Mechanising Complexity Theory: The Cook-Levin Theorem in Coq

Lennard Gäher
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Fabian Kunze
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

We mechanise the Cook-Levin theorem, i.e. the NP-completeness of SAT, in the proof assistant Coq. We use the call-by-value λ-calculus L as the model of computation to formalise time complexity, the class NP, and polynomial-time reductions. The latter two notions agree with the usual characterisations via Turing machines (TMs), as L and TMs are polynomial-time equivalent.

The use of L as the computational model, as opposed to TMs, significantly eases program verification and the derivation of resource bounds. However, for showing the NP-hardness of SAT, computations of L need to be encoded in SAT, which is complicated by L's more complex computational structure. Thus, the polynomial-time reduction chain to SAT employs TMs as an intermediate *problem*, for which we neatly factor out a known textbook reduction from TMs to SAT. Still, all *reduction functions* are implemented and analysed in L.

To the best of our knowledge, this is the first result in *computational* complexity theory that has been mechanised with respect to any concrete computational model.

We discuss what makes this area of computer science hard to mechanise and highlight the design choices which enable our mechanisations.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Theory of computation → Complexity classes; Theory of computation → Lambda calculus; Theory of computation → Logic and verification; Theory of computation → Constructive mathematics

Keywords and phrases computational model, NP completeness, Coq, Cook, Levin

Supplementary Material `https://github.com/uds-psl/cook-levin`

Funding *Lennard Gäher*: was supported in part by the International Max-Planck Research School on Trusted Computing (IMPRS-TRUST)

1 Introduction

Computational complexity theory studies how efficiently problems can be solved. This subsumes the mere analysis of the resource usage of specific algorithms: Problems are classified according to their inherent (time and space) resource requirement, revealing a rich structure of relations between these complexity classes. Even practitioners make use of NP, the class of problems for which solution candidates can be verified on polynomial time: Showing that a problem p is NP-hard, i.e. at least as hard as the hardest problem in NP, is an established criterion for the infeasibility of solving p efficiently in the general case. And NP is even known outside the area of computer science due to one of the most famous[1] open questions of modern mathematics, whether $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, i.e. whether every problem whose solutions can be verified efficiently can also be solved efficiently.

Despite their relevance, complexity-theoretic results are seldom if ever proven in all detail, e.g. relying on the reader's intuition for the exact implementation or resource bounds of

[1] `https://www.claymath.org/millennium-problems/p-vs-np-problem`

algorithms. While computability-theoretic results have been successfully mechanised in proof assistants [15, 6, 24], few mechanisations of even basic complexity theory are available. For computability theory, *synthetic* approaches [9, 4] have proved successful, by avoiding the use of an explicit computational model and relying on the fact that all functions definable in constructive theories such as Coq's are computable. Many results in *computational complexity theory*, on the other hand, inherently require proofs with respect to a concrete model of computation, for two reasons: First, the resource analysis of "programs" needs to be carried out with a *reasonable* resource model that is connected to the definitions of complexity classes, which rules out a synthetic approach. Secondly, many *structural* results involve universal quantifications over problems in certain complexity classes, necessitating the use of properties of an underlying computational model. For instance, showing that a problem p is NP-complete requires proving that *all* problems q verifiable in polynomial-time reduce to p in polynomial time. Thus, the essential requirements for mechanising complexity theory are to be able to program in a model of computation and to use properties of it.

Related Work Past mechanisations of computational complexity theory have usually either assumed an abstract computational model satisfying certain conditions, but have not proved the existence of such a model, or have focused on merely proving functional correctness of constructions from complexity theory, without connecting the result to a cost model proved reasonable.

Asperti [2, 1] axiomatically proves the hierarchy theorems and Borodin's gap theorem in the setting of *reverse complexity theory* using the proof assistant Matita, in an effort to determine the minimal assumptions on a computational model needed to obtain the theorems.

Gamboa and Cowles [17] verify the construction of the Cook-Levin theorem for Turing machines in ACL2. They define functions counting the steps taken by their implementation, and, without any computational model, cannot establish any connection to the class NP.

Other attempts [13] have been made at mechanising basic complexity theory using Turing machines (TMs) as the computational model. Forster et al. [13] implement a multi-tape to single-tape Turing machine compiler and a universal Turing machine, both with polynomial time overhead and constant-factor space overhead. They conclude, after implementing a framework for programming TMs consisting of 19K lines of Coq code, that 'TMs as model of computation are inherently infeasible for the formalisation of any computability or complexity theoretic result', owing to the non-compositionality and low-level structure of TMs. Instead, they conjecture that using a λ-calculus as model of computation may prove more successful.

Key Idea: L for Complexity Theory We follow the idea of using a λ-calculus and choose the call-by-value λ-calculus L [15] as our model of computation. One big overhead of mechanised complexity theory is the verification and resources analyse of algorithms in the model of computation. It is simple to compositionally program in L since inductive datatypes and recursive functions can be encoded systematically, similar to functional programming languages. We use the certifying extraction mechanism from Coq to L by Forster and Kunze [10] to prove computability and semi-interactively deduce the running time of algorithms. This allows us to program and verify algorithms using the usual comforts of Coq, automatically establish computability in L, and derive resource bounds in Coq almost completely without descending into the computational model.

L has been shown to be a *reasonable* computational model [11], in the sense of the invariance thesis of Slot and van Emde Boas [27]: TMs and L can simulate each other with a polynomial overhead in time and a constant-factor overhead in space for decision problems. Thus, one obtains the same polynomial-time complexity classes for L as for TMs, which justifies using L for the mechanisation of complexity theory. The two key requirements for

mechanising *structural* results are thus elegantly satisfied by L.

Overview In this paper, we formalise the basics of polynomial-time complexity theory for L, namely the classes P and NP, as well as the notions of NP-hardness and NP-completeness.

As a first example of using L for complexity theory, we mechanise the popular Cook-Levin theorem in Coq. This theorem essentially founded the class of NP-complete problems when it was independently discovered by Cook [7] and Levin [22] in 1971. In the formulation by Cook, the satisfiability problem of Boolean formulas SAT is shown NP-complete.

The importance of the Cook-Levin theorem lies not in proving that *specifically* SAT is NP-complete, but that a *natural*, *machine-independent* problem is NP-complete. Once such a problem p has been shown to be NP-complete, it is relatively easy to derive NP-completeness of other problems by reduction from p.

In contrast, in any machine model one can usually define a *generic* NP*-complete problem* which is machine-dependent and usually straightforward to prove NP-complete. Essentially, the existence of an input accepted by a polynomial-time-verifier is reformulated as a problem on triples of a program (the verifier), a size bound, and a time bound.

Usually, the proof of the Cook-Levin theorem proceeds by reduction from a generic problem. Specifically, for L this would require us to encode the computation of L-terms using Boolean formulas. As λ-calculi have a very much non-local computational structure due to variable binding, formally verifying a direct encoding seems intractable.

