

Existential length universality

Paweł Gawrychowski
Institute of Computer Science
University of Wrocław
Wrocław, Poland
gawry@cs.uni.wroc.pl

Narad Rampersad
Department of Math/Stats
University of Winnipeg
515 Portage Ave.
Winnipeg, MB R3B 2E9
Canada
narad.rampersad@gmail.com

Jeffrey Shallit
School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
shallit@cs.uwaterloo.ca

Marek Szykuła
Institute of Computer Science
University of Wrocław
Wrocław, Poland
msz@cs.uni.wroc.pl

October 9, 2019

Abstract

We study the following natural variation on the classical universality problem: given a language $L(M)$ represented by M (e.g. DFA/RE/NFA/PDA), does there exist an integer $\ell \geq 0$ such that $\Sigma^\ell \subseteq L(M)$? The case of an NFA was an open problem since 2009. Here, using a novel construction, we prove that the problem is NEXPTIME-complete, and the smallest such ℓ can be doubly exponential in the number of states. In the case of a DFA the problem is NP-complete, and $e^{\sqrt{n \log n}(1+o(1))}$ is a tight upper bound for the smallest such ℓ , where n is the number of states. In the case of a PDA this problem is recursively unsolvable, while the smallest such ℓ cannot be bounded in the number of states by any computable function. In all these cases the problem becomes computationally easier when the length ℓ is also given in the input.

1 Introduction

The classical universality problem is the question, for a given language L over an input alphabet Σ , whether all words over Σ belong to L , i.e. whether $\Sigma^* = L$. Depending on how

L is specified, the complexity of this problem varies. For example, when L is given as a DFA, the problem is easily solvable in linear time (reachability of a non-final state) and is NL-complete [15]. In the case of an NFA or a regular expression it is PSPACE-complete [1]. When L is a PDA (push-down automaton) or a context-free grammar, the problem is undecidable [11].

Studies on universality problems have a long tradition in computer science and still gain a lot of interest (see e.g., [1, 2, 5, 6, 7, 9, 10, 12, 13, 14, 16, 17, 18, 19, 24]). The question about universality is fundamental and is commonly studied as one of the first problems for a new class of automata/languages, as in some sense it measures overall expressiveness and difficulty of the class. For instance, universality has been studied for visibly push-down automata [2], where the question was shown to be decidable in this model in contrast to undecidability in the ordinary model, timed automata [5], the language of all prefixes (resp. suffixes, factors, subwords) of the given language [19], and recently, for partially (and restricted partially) ordered NFAs [17]. Some other applications of universality can be found for example in the context of knowledge representation and database theory (e.g. [4, 23]).

Even though that complexity of deciding universality gives us some insight into the expressiveness of the given formalism, a universal language is actually trivial and so we would prefer a more fine-grained measure. Therefore, it is natural to ask about variations of this basic question, such as restricting our attention to words of certain lengths.

In this paper we consider the very basic variation of universality asking whether there exists just a single length that is universal in the language. Aside from the fact that it already involves interesting computational complexity issues, this variation is a natural starting point for further investigations of other restricted universality problems.

Problem 1 (Existential length universality). *Given a language L represented by M of some type (DFA/RE/NFA/PDA) over an input alphabet Σ of a fixed size, does there exist an integer $\ell \geq 0$ such that $\Sigma^\ell \subseteq L(M)$?*

Furthermore, if such an ℓ exists, we are interested in how large it can be. From the mathematical point of view, this generalizes the classical Chinese remainder theorem to languages, in the sense that given periodicities with multiple periods, we ask where all these periodicities coincide. In fact, the standard encoding by the remainders when divided by prime numbers can be represented by a DFA with prime cycles with final states corresponding to the remainders: from the initial state we can enter each of the cycles by different letters, and then the DFA accepts if and only if the rest of the word's length modulo the cycle length equals the remainder of the encoding.

Second, this gives us some insight into expressiveness and succinctness of particular representations of languages, and in particular relates to their computational complexity. Indeed, a natural approach to solve Problem 1 is to guess ℓ and then verify that indeed $\Sigma^\ell \subseteq L(M)$. If we can argue that for a particular class of machines, if there exists such an ℓ then there actually exists such an ℓ that is not too large, then this approach might lead to an efficient algorithm. Therefore, we should analyze how large such smallest ℓ , called the *minimal universality length*, can be, and consider the complexity of the following decision problem:

Table 1: Computational complexity of universality problems.

