp. 324–335. ISSN: 0362-1340. DOI: 10.1145/291251.289462. URL: https://doi.org/10.1145/291251.289462.

[20] Dominique Larchey-Wendling and Yannick Forster. "Hilbert's Tenth Problem in Coq". In: *4th International Conference on Formal Structures for Computa-*

tion and Deduction, FSCD 2019, Dortmund, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Feb. 2019, 27:1–27:20.

[21] L. A. Levin. "Universal Sequential Search Problems". In: vol. 9. [Probl. Peredachi Inf., **9**,3:115-116, 1973]. 1973, pp. 265–266.

[22] Jan Christian Menz. "A Coq Library for Finite Types". Bachelor's thesis. 2016.

[23] *Millenium Problems*. Accessed: 01.02.2020. Clay Mathematics Institute. URL: http://www.claymath.org/millennium-problems.

[24] Michael Sipser. *Introduction to Theory of Computation*. 1st ed. PWS Publishing Company, 1997.

[25] Cees Slot and Peter van Emde Boas. "On Tape Versus Core; an Application of Space Efficient Perfect Hash Functions to the Invariance of Space." In: *Elektronische Informationsverarbeitung und Kybernetik* 21 (Jan. 1985), pp. 246–253. DOI: 10.1145/800057.808705.

[26] Gert Smolka. *Abstract Reduction Systems*. Accessed: 09.03.2020. Nov. 2019. URL: http://www.ps.uni-saarland.de/courses/sem-ws19/confluence.pdf.

[27] Gert Smolka. *Computational Type Theory and Interactive Theorem Proving with Coq*. Accessed: 01.02.2020. Aug. 2019. URL: http://www.ps.uni-saarland.de/~smolka/drafts/icl2019.pdf.

[28] *The Coq Proof Assistant*. Accessed: 01.02.2020. URL: https://coq.inria.fr/.

[29] *The Matita Proof Assistant*. Accessed: 16.03.2020. URL: http://matita.cs.unibo.it/.

[30] *The MetaCoq Project*. Accessed: 12.03.2020. URL: https://github.com/MetaCoq/metacoq.

[31] G. S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28. URL: https://doi.org/10.1007/978-3-642-81955-1_28.