On the other hand, encodings of TMs are conceptually simple (though still technically challenging): single-tape TMs can only locally change a constant amount of data in each step of computation. Therefore, we use TMs as an intermediate problem: L is encoded using TMs (employing previous related mechanisations [13, 12]) and in turn TMs are encoded as Boolean formulas. The expressivity of TMs seems better-suited than Boolean formulas for an encoding of L-computations. The full reduction is still computed and analysed in L. TMs serve only as an intermediate construct.

Definitions, lemmas, and theorems in this document are hyperlinked with the documentation of the Coq development; the links are marked by the symbol .

Contribution In summary, our contributions are as follows:

1. We mechanise the theory of polynomial-time complexity for L, in particular the notions of polynomial-time computability, NP, NP-hardness, and NP-completeness,
2. and provide the first mechanisation of a *structural* result in complexity theory, namely the Cook-Levin theorem, that includes a running time analysis with respect to the reasonable computational model L, without any axioms.
3. Our reduction from Turing machines to SAT neatly factorises and formalises a textbook proof by Sipser [26].
4. We have developed techniques and automation to make time resource analyses on top of the L extraction framework [10] more feasible.

Structure Section 2 introduces notation and type-theoretic preliminaries, as well as the most important aspects of TMs. We then continue with a detailed account of our basic complexity definitions in Section 3. Section 4 contains a high-level overview of the reduction and the involved problems. Section 5 and Section 6 elaborate on some details of the reductions. Section 7 gives a general overview of the mechanisation and in particular the mechanisation of resource analyses. Finally, we discuss our results and future work in Section 8.

2 Preliminaries

We work in a constructive type theory with inductive types and an impredicative universe of propositions $\mathbb{P}$ as implemented in Coq.

The basic inductive types we use are the Booleans $\mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$, the unit type $\mathbb{1} ::= ()$, the natural numbers $\mathbb{N} ::= 0 \mid \mathsf{S}\, n$ for $n : \mathbb{N}$, product types $X \times Y$, and sum types $X + Y$. π_1 and π_2 denote the projections out of the product type. Given a type X, we further define options $\mathcal{O}(X) ::= \emptyset \mid \lfloor x \rfloor$ and lists (or strings) $X^* ::= [] \mid x :: A$ for $x : X$ and $A : X^*$. On lists we employ the standard notation for membership $x \in A$, inclusion $A \subseteq B$, concatenation $A +\!\!+ B$, and length $|A|$. The notation $A[i, j]$ is used to denote the sublist of A from positions i to j (inclusive), possibly having less than $j - i + 1$ elements if the indices are (partially) invalid. $[a, b]$ denotes the list of numbers between a and b (inclusive, possibly empty). $[\ f\ a \mid a \in A\ ]$ is the list obtained by mapping f over A.

We say that a type X is discrete if there is a function $X \to X \to \mathbb{B}$ deciding equality. We say that a type X is finite if it is discrete and there is an exhaustive list of elements $[x_0, \ldots, x_n]$. In this case, we may write $\{x_1, \ldots, x_n\}$ for X. In particular, we will use the letters Σ, Γ to denote finite types representing alphabets.

Turing Machines We use the formalisation of deterministic, two-sided infinite TMs by [3] and the mechanisation by [13]. Unlike standard presentations, the tape alphabet Σ does not feature a distinguished blank symbol. Although we make use of multi-tape machines, for this exposition we restrict our attention to single-tape machines. We assume a type of tapes Tape_Σ with operations $\mathsf{left}, \mathsf{right} : \mathsf{Tape}_\Sigma \to \Sigma^*$ giving the parts of the tape left and right of the head, as well as $\mathsf{current} : \mathsf{Tape}_\Sigma \to \mathcal{O}(\Sigma)$ yielding an optional symbol under the head. $|tp|$ denotes the number of symbols on a tape tp.

A Turing Machine $M : \mathsf{TM}_\Sigma$ is a tuple $(Q, q_0, \delta, \mathsf{halt})$ consisting of a finite type of states Q, an initial state q_0, a transition function $\delta : Q \times \mathcal{O}(\Sigma) \to Q \times \mathcal{O}(\Sigma) \times \mathsf{Move}$, where $\mathsf{Move} ::= \mathsf{L} \mid \mathsf{N} \mid \mathsf{R}$, and a function determining the halting states. A configuration is a pair (q, tp) of a state and a tape. We use the notation $(q, tp) \succ (q', tp')$ for configuration changes according to δ and write $(q, tp) \rhd^{\leq t} (q', tp')$ for $\leq t$ steps where additionally $\mathsf{halt}(q') = \mathsf{true}$.

3 Polynomial-Time Complexity Theory in L

L [15], the underlying model of computation in this work, is a standard untyped λ-calculus with weak call-by-value reduction $\succ$. This section presents the relevant definitions for L and the definitions of standard notions of polynomial-time complexity theory, adapted to L as the computational model.

Its terms $s, t : \mathsf{Ter} ::= x \mid \lambda x.t \mid s\ t$ feature lambda abstractions and applications (the formalisation uses de-Bruijn indices instead of named variables). We use $\|s\|$ to denote the (syntactic) size of a term s. The reduction relation $\succ$ is uniformly confluent [16], meaning that each reduction path to a normal form has the same length. This allows us to speak about *the* number of steps that a terminating term takes. Many types X can be encoded as L-terms by defining an encoding function $\ulcorner \cdot \urcorner : X \to \mathsf{L}$. Inductive first-order datatypes (like Peano numbers and lists) can be represented systematically using the Scott encoding [15, 23, 20], to which we will default from now on.

The following definitions that are parametric over types X, Y have the implicit requirement that these types are L-encodable. Note that whenever using a notion from this section, the precise encoding used is an important part of the statement, e.g. the addition of binary numbers can be performed more efficiently than the addition on a unary encoding.

As the time measure, we use the number of β-reduction steps to a normal form, which has been shown to be a reasonable computational model [5]. Forster and Kunze [10] have mechanised a notion of L-computability with running time for meta-level functions that can be encoded as L-terms. We refer to this work, and use this notion slightly informally here.

▶ **Definition 1** (Polynomial Boundedness). *We say that a function $f : \mathbb{N} \to \mathbb{N}$ is polynomial if there exists $k : \mathbb{N}$ such that for sufficiently large n, we have $fn \leq k \cdot n^k$. We say that a function $g : \mathbb{N} \to \mathbb{N}$ is polynomially bounded if there exists a monotonic, polynomial function f with $\forall x.g(x) \leq f(x)$.*

The monotonicity requirement in the definition eases formal reasoning by allowing rewriting in the function argument.