	DFA	RE	NFA	PDA
Universality	NL-c	PSPACE-c	PSPACE-c	Undecidable
Existential length universality (Problem 1)	NP-c	PSPACE-hard, in NEXPTIME	NEXPTIME-c	Undecidable
Given length universality (Problem 2)	PTIME	PSPACE-c	PSPACE-c	PSPACE-hard, in coNEXPTIME

Problem 2 (Given length universality). *Given an M and an integer ℓ (represented in binary), is $\Sigma^\ell \subseteq L(M)$?*

1.1 Contributions

The outline of the paper and our contributions are as follows: In Section 2, we study the problem where M is a pushdown automaton. Here existential length universality is recursively unsolvable, while the minimal universality length grows faster than any computable function.

In Section 3, we consider the case when M is a deterministic finite automaton. Here existential length universality is NP-complete, and there exist n -state DFAs for which the minimal universality length is of the form $e^{\sqrt{n \log n}(1+o(1))}$, which is the best possible even when the input alphabet is binary.

The main technical contribution is presented in Section 4, where M is a non-deterministic finite automaton. This particular case was an open question since 2009 [22]. It is easy to show PSPACE-hardness (by modifying the proof from [1, Section 10.6]) and an NEXPTIME algorithm (by determinization and applying the bounds for DFAs), and it was not known what is the right complexity class. We show that, in fact, the problem is NEXPTIME-complete for NFAs. While to this end we reduce a standard NEXPTIME-hard problem, designing the reduction requires a non-trivial insight into the structure of the problem and quite a bit of work. We start with designing a particular intermediate formalism that forms a programming language and makes designing the reduction easier. As the first byproduct of the formalism, we are then able to show that there exist NFAs with the minimal universality length that is doubly exponential in the number of states. Then we proceed to the reduction itself.

We finish the paper with Section 5, where we consider the case when M is a regular expression. The existential length universality is PSPACE-hard and in NEXPTIME, and there are examples where the minimal universality length is exponential. The question about the exact complexity class in this case remains open.

The given length universality problem is easier than the existential length universality in all cases (except we do not know this for regular expressions): it is in coNEXPTIME for PDAs, polynomial for DFAs, and PSPACE-complete for both NFAs and regular expressions.

Our results for both problems are summarized in Tab. 1.

While in proving hardness we use larger alphabets than binary, a standard binarization applies to our problems and so all the complexity results remain valid when the input alphabet is binary.

Lemma 3. *Let Σ be an alphabet of size $k \geq 2$.*

1. *For an M being a DFA/NFA/PDA with n states over Σ , we can construct in polynomial time a DFA (NFA, PDA, respectively) M' over a binary alphabet with $(2^{\lceil \log_2 k \rceil} - 1)n$ states such that its minimal universality length is equal to $\ell \cdot \lceil \log_2 k \rceil$ where ℓ is the minimal universality length for M , or it does not exist if the minimal universality length does not exist for M .*
2. *For a regular expression M with n input symbols over Σ , we can construct in polynomial time a regular expression M' over a binary alphabet with at most $(2^{\lceil \log_2 k \rceil} - 1)n$ input symbols such that its the minimal universality length is equal to $\ell \cdot \lceil \log_2 k \rceil$ where ℓ is the minimal universality length for M , or it does not exist if the minimal universality length does not exist for M .*

Proof. (1) We create M' over the alphabet $\{0, 1\}$ by replacing every state with a full binary tree of height $\lceil \log_2 k \rceil - 1$ (so with $2^{\lceil \log_2 k \rceil} - 1$ states). The transitions are set in the way that for an i 'th letter $a_i \in \Sigma$, the binary representation of i acts on the roots of the states in M' as a_i in M on the corresponding states. The remaining binary representations just duplicate the action of any other one. The final states of M' are the roots of the trees corresponding to the final states of M .