Definition 2 (Polynomial-time Computability). *A function $f : X \to Y$ is polynomial-time computable, if there exists $t : \mathbb{N} \to \mathbb{N}$ such that*

- *f is L-computable in time $\lambda x.t\ \|\ulcorner x \urcorner\|$,*
- *t is polynomially bounded, and*
- *the result size is polynomially bounded by the input size, i.e. there is a polynomial p with $\|\ulcorner f\ x \urcorner\| \leq p\|\ulcorner x \urcorner\|$.*

We use the term size of the (Scott) encoding, $\|\ulcorner \cdot \urcorner\|$, as the input size of computations. Note that the last requirement of Definition 2 is non-standard: For TMs, this condition holds automatically due to the known "time-bounds-space" lemma of TMs, while for L, a computation can produce terms of a size exponential in the number of reduction steps [11].

Definition 3 (P). *A decision problem $p : X \to \mathbb{P}$ can be decided in polynomial time, written $p \in \mathsf{P}$, if there exists $f : X \to \mathbb{B}$ such that f is polynomial-time computable and f decides p, i.e. $p\ x \leftrightarrow f\ x = \mathsf{true}$ for all x.*

We define NP using the well-known characterisation via deterministic certificate verifiers [25, p. 181], instead of e.g. extending L to support non-deterministic computations.

Definition 4 (NP). *A decision problem can be verified in polynomial time, written $p \in \mathsf{NP}$, if there exists a verifier relation $R : X \to \mathsf{Ter} \to \mathbb{P}$ and a certificate size bound $f : \mathbb{N} \to \mathbb{N}$ such that*

1. *$\lambda(x, y).R\ x\ y$ is polynomial-time decidable (we call this decider* verifier*),*
2. *$R\ x\ y$ implies $p\ x$,*
3. *$p\ x$ implies $R\ x\ y$ for some y with $\|\ulcorner y \urcorner\| \leq fx$, and*
4. *f is polynomially bounded.*

Note that this definition (unlike [25]) does not require the verifier to reject certificates which are too large, i.e. exceed the polynomial bound. This simplifies the verifier relation and implementation in formalised NP-containment proofs, as properties on the logical level of R are less tedious than implementing more tests in the function. Moreover, this definition only allows L-terms as certificates. This simplifies the formalisation in places where NP-containment is an assumption (e.g. when proving NP-hardness), as otherwise one would need to spell out requirements on the certificate type, like efficient enumerability, of the definition of NP.

In the other direction, when establishing NP-containment, a generalisation allows any type Y for certificates, as long as its encoding is injective and decodable in polynomial time.

Finally, the notion of polynomial-time reducibility enables us to define the two crucial notions relating problems to NP.

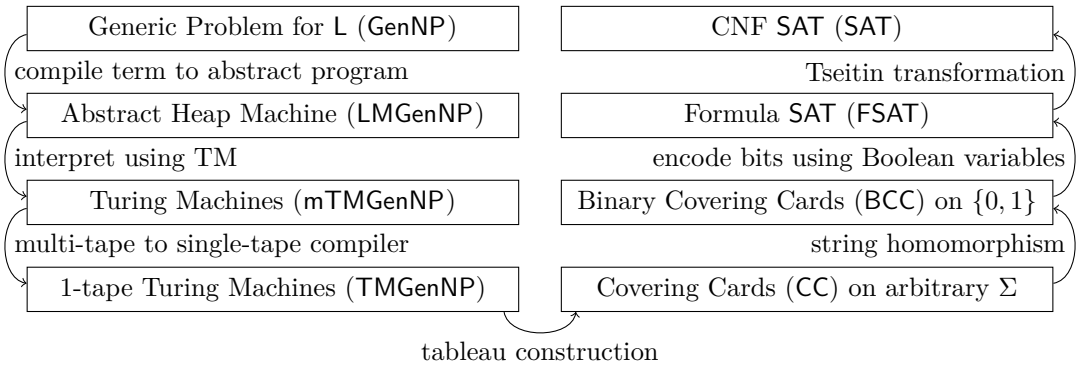

Figure 1 Chain of reductions from GenNP to SAT.

Definition 5 (Polynomial-time Many-one Reductions). *A problem $p : X \to \mathbb{P}$ reduces to $q : Y \to \mathbb{P}$ in polynomial-time, written $p \preceq_{\mathfrak{p}} q$, if there exists a polynomial-time computable function $f : X \to Y$ such that $p\ x \leftrightarrow q(fx)$.*

Definition 6 (Hardness and Completeness). *A problem $p : X \to \mathbb{P}$ is* NP*-hard if for all problems $q \in$ NP, $q \preceq_{\mathfrak{p}} p$. If additionally $p \in$ NP, then p is* NP*-complete.*

The standard closure properties hold, enabling reductions as a useful tool:

▶ **Lemma 7.**

1. *If $p \in$ NP and $q \preceq_{\mathfrak{p}} p$, then $q \in$ NP.*
2. *If p is* NP*-hard and $p \preceq_{\mathfrak{p}} q$, then q is* NP*-hard.*
3. *$\preceq_{\mathfrak{p}}$ is a pre-order.*

4 Overview of the Reduction

The hard part of proving the Cook-Levin theorem is the NP-hardness proof of SAT, on which we focus in this paper. For this, we construct a polynomial-time many-one reduction from a generic problem, which can be easily shown NP-hard, to SAT. Our reduction factorises in several intermediate problems, of which we give a brief overview in this section. For the most interesting parts, we give more technical details in Sections 5 and 6.

It turns out that for different models of computation, so-called *generic* NP*-hard problems* are of interest in this reduction chain: They ask the question whether, for the input triple $(\mathcal{P}, k, t) : \mathsf{Prog} \times \mathbb{N} \times \mathbb{N}$, there is a certificate c such that the program $\mathcal{P}$ halts on input c in no more than t time steps and such that k bounds the size of c. The exact notions of programs, inputs, size, and time differ per concrete model of computation. The generic problem for L can be shown NP-hard quite easily, as its definition lines up nicely with NP.

Our full reduction chain can be divided into two segments (Figure 1): First we reduce the generic NP-hard problem for L to a generic NP-hard problem for (single-tape) TMs. In a second step, single-tape TMs are encoded as Boolean formulas. While this second step closely follows the idea of a common textbook construction [26], our proof neatly separates different aspects of the construction into different reductions. One important detail is that L is used to *implement* all the reduction functions, as mentioned earlier. The reason for still having TMs as an intermediate problem is that their computational structure is much more local than that of L, simplifying the encoding as a formula.

Generic Problem for L

The generic problem for L, GenNP, asks, for the input triple (s, k, t), whether there exists a list of Booleans bs, the certificate, such that (1) the term s applied to the encoding $\ulcorner bs \urcorner$ halts in no more than t steps and (2) the encoding of bs is smaller than k. The definition of NP allows to easily establish NP-hardness of this problem, see Section 5.

Instead of directly reducing the generic problem for L to TMs, we employ an *abstract heap machine* as an intermediate step: It is a stack machine for L that has built-in structure sharing due to the use of closures and an explicitly modelled heap, similar to the one used by Forster et al. [11]. The reduction from GenNP to the generic problem for this abstract machine, LMGenNP, uses the fact that the abstract machine implements L with a linear factor overhead.

Reducing to Turing Machines

The next step is to encode the heap machine using a multi-tape TM, for which we use the interpreter by Forster et al. [12]. The TM (i.e. state space and transition function) produced by the translation is fixed, i.e. this is an interpreter, not a compiler.

The most difficult part here is that the generic problem for multi-tape TMs, mTMGenNP, uses an arbitrary tape as certificates, while the L-problems use a list of Booleans. Therefore, we need to define and verify a TM that performs some preprocessing: Each possible TM-encoding of an L-encoding of a list of Booleans must be the result of this TM on some tape, and this TM never produces something that does not encode a Boolean list.

We now use an existing translation from multi-tape to single-tape TMs [13] to reduce from mTMGenNP to the generic problem for single-tape TMs, TMGenNP: Instances are dependent tuples $(\Sigma, M : \mathsf{TM}, in : \Sigma^*, k : \mathbb{N}, t : \mathbb{N})$, where Σ is a finite tape alphabet and M is a (two-sided) single-tape machine over Σ. An instance is a yes-instance if there exists a certificate $cert : \Sigma^*$ with $|cert| \leq k$ such that M halts on $in +\!\!+ cert$ in $\leq t$ steps.

Reducing to SAT using Covering Cards

For the reduction from TMGenNP to a Boolean formula, we follow the textbook proof by Sipser [26] at a high level. The key idea is to make use of the well-known "time-bounds-space" fact for TMs. From the fixed input part in, the maximum certificate size k, and the time bound t we can determine a maximum amount of space s the TM may use. All of the machine's computation can be layed out in a tableau of $t + 1$ lines and width s, with each line representing one configuration of the machine and each cell containing an encoding of a tape symbol or the state, see Figure 2. We need a special blank symbol $\sqcup$ filling the space not currently in use by the TM. The state symbol, initially $q_0^{\sqcup}$, not only contains the state q_0 but also marks the head's position and contains the current symbol under the head. Eventually, this tableau determines the layout of the Boolean formula.

Exploiting the local computational structure of TMs, we constrain succeeding lines of the tableau locally with *covering cards* of 3 by 2 symbols to enforce valid configuration changes. At each possible offset of two succeeding lines, a card matching the symbols at that position needs to exist. Importantly, the cards need to overlap, which allows to express global constraints using only local cards. The set of cards available encodes the valid transitions the TM can take. Cards *can* be re-used.

Instead of directly encoding this tableau as a formula, as is done by Sipser, we introduce a string-based intermediate problem called *Covering Cards* (🐓 CC) making the tableau of configurations and covering cards explicit. This allows us to separately deal with the

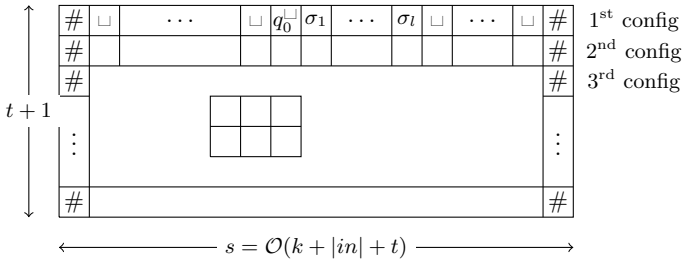

Figure 2 The tableau of configurations. Each line is delimited by #.

construction of the tableau and the TM encoding, reducing the special symbols introduced by the tableau encoding to a binary alphabet, and finally constructing a Boolean formula.

CC may use arbitrary finite alphabets for the contents of the tableau. This expressivity is needed for the reduction even if the original TM only uses a binary alphabet, as the tableau construction needs "control symbols". As a first step towards encoding this as a Boolean formula, we reduce to a binary alphabet $\{0, 1\}$ by replacing every element σ of the alphabet Σ by a unique binary string of length $|\Sigma|$. We call this variant *Binary Covering Cards* (🐓 BCC).

The next step of the reduction are general Boolean formulas φ🐓 featuring conjunction, disjunction, and negation. The general formula satisfiability problem (🐓 FSAT) has as instances formulas φ and asks whether there exists an assignment a such that φ is true under this assignment. BCC can be encoded as a Boolean formula $\varphi_{init} \wedge \varphi_{cards} \wedge \varphi_{final}$ of three gadgets for the first line of the tableau, the covering cards, and the final substring constraint. For these formulas, each cell of the tableau can be represented as one Boolean variable due to the binary alphabet.

However, usual statements of the Cook-Levin theorem reduce to the satisfiability problem of conjunctive normal forms (🐓 SAT). The formulas produced by the reduction to FSAT are not yet in CNF and so we transform formulas φ into their normal form. Since the naive way of implementing this transformation can incur an exponential blowup in the formula size, we use the well-known Tseitin transformation [28] to obtain a polynomial-time reduction.

5 From L to Turing Machines

In this section, we give some of the details of showing the generic problem for Turing machines TMGenNP NP-hard. We first show the problem GenNP to be NP-hard. We then give a path of polynomial-time reductions to TMGenNP.

The generic problem for L, GenNP, asks whether, for a triple consisting of a L-term s, and two unarily encoded natural numbers k (the size bound for certificates) and t (the time bound), there exists a list of Booleans bs with size no greater than k such that s, on the encoding of bs, halts in no more than k steps[2].

Instead of allowing all triples as possible input, we restrict the definition to a subset of *admissible* triples. Formally, the type of GenNP is a subtype, i.e. $\{x : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N} \mid \mathrm{adm}\ x\} \to \mathbb{P}$ for some restriction $\mathrm{adm} : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N} \to \mathbb{P}$. The reason for this is that later, when reducing from multi to single-tape TMs, this restriction simplifies the proof.

[2] The mechanisation generalizes certificates to any type X that is polynomial-time surjectable to Ter🐓.

Definition 8 (Generic NP-complete problem for L). *A triple* $(s, k, t) : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N}$ *is a yes-instance for* GenNP *if there exists a certificate* $bs : \mathbb{B}^*$ *with* $\|\ulcorner bs \urcorner\| \leq k$ *such that* $s\ \ulcorner c \urcorner$ *halts in no more than* t *steps.*

For GenNP*, the subtype of admissible input triples is restricted to closed abstractions* s *satisfying two constraints: If* s *halts when applied to some certificate, then there is a certificate of size* $\leq k$ *on which* s *halts. If* s *halts on a certificate, then it halts in no more than* t *steps.*

Theorem 9. GenNP *is* NP*-hard*

Proof. Assume a problem $p : Y \to \mathbb{P}$ is in NP with verifier $v : Y \times \mathsf{Ter} \to \mathbb{B}$, and assume that f is a polynomial-time surjection from $\mathbb{B}^*$ to Ter.

The reduction function to GenNP now maps $y : Y$ to a triple: The first component is the term $s := \lambda x.\mathrm{HaltOnTrue}(s'x)$, where s' is the term computing (in polynomial time) the function $\lambda x.v(y, fx)$, and HaltOnTrue is a combinator that, applied to an encoded Boolean $\ulcorner b \urcorner$, halts only on true.