(2) Similarly, a regular expression can be converted to one over $\{0, 1\}$ by replacing each i 'th letter $a_i \in \Sigma$ with either its binary representation or the sum (union) of two binary representations, which all are of length $\lceil \log_2 k \rceil$. Since the number of representations $2^{\lceil \log_2 k \rceil}$ is smaller than $2k$, in this way all of them can be assigned to some letter. \square

2 The case where M is a PDA

Theorem 4. *Existential length universality (Problem 1) is recursively unsolvable for PDAs.*

Proof. We modify the usual proof [11, Thm. 8.11, p. 203] that “Given a PDA M , is $L(M) = \Sigma^*$?” is an unsolvable problem.

In that proof, we start with a Turing machine T , assumed to have a single halt state h and no transitions out of this halting state. We consider the language L of all valid accepting computations of T . A valid computation consists of a sequence of configurations of the machine, separated by a delimiter $\#$. Every second configuration is reversed, i.e. the configurations are written unreversed and reversed alternatingly. Each unreversed configuration is of the form xqy , where xy is the TM's tape contents, q is the TM's current state, and the TM is scanning the first symbol of y . A valid computation must start with $\#q_0x\#$ for some string x , and end with $\#yhz\#$ for some strings y and z , where h is the halt state.

Furthermore, two consecutive configurations inside a valid computation must follow by rules of the TM.

Thus we can accept \bar{L} with a PDA by checking (non-deterministically) if a given string begins wrong, ends wrong, is syntactically invalid, or has two consecutive configurations that do not follow by rules of T . Only this last requirement presents any challenge. The idea is to push the first configuration on, then pop it off and compare to the next (this is why every other configuration needs to be reversed). Comparing two configurations just requires matching symbols, except in the region of the state, where we must verify that the second configuration follows by rules of T from the first.

Thus, a PDA can be constructed to accept \bar{L} , the set of all strings that *do not* represent valid accepting computations of T . Hence “Given T , is $L(T) = \Sigma^*$?” is unsolvable, because if we could answer it, we would know whether or not T accepts some string. This concludes our sketch of the usual universality proof.

Now, to prove our result about existential length universality, we modify the above construction in two ways.

First, we assume that our TM T has the property that there is always a next move possible, except from the halting state. Thus, the only possibilities on any input are (1) arriving to the halting state h , after which there is no move or (2) running forever without halting.

Second, we make a PDA based on a TM T such that our PDA fails to accept all strings that represent valid halting computations of T that start with empty input, and *also* fails to accept all strings that are prefixes of a valid computation. In other words, our PDA is designed to accept if a computation starts wrong, is syntactically invalid, or if a configuration does not follow from the previous one. If the last configuration is incomplete, and what is actually present does not violate any rules of T , however, our PDA *does not* accept.

Now, suppose that T does not accept the empty string. Then T does not halt on empty input, so there are valid computations on every input for every number of steps. So there are strings representing valid computations, or prefixes of valid computations, of every length. Since M fails to accept these, there is no ℓ such that M accepts all strings of length ℓ .

On the other hand, if T does accept the empty string, then there is exactly one valid halting computation for it; say it is of length s , where the last configuration is of length t . Consider every putative computation of length $\geq s + t + 2$; it must violate a rule, since there are no computations out of the halting state. So M accepts all strings of length $s + t + 2$. Thus we could decide if T accepts the empty string, which means the existential length universality for PDAs is unsolvable. \square

Corollary 5. *Fix an alphabet Σ . For a PDA M , let $\ell(M)$ be the minimal universality length (if it exists) of $L(M)$. Let $f(n)$ be the maximum of $\ell(M)$ over all PDA's M of size n . Then $f(n)$ grows faster than every computable function.*

Proof. If $f(n)$ grew slower than some computable function $g(n)$, then we could solve existential universality for a given PDA M of size n by (1) computing $g(n)$; (2) testing, by a brute-force enumeration, whether M accepts all strings of length ℓ for all $\ell \leq g(n)$. (We

can deterministically test if a PDA accepts a string by converting the PDA to an equivalent context-free grammar and then using the usual Cocke-Younger-Kasami dynamic programming algorithm for CFL membership.) But existential length universality is undecidable for PDA's. \square

In contrast, the given length universality problem is solvable in the case of a PDA. Furthermore, it is in coNEXPTIME : We can convert a PDA in polynomial time to a context-free grammar generating the same language. Then we can guess an (exponentially long in the size of the input) word of length ℓ such that it is not accepted, and verify if it is indeed not generated by the context-free grammar (using, for example, the Cocke-Younger-Kasami dynamic programming solution). We also know that this problem is PSPACE-hard (Theorem 24).