The second component, the size bound, is chosen large enough so that there is a smaller certificate x iff the verifier v accepts some fx as certificate for y. The third component, the time bound, can be chosen large enough by determining the running time function of s', and then plugging in k and the size of y.

We skip over the actual correctness proof, which does not only have to account for two directions, but also the side conditions. We also skip over the polynomial-time computability proof in this presentation. ◀

We now reduce from GenNP to $\mathsf{mTMGenNP}_M$, a generic problem on multi-tape TMs [3]. Syntactically, for a fixed $n+1$ tape TM M, the inputs to $\mathsf{mTMGenNP}_M$ are triples consisting of a n-vector v of tapes, and two unarily encoded natural numbers k (the size bound for certificates) and t (the time bound). As for GenNP, only a subtype of those are admissible.

Definition 10 ($\mathsf{mTMGenNP}_M$). *A triple* $(v, k, t) : \mathrm{tape}^n \times \mathbb{N} \times \mathbb{N}$ *is a yes-instance for* $\mathsf{mTMGenNP}_M$ *if there exists a tape* t_c *with* $|t_c| \leq k$ *such that* M *on input* $t_c :: v$ *halts in no more than* t *steps.*

For $\mathsf{mTMGenNP}_M$*, the subtype of admissible input triples is restricted by two constraints: (1) If* M *on* $t_c :: v$ *halts when applied to some certificate* v*, then there is a certificate shorter than* k *on which it halts. (2) If* M *halts on* $t_c :: v$*, then it halts in no more than* t *steps.*

Compared to the definition of GenNP, we separate the program into two parts: A fixed $n + 1$ tape TM M and, as part of the actual input, all-but-the-first tapes v. The reduction now needs two TMs as key ingredients: A TM CheckCert that transforms the guessed first tape, containing arbitrary symbols, into all possible encodings of Boolean lists, and a polynomial-time L-evaluator Eval_L, mechanised by Forster et al. [12].

Theorem 11. GenNP *reduces to* $\mathsf{mTMGenNP}_M$ *in polynomial time for some* M*.*

Proof. For a triple $(s, k, t) : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N}$, the reduction function produces a triple $(v, k', t') : \mathrm{tape}^n \times \mathbb{N} \times \mathbb{N}$ such that v contains an encoding of s. Then, k' and t' are chosen large enough such that the M we now construct satisfies all side conditions, which is tedious to mechanise and would be even more tedious to describe here.

[3] The mechanisation uses an abstract machine for L as intermediate step, which in hindsight is not the optimal factorisation, as nothing of interest happens in the first reduction. The better factorisation is to explicitly separate out a TM evaluating L. Since this focuses on the more interesting parts, we explain that approach here.

The machine M' is constructed as follows: It tests if the first tape contains a Boolean list and runs the simulator $\mathsf{Eval}_{\mathsf{L}}$ on the application of s (read from v) to the encoding of the Boolean list. Otherwise, it diverges. ◀

The last step now is a reduction to the generic problem TMGenNP for single-tape TMs, for which we use the multi-to-single-tape translation by Forster et al. [13].

Definition 12 (TMGenNP). *A tuple* $(\Sigma, M : \mathsf{TM}, in : \Sigma^*, k : \mathbb{N}, t : \mathbb{N})$ *is a yes-instance for* TMGenNP *if there exists a tape* t_c *with* $|t_c| \leq k$ *such that* M *halts on* $in +\!\!+ t_c$ *in* $\leq t$ *steps.*

Note that the used tape is a combination of in from the input and the 'guessed' certificate.

Theorem 13. *For every* M, $\mathsf{mTMGenNP}_M$ *reduces to* TMGenNP *in polynomial time.*

Proof. Given a $n+1$ tape machine M and an instance (v, k, t) of $\mathsf{mTMGenNP}_M$, we construct the instance (Σ, M', in, k', t') for TMGenNP as follows:

M' is a single tape machine that basically is the single-tape compilation of M, but we prepend an auxiliary machine that checks that the tape is the encoding of some $n+1$-vector of tapes, according to the single-to-multi-tape construction we use. M' can be hard-coded and does not have to be computed in L, as M is a parameter[4]. in is the proper encoding of v, for use by M'. k' is k plus some small constant due to overhead in the encoding of tapes. t' is chosen so that M' terminates, according to the running time analysis, and polynomial in k, t and v.

For the correctness of the reduction, the two restrictions for admissible inputs to $\mathsf{mTMGenNP}_M$ are crucial for the direction where one obtains a multi-tape certificate from a terminating run of M': The construction we use only gives an upper bound for the running time of M' in terms of the running time of M. Therefore, we cannot be sure that only because M' halts in no more than k' steps, the multi-tape machine halts in k steps. Using the side conditions, just by the fact that the multi-to-single tape compiler preserves the halting behaviour, we get the existence of a certificate with the right bounds. ◀

We omit more details[5] and only note that the actual running time analysis of M', and the poly-time computability proofs of the reduction function are at least an order of magnitude larger than the parts sketched in this proof.

6 Reducing TMGenNP to SAT

In this section, we focus on the remaining steps to show SAT to be NP-complete, starting from the NP-hardness of TMGenNP.

6.1 Encoding of Turing Machines using Covering Cards

First, we present a more detailed account of encoding TMs using Covering Cards. The key steps of this proof are to find an encoding of configurations which is essentially unique and to construct the covering cards for the TM simulation, *such that* there is a one-to-one correspondence between valid CC-steps and TM steps, and to construct a *prelude* which, before the TM simulation runs, 'guesses' a certificate string in a single CC-step.

[4] A weaker definition of TMGenNP that is parameterised over Σ and M would suffice for our purposes, but our construction in Section 6 works for the actual, more general definition.

[5] For example that for technical reasons, we have another version of TMGenNP as intermediate step.

Definition of CC

First, we specialise CC to a specific case we call 3-CC. 3-CC instances are dependent tuples (Γ, in, c, fin, t), where Γ is the finite alphabet for the cells of the tableau, $in : \Gamma^*$ is the first line of the tableau, and $t + 1$ is the number of lines. $C : {C_\Gamma}^*$ is a list of covering cards consisting of a premise and a conclusion, where $C_\Gamma := \Gamma^3 \times \Gamma^3$. Finally, $fin : \Gamma^*$ is a constraint on the tableau's final line s_{t+1} used for encoding that the TM has halted. An instance is a Yes-instance if there exists a sequence of strings $s_0 = in, \ldots, s_{t+1}$ such that $s \in s_{t+1}$ for an $s \in fin$ and $s_i \rightsquigarrow_c s_{i+1}$, denoting that every succeeding line follows validly from the previous one according to the cards C. Formally, $s \rightsquigarrow_C t := \forall i \in [0, w-3].\exists a \in c.s[i, i+2] = \pi_1 a \wedge t[i, i+2] = \pi_2 a$. We call this a CC-step from s to t.

The generalisation of 3-CC to CC (needed for the reduction to a binary alphabet) makes the following changes: the width of the cards, originally 3, becomes a variable ω, the offset at which cards need to hold, originally 1, becomes a variable o, and the list of final subsymbols is generalised to a list of final substrings. Abstractly, o symbols grouped together form a unit and covering cards need to hold only at positions which are multiples of o. Every 3-CC instance is a CC instance by an easy transformation.

Encoding Configurations

For the rest of this section, we fix a TMGenNP instance (Σ, M, in, k', t). The maximum length the fixed input in and the certificate can have is bounded by $k := |in| + k'$. We use the metavariables $\sigma : \Sigma$ and $m : \Sigma + \{\sqcup\}$. First, we consider how to *deterministically* simulate the machine on a given initial input and later deal with 'guessing' a certificate.

A configuration consists of the current TM state, the tape contents, and the position of the single head. A configuration string $s : \Gamma^*$ over an extended alphabet Γ has a fixed width and encodes a configuration in an essentially unique way. Our encoding features the current state q and the symbol under the head m combined in a single symbol q^m *at the exact center* of the string. The tape halves $u : \Sigma^*$ to the left and to the right of the head are represented by the substrings $h : \Gamma^*$ left and right of the center symbol. Fixing the state symbol to the center simplifies the inductive invariants in the presence of two-sided infinite tape machines, but necessitates that, when the head is moved, the whole tape representation in the string is shifted.

Figure 3 shows an example of a configuration change, where the head is moved to the left and thus the tape needs to be shifted to the right. The outer-most character on both sides is the delimiter #. We denote the amount of space maximally needed by the TM by $z := k + t$[6]. The tape symbols of the successor string are annotated with polarities p. They denote the direction in which the tape is shifted (left $\overleftarrow{m}$, stay $\overline{m}$, and right $\overrightarrow{m}$) and are added to every tape symbol. Polarities are needed in order to enforce a consistent tape shift across the whole tape string. A symbol with unknown polarity p is written as m^p. We omit the polarity of a symbol when it is irrelevant.

We define two relations $\sim_t$ and $\sim_c$ to capture the representation of tape halves and configurations. The relation $\sim_t$ is parameterised by the amount of space n available for the TM simulation and the polarity p. The empty tape half of length n is represented by $E\ p\ n$, defined as $E\ p\ 0 := [\#]$ and $E\ p\ Sn := \sqcup^p :: E\ p\ n$. A general tape half u is represented by u with an arbitrary polarity p added and trailing blank space, denoted $u \sim_t^{(n,p)} h$ (with two

[6] but we add two further blanks to each side as having at least three symbols in each tape string simplifies the proofs.

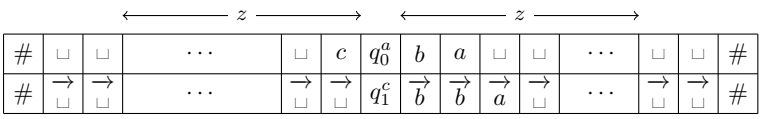

Figure 3 Two successive configuration strings, where $\Sigma = \{a, b, c\}$, q_0 is the initial state, and $\delta(q_0, \lfloor a \rfloor) = (q_1, L, \lfloor b \rfloor)$. The strings are delimited by #.

blanks added as noted above):

$$u \sim_t^{(n,p)} h := |u| \leq n \wedge h = [\ x^p \mid x \in u\ ] +\!\!+ E\ p\ (2 + n - |u|).$$

A configuration string is pieced together from three parts, where the left tape string must be reversed. The polarity is irrelevant, as long as it is consistent:

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

This relation determines the string representing a configuration uniquely, up to the polarity.

Simulation Cards

As a reminder, the covering cards used for the simulation encode valid configuration changes through overlaps. We distinguish three types of covering cards that encode valid configuration changes: cards that are used for shifting the tape, cards that affect the center state symbol, and cards that replicate a configuration string in case the machine halts in less than t steps. As most cards need to be applicable independently of the current tape contents or the polarity, we give *rules* for creating cards, containing metavariables σ_i and m_i. Moreover, for their premises we do not give the polarities. In order to obtain the actual cards, the rules are instantiated with all possible assignments to the metavariables. For instance, for transitions, three kinds of rules are needed, having the center state symbol in the left, middle, or right position. The middle position rules for $\delta(q_1, \lfloor \sigma_1 \rfloor) = (q_2, \lfloor \sigma_2 \rfloor, \mathsf{L})$ are:

$$\begin{array}{|c|c|c|}\hline \sigma_3 & q_1^{\sigma_1} & m_1 \\ \hline \overrightarrow{m_2} & q_2^{\sigma_2} & \overrightarrow{\sigma_2} \\ \hline \end{array} \qquad \begin{array}{|c|c|c|}\hline \sqcup & q_1^{\sigma_1} & m_1 \\ \hline \rightarrow & q_2^{\sqcup} & \overrightarrow{\sigma_1} \\ \sqcup & & \\ \hline \end{array}$$

One can see that a large number of rules is needed. The full set of rules can be found in the Coq development. We denote the set of covering cards obtained by instantiating the rules by R_{sim}.

The next theorem is the main result needed to prove that TM runs are in one-to-one correspondence with CC-step sequences.

Theorem 14 (Step Simulation)**.**
If $(q, tp) \sim_c s$, $(q, tp) \succ (q', tp')$ and $|tp| < z$, then there exists a unique s' such that $s \rightsquigarrow_{R_{sim}} s'$. Moreover, s' satisfies $(q', tp') \sim_c s'$.

Proof. 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 TM takes. By analysing the heads h_l and h_r of the tape halves, we can find out the cards to use at the center. These cards then uniquely determine the heads h_l' and h_r' of the successor tape halves. By an induction, the successor tape halves are therefore fully determined. The coverings can then be justified separately. ◀

A similar correspondence can be proved for halting configurations. As consequences, we get soundness and completeness for the simulation of deterministic TMs. Both proofs are by induction with some intermediate lemmas. The predicate haltingString s denotes that s contains a symbol q^m and halt q = true.

Theorem 15 (Completeness). *If* $|tp| \leq k$, $(q, tp) \sim_c s$, *and* $(q, tp) \rhd^{\leq t} (q', tp')$, *then there is* s' *with* $s \rightsquigarrow^t s'$, $(q', tp') \sim_c s'$ *and* haltingString s'.

Theorem 16 (Soundness). *If* $(q, tp) \sim_c s$, $|tp| \leq k$, $s \rightsquigarrow^t s'$, *and* haltingString s', *then there are* q', tp' *with* $(q', tp') \sim_c s'$, $(q, tp) \rhd^{\leq t} (q', tp')$ *and* $|tp'| \leq z$.