Open Question 6. What is the computational complexity of the given length universality problem in the case of a PDA?

3 The case where M is a DFA

Theorem 7. *Existential length universality (Problem 2) for DFAs is in NP.*

Proof. Start with a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that we assume to be complete (that is, $\delta(q, a)$ is defined for all $q \in Q$ and $a \in \Sigma$).

From M create a unary NFA M' that is defined by taking every transition of M labeled with an input letter and replacing that letter with the single letter a . Now it is easy to see that M accepts all strings of length ℓ if and only if every path of length ℓ in M' , starting with its initial state, ends in a final state.

Now create a Boolean matrix B with the property that there is a 1 in row i and column j if and only if M' has a transition from q_i to q_j , and 0 otherwise. It is easy to see that M' has a path of length ℓ from state q_i to state q_j if and only if B^ℓ has a 1 in row i and column j , where by B^ℓ we mean the Boolean power of the matrix B .

So M accepts all strings of length ℓ if and only if every 1 in row 0 (corresponding to q_0) of B^m occurs only in columns corresponding to the final states of M' . Hence, to verify that M is length universal for some ℓ , we simply guess ℓ , compute B , raise B to the ℓ 'th Boolean power using the usual “binary method” or “doubling up” trick, and check the positions of the 1's in row 0. This can be done in polynomial time provided ℓ is exponentially bounded in magnitude.

To see that m is exponentially bounded, we can argue that $\ell \leq 2^{|Q|^2}$. This follows trivially because our matrix is of dimension $|Q| \times |Q|$; after we see all $2^{|Q|^2}$ different powers, we have seen all we can see, and the powers must cycle after that. \square

Actually, we can do even better than the $2^{|Q|^2}$ bound in the proof. It is an old result of Rosenblatt [20] that powers of a $t \times t$ Boolean matrix are ultimately periodic with preperiod of size $O(t^2)$ and period of size at most $e^{\sqrt{t \log t}(1+o(1))}$. Thus we have

Theorem 8. *Let M be a DFA with n states. If there exists a minimal universality length ℓ for M , then $\ell \leq e^{\sqrt{n \log n}(1+o(1))}$.*

The upper bound in the previous result is tight, even for a binary alphabet.

Theorem 9. *For each sufficiently large n , there exists a binary DFA M with n states for which the minimal universality length ℓ is $\geq e^{\sqrt{n \log n}(1+o(1))}$.*

Proof. First we construct our DFA with t input symbols: There is a non-accepting initial state with transitions out on symbols a_1, a_2, \dots, a_t to cycles of size $p(1), p(2), \dots, p(t)$, respectively, where $p(i)$ is the i 'th prime number. The transitions on each cycle are on all symbols of the alphabet. Inside each cycle all states are non-accepting, except the state immediately before the state with an incoming transition from the initial state, which is accepting. This DFA accepts the language

$$\bigcup_{1 \leq i \leq t} a_i(\Sigma^{p(i)})^* \Sigma^{p(i)-1}.$$

For each length $\ell' < p(1)p(2) \cdots p(t) - 1$, there exists a prime number $p(i)$, $1 \leq i \leq t$, such that $\ell' \not\equiv p(i) - 1 \pmod{p(i)}$, so no string in $a_i \Sigma^{\ell'}$ is accepted. However, for $\ell = p(1)p(2) \cdots p(t)$, all strings of length ℓ are accepted.

Now we convert the DFA to a binary DFA over $\{0, 1\}$ by replacing the initial state with a full binary tree of height $h = \lceil \log_2 t \rceil - 1$, so that the binary representation of i maps the initial state (root) to the state from the cycle of length $p(i)$ (cf. Lemma 3). This DFA has $p(1) + p(2) + \cdots + p(t) + 2^{h+1} - 1$ states, and the least ℓ for which all strings of length ℓ is accepted is $p(1)p(2) \cdots p(t) + h - 1$.

From the prime number theorem we know that

$$p(1) + p(2) + \cdots + p(t) + O(t) \sim \frac{1}{2} t^2 \log t,$$

(see, for example, [3, p. 29]) and $p(1)p(2) \cdots p(t) + O(t) \sim e^{t(1+o(1))}$, from which the claim follows. \square

Theorem 10. *Existential length universality (Problem 1) is NP-hard for DFAs.*

Proof. We reduce from 3-SAT. Given a formula φ in 3-CNF form we create a DFA $M = M_\varphi$ having the property that there exists an integer $\ell \geq 0$ such that M accepts all strings of length ℓ if and only if φ has a satisfying assignment.