Nondeterministic Certificate

Finally, we need to generate an initial string that accounts for the certificate input of the TM. Since the 3-CC instance should be satisfiable iff there exists a valid certificate, this needs to be encoded using a CC-step. We prefix another line to the tableau and add additional cards that allow for a non-deterministic successor string that is used as the initial string of the TM simulation. By making these cards use a disjoint alphabet, they cannot interfere with the succeeding TM simulation.

Theorem 17. TMGenNP *reduces to* CC *in polynomial time.*

6.2 From 3-CC to SAT

For lack of space and since the constructions are conceptually simple, we do not give a more detailed account of the remaining steps from 3-CC to SAT but instead refer to Section 4 and our mechanisation. In total, we show the following reductions:

▶ **Theorem 18.**

1. CC *reduces to* BCC *in polynomial-time.*
2. BCC *reduces to* FSAT *in polynomial time.*
3. FSAT *reduces to* SAT *in polynomial time.*

6.3 NP Containment

For proving SAT NP-complete, we also need to show that SAT $\in$ NP, which is far simpler than the chain of reductions. We construct a standard certificate verifier that takes assignments as certificates and verify it to run in polynomial-time. Further details are left to the Coq formalisation.

Lemma 19 (SAT is in NP). SAT $\in$ NP

In total, we now have proved the Cook-Levin theorem:

Theorem 20 (Cook-Levin). SAT *is* NP*-complete*[7].

[7] Actually, we mechanise that even 3-SAT, where clauses are restricted to have size 3, is NP-complete.

Table 1 Line counts (grouped, counted with `coqwc`)

Component		Spec	Proof
Libraries definitions and frameworks for L and TM, ...		17000+	17000+
Complexity Definitions		317	675
Problem Definitions	Normal Versions	444	541
	(Add.) Flat Versions	325	674
GenNP is NP-hard		43	230
GenNP to mTMGenNP$_{\text{excl. Eval}_{\mathsf{L}}}$		203	574
mTMGenNP to TMGenNP$_{\text{excl. multi-to-single-tape compiler}}$		264	889
TMGenNP to CC		3470	4442
CC to SAT		755	2312
SAT $\in$ NP		108	297

7 Mechanisation in Coq

Our development compiles with Coq 8.12.1, without any axioms and with a code size as denoted in Table 1. We comment on a few challenges involved in the mechanisation.

Flat First-Order Encodings One of the limitations of the extraction framework of [10] is that higher-order types and types having propositional components cannot be extracted directly. For instance, arbitrary finite types, dependent function types, and dependent pairs are hard to handle generically. We call a particular formulation of a problem $P : X \to \mathbb{P}$ *flat* if X is L-encodable and will also refer to the type of instances X as *flat*.

Directly formulating all problems discussed in this paper such that their instances are flat is unpleasant, as that strips away many of the amenities of Coq's dependently-typed language which are useful for stating problems. This does in particular affect the generic problems for TMs and the different variants of CC. Instead, we first formulate a problem P without paying attention to flatness and use this nice formulation to prove correctness statements. For the extraction, a separate flat version P' is defined, for instance by representing a finite type Σ by the number of its elements $|\Sigma|$ and its elements by the natural numbers up to $|\Sigma|$. The functions computing the reduction are defined on the flat versions. To prove their correctness, a natural notion of agreement to the non-flat definitions is proved.

For the variants of CC, this is straightforward. A flat definition of TMs was already available in the Coq Library of Undecidable Problems [14].

Encoding Turing machines in CC A major difficulty in the reduction to CC is the handling of the covering cards. A large number of rules are needed to generate the cards and to handle all cases. This is in significant parts due to the fact that blank symbols $\sqcup$ are unknown to the TMs and have to be handled separately. Theorem 14 splits up into a large number of cases (100 are non-contradictory) due to cases analyses on the machine's behaviour and the local shape of the tape around the head (for selecting the right cards at the center). This is only feasible using custom Ltac automation (around 160LOC for this theorem). Further custom automation is used to invert the relations $\sim_t, \sim_c$ and to eliminate contradictions.

For easing automation, the cards are not directly formalised using lists, but instead using indexed inductive predicates. This enables an easy use of `eauto` to justify coverings and inversions to analyse coverings.

To make an automatic extraction into L and a resource analysis possible *with the extraction framework* [10], separate flat versions FlatCC of CC and FlatTMGenNP of TMGenNP are used,

representing the finite type for the alphabet by natural numbers. In particular, a list of *flat* cards needs to be computed. In order to transfer the proof of correctness from CC to FlatCC, functions for computing the cards are parameterised and can be instantiated either with a finite type or with natural numbers. The proof then first shows that the cards determined by the indexed inductive predicates and the list-based cards using finite types agree and in a second step that both instantiations of the list-based cards are suitably equivalent.

Prop vs Type Definitions like polynomial-time computability contain several components that are explicitly used in the constructions that we perform, like the explicit running time or the function bounding the result size. Therefore, our proofs get more concise by modelling definitions not as propositions, but in Type, which allows to define the projections.

7.1 Resource Analyses

A major part of complexity theory is making sure that the constructions satisfy the required resource bounds. While this aspect is uninteresting from a formalisation perspective, the mechanisation of resource bounds is work-intensive and benefits from the right abstractions. We need to derive not only running time bounds for L computations, but also for their result size, and similar bounds for the TMs constructed in Section 5.

We define a preorder $\leq_c$ on functions, used to give more readable upper bounds.

$$f \leq_c g := \Sigma c.\forall x.f\ x \leq c \cdot g\ x$$

This definition has many of the desired properties of O notation, like abstraction from constant factors and the ability to bound a (finite) sum by the largest summands. But in contrast to 'full' O notation, which was mechanised and used by Armaël et al. [18], our definition is lightweight and covers functions with multiple (uncurried) arguments of arbitrary type, without the need of a notion of limits or ultrafilters.

We use two notational tricks in Coq: Using coercions and a record we can concicely express the existence of some function $\leq_c g$ in specifications. Using the projection to the c in the definition of $\leq_c$, we can employ the concrete bound $c \cdot g\ x$ whenever needed and still hide the exact value of c.

Intuitive Time Bounds for L

Since L is a low-level machine model, we inevitably need to connect our analyses to the notions of time for L. Formally, the running time function simply is a function that computes the number of steps needed for each input. A priori, the syntactic representation of this function can be very messy (e.g. recursive), mention many constants whose exact value is of no interest, and is not expressed in terms of the size of the input, but the input itself. Deriving concise, intuitive bounds is an important aspect of our mechanisation. As an example, we consider deriving time bounds for the evaluation of Boolean clauses (conjunctions of literals) used for the SAT verifier of Section 6.3. We represent variables by natural numbers, literals as pairs of variables and their Boolean sign, and clauses as lists of literals. Assignments are lists containing the variables which are assigned true, all other variables are implicitly false.

$$v : \mathsf{var} := \mathbb{N} \qquad l : \mathsf{lit} := \mathbb{B} \times \mathsf{var} \qquad C : \mathsf{cla} := \mathsf{lit}^* \qquad a : \mathsf{assgn} := \mathsf{var}^*$$

$$\begin{aligned} \mathsf{existsb}\ p\ [\,] &:= \mathsf{false} & \mathcal{E}_{\mathsf{lit}}\ a\ (s, v) &:= (v \overset{?}{\in} a) \overset{?}{=} s \\ \mathsf{existsb}\ p\ (x :: l) &:= p(x) \mid \mathsf{existsb}\ p\ l & \mathcal{E}_{\mathsf{cla}}\ a\ C &:= \mathsf{existsb}\ (\mathcal{E}_{\mathsf{lit}}\ a)\ C \end{aligned}$$

Let us assume that we have already analysed $\mathcal{E}_{\mathsf{lit}}$. For the time analysis of $\mathcal{E}_{\mathsf{cla}}$, we first analyse the higher-order function existsb used in its definition. We start with the (simplified) recurrences generated by the certifying extraction framework of Forster et al. [10], which are phrased in terms of the number of β-steps of the extracted L-term:

$$8 \leq T_{\mathsf{existsb}}(T_f, [])$$
$$T_f\ h + T_{\mathsf{existsb}}(T_f, l) + 15 \leq T_{\mathsf{existsb}}(T_f, h :: l)$$

where C is an L-encodable type and $T_f : X \to \mathbb{N}$ is the running time function of the argument function $f : X \to \mathbb{B}$. Instead of solving this recurrence explicitly for T_{existsb}, involving all the constants, we strive for more intuitive bounds in terms of abstract size functions, such as the length of the list. Therefore, we give a bounding solution $T_{\mathsf{existsb}} \leq_c B_{\mathsf{existsb}}$ for $B_{\mathsf{existsb}} := \lambda(T_f, C).\sum_{a \in C} T_f a + |C| + 1$ which hides the constants.

With this, we derive bounds for $\mathcal{E}_{\mathsf{cla}}$. We make use of the abstract size function maxVar giving the (encoding size of the) maximum variable used in a clause. Assuming that $T_{\mathcal{E}_{\mathsf{lit}}} \leq_c B_{\mathcal{E}_{\mathsf{lit}}} := \lambda a.(|a| + 1) \cdot (\mathsf{maxVar}(a) + 1)$, we compositionally obtain the bound $T_{\mathcal{E}_{\mathsf{cla}}} \leq_c B_{\mathcal{E}_{\mathsf{cla}}} := \lambda(a, C).(|C| + 1) \cdot (|a| + 1) \cdot (\mathsf{maxVar}(a) + 1)$ by instantiating the bound for existsb.

To get back to the concrete resource measures of L in the end and obtain bounds in terms of the encoding size, we simply have to prove that the abstract size functions are bounded in terms of the encoding size, e.g. that $|a| \leq \|\ulcorner a \urcorner\|$.

8 Discussion

To the best of our knowledge, this paper contains the first mechanisation of a complexity-theoretic result all the way down to a reasonable computational model. Our experiences seem to support the conjecture [13] that a λ-calculus is more amenable to mechanising complexity theory than TMs are. This is in large parts due to being able to automatically generate L-code from Coq-code, which makes programming and verifying in L simple, and having recurrences generated in the same process. Still, the resource analysis poses a major overhead over just verifying the functional correctness of reductions as is done in synthetic computability theory [9], since we need to derive flat first-order encodings of the problems and of course analyse the running time of the reduction functions.

It may seem ironic that we started out with a quest to move away from TMs, but ended up still using them as an intermediate problem in the reduction chain because of their local computational structure, and because of our experience with mechanising TMs [13, 14]. But the techniques we have for poly-time computability proofs in L are vastly superior to the hypothetical alternative of using TMs as underlying model of computation and veryfing TM-computability of all constructions.

Future work This work, the NP-hardness of SAT, now enables a variety of fully mechanised NP-completeness proofs, like the well known 21 NP-complete problems by Karp [21], without even knowing what a Turing machine is.

We conjecture that many interesting results in computational complexity theory can be mechanised when phrased in terms of L, as evident by a mechanisation of the time hierarchy theorem we are working on. We want to explore these possibilities in more detail.

Mechanising the agreement between our definition of NP and a TM-definition of NP should be straightforward by reusing simulations in L/TMs we already have.

We are interested in understanding how characterisations of P and NP that are independent from a computational model, like via Fagin's theorem [8] or certain type systems [19], can be shown equivalent to our characterisations.

References

1 Andrea Asperti. A formal proof of borodin-trakhtenbrot's gap theorem. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 163–177, Cham, 2013. Springer International Publishing.

2 Andrea Asperti. Reverse complexity. *Journal of Automated Reasoning*, 55:373–388, 2015.

3 Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, 2015.

4 Andrej Bauer. First steps in synthetic computability theory. *Electron. Notes Theor. Comput. Sci.*, 155:5–31, May 2006. doi:10.1016/j.entcs.2005.11.049.

5 Guy E. Blelloch and John Greiner. Parallelism in Sequential Functional Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 226–237, 1995. doi:10.1145/224164.224210.

6 Mario M. Carneiro. Formalizing computability theory via partial recursive functions. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.12.

7 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. doi:10.1145/800157.805047.

8 Ronald Fagin. Generalized first-order spectra, and polynomial. time recognizable sets. *SIAM-AMS Proc.*, 7, 01 1974.

9 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, Jan 2019. ACM.

10 Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

11 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, January 2020. doi:10.1145/3371095.

12 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. A mechanised proof of the time invariance thesis for the weak call-by-value λ-calculus. under review.

13 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, 2020. ACM.

14 Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. URL: https://github.com/uds-psl/coq-library-undecidability.

15 Yannick Forster and Gert Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *International Conference on Interactive Theorem Proving*, pages 189–206. Springer, 2017.

16 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.

17 Ruben Gamboa and John Cowles. A mechanical proof of the Cook-Levin theorem. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics*, pages 99–116, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

18 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 533–560, Cham, 2018. Springer International Publishing.

19 Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, page 464, USA, 1999. IEEE Computer Society.

20 Jan Martin Jansen. *Programming in the λ-Calculus: From Church to Scott and Back*, pages 168–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40355-2_12.

21 Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. doi:10.1007/978-1-4684-2001-2_9.

22 L. A. Levin. Universal sequential search problems. volume 9, pages 265 – 266, 1973. [Probl. Peredachi Inf., **9**,3:115-116, 1973].

23 Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 2:345–364, 1994.

24 Michael Norrish. Mechanised computability theory. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 297–311, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

25 Christos H. Papadimitriou. *Computational Complexity*. John Wiley and Sons Ltd., GBR, 2003.

26 Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

27 C. Slot and P. van Emde Boas. On tape versus core an application of space efficient perfect hash functions to the invariance of space. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 391–400, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800057.808705.

28 G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. doi:10.1007/978-3-642-81955-1_28.