To do so, we use the ideas from the usual Chinese-remainder-theorem-based proof of the coNP-hardness of deciding if $L(M) = \Sigma^*$ for a unary NFA M . [24].

Suppose there are t variables in φ , say v_1, v_2, \dots, v_t . Then satisfying assignments are coded by integers which are either congruent to 0 or 1 modulo the first t primes. Let $p(i)$ be the i 'th prime number. If the integer is $1 \pmod{p(i)}$, then it corresponds to an assignment where v_i is set to 1; if the integer is $0 \pmod{p(i)}$, then it corresponds to an assignment where v_i is set to 0. Given a clause C consisting of 3 literals (variables v_i, v_j, v_k or their negations), all

integers corresponding to a satisfying assignment of this particular clause fall into a number of residue classes modulo $p(i)p(j)p(k)$.

We now construct a DFA with an alphabet Σ consisting of the integers $1, 2, \dots, s$, where s is the number of clauses. The non-accepting initial state has a transition on each letter of Σ to a cycle corresponding to the appropriate clause. Inside each cycle, the transitions go to the next state on all letters of Σ . Each cycle is of size $p(i)p(j)p(k)$, where v_i, v_j, v_k are the variables appearing in the corresponding clause. The accepting states in each cycle are the integers, modulo $p(i)p(j)p(k)$, that correspond to assignments satisfying that clause.

We claim this DFA accepts all strings of length ℓ if and only if ℓ corresponds to a satisfying assignment for φ . To see this, note that if the DFA accepts all strings of length ℓ for some ℓ , then the path inside each cycle must terminate at an accepting state, which corresponds to a satisfying assignment for each clause. On the other hand, if ℓ corresponds to a satisfying assignment, then every string is accepted because it satisfies each clause, and hence corresponds to a path beginning with any clause number and entering the appropriate cycle.

The transformation uses polynomial time because the i 'th prime number is bounded in magnitude by $O(i \log i)$ (e.g., [21]), and each cycle is therefore of size at most $O((t \log t)^3)$, so the total number of states is $O(s(t \log t)^3)$. \square

Theorem 11. *Given length universality (Problem 2) for DFAs is solvable in polynomial time (in the size of M and $\log \ell$).*

Proof. It is similar to the proof of Theorem 7, but we do not have to guess ℓ , so it is verified in deterministic time. \square

4 The case where M is an NFA

The classical universality problem for regular expressions and so for NFAs is known to be PSPACE-complete [1, Section 10.6]. Also, if the NFA does not accept Σ^* , then the length of the shortest non-accepted words is at most exponential. Given length universality for NFAs is also PSPACE-complete (Theorem 24).

However, we show that existential length universality is harder: it is NEXPTIME-complete, and there are examples where the minimal universality length is approximately doubly exponential in the number of states of the NFA.

We begin with upper bounds, which follow from the results for DFAs.

Proposition 12. *Let M be an NFA with n states. If there exists ℓ such that M accepts all strings of length ℓ , then the smallest such ℓ is $\leq e^{2^{n/2} \sqrt{n \log 2} (1+o(1))}$.*

Proof. By determinizing M to a DFA with at most 2^n states and applying Theorem 8 we get

$$\ell \leq e^{\sqrt{2^n \log 2^n} (1+o(1))} = e^{2^{n/2} \sqrt{n \log 2} (1+o(1))}.$$

\square

Proposition 13. *Existential length universality (Problem 1) for NFAs is in NEXPTIME.*

Proof. We determinize M to a DFA with at most 2^n states. Then by Theorem 7 the exponential length universality is solvable in non-deterministic exponential time. \square

The difficult part is to show that the existential length universality for NFAs is NEXPTIME-hard. To this end, we reduce the canonical NEXPTIME-complete problem of deciding whether a non-deterministic Turing machine accepts an empty input within at most exponential number of steps. Also, we construct NFAs with large minimal universality lengths that are close to the upper bound given in Proposition 12. The constructions are rather involved, and hence we first develop an intermediate formalism that will be used for both tasks.

4.1 A programming language

We define a simple programming language that will be translated to an NFA. The general idea is that a word not accepted by an NFA will have to describe a proper computation of the program of this NFA, and sometimes additionally satisfy some other conditions, whereas bad computations will be always accepted.

Let $m \geq 1$ be a fixed integer. A *variable* V is a set of states $\{v_1, \dots, v_m, \bar{v}_1, \dots, \bar{v}_m\}$. These states are called *variable states*, and m is the *width* of the variable. Besides variable states, in our NFAs there will be also *control flow states*, and the unique special final state q_{acc} , which will be fixed by all transitions.

A *gadget* G is a 7-tuple $(P^G, \mathcal{V}^G, \Sigma^G, \delta^G, s^G, t^G, F^G)$. When specifying the elements of such a tuple, we usually omit the superscript if it is clear from the context. P is a set of control flow states, \mathcal{V} is a set of (disjoint) variables on which the gadget *operates*, $s, t \in P$ are distinguished *start* and *target* control flow states, respectively, and $F \subseteq P$ is a set of final states. The set of states of G is $Q = \{q_{\text{acc}}\} \cup P \cup \bigcup_{V \in \mathcal{V}} V$. Then $\delta: Q \times \Sigma \rightarrow 2^Q$ is the transition function, which is extended to a function $2^Q \times \Sigma^* \rightarrow 2^Q$ as usual. We always have $\delta(q_{\text{acc}}, a) = \{q_{\text{acc}}\}$ for every $a \in \Sigma$.

The NFA of G is $(Q, \Sigma, \delta, s, F)$. A *configuration* is a subset $C \subseteq Q$. Given C , we say that a state is *active* if it belongs to C . We say that a configuration C is *proper* if it does not contain q_{acc} . Given a proper configuration C and a word w , we say that w is a *computation from* C . It is a *proper computation from* C if the obtained configuration after reading this word is also proper (i.e. $\delta(C, w)$ is proper). Therefore, from a non-proper configuration we cannot obtain a proper one after reading any word, since q_{acc} is always fixed, and so every non-proper computation from $\{s\}$ is an accepted word by the NFA.

We say that a variable V is *valid* in a configuration $C \subseteq Q$ if $v_i \in C$ if and only if $\bar{v}_i \notin C$. In other words, the states $\bar{v}_1, \dots, \bar{v}_m$ are complementary to the states v_1, \dots, v_m . A valid variable stores an integer from $\{0, \dots, 2^m - 1\}$; the states v_1 and v_m represent the least and the most significant bit, respectively. Formally, if V is valid in a configuration C , then its

value $V(C)$ is defined as

$$V(C) = \sum_{\substack{1 \leq i \leq m \\ v_i \in V \cap C}} 2^{i-1}.$$

We say that a configuration C is *initial* for a gadget G if it is proper, contains the start state s but no other control flow states, and the gadget's variables are valid in C (if not otherwise stated, which is the case for some gadgets). A *final* configuration is a proper configuration that contains the target state t . A *complete computation* is a proper computation from an initial configuration to a final configuration. Every gadget will possess some properties about its variables and the length of complete computations according to its semantics. These properties are of the form that, depending on an initial configuration C , there exists or not a complete computation of some length from C to a final configuration C' , where C' also satisfies some properties. Also, usually proper computations from an initial configuration will have bounded length. If a variable is not required to be valid in C , then these properties will not depend on its active states in C .

We start from defining *basic* gadgets, which are elementary building blocks, and then we will define *compound* gadgets, which use other gadgets inside.

4.1.1 Basic gadgets

- **Selection Gadget.**

It is denoted by $\text{SELECT}(V)$, where V is a variable. The gadget allows a non-deterministic selection of an arbitrary value for V . For every integer $c \in \{0, \dots, 2^m - 1\}$, for every initial configuration C there exists a complete computation from C to C' such that $V(C') = c$. An initial configuration for this gadget does not require that V is valid.

We have the control flow states $P = \{s = p_0, p_1, \dots, p_{m-1}, p_m = t\}$, one variable V , and the letters $\Sigma = \{\alpha_1, \alpha_2\}$. The gadget is illustrated in Fig. 1.

The letters α_1 and α_2 allow mapping the active control flow state to the states $p_0, p_1, \dots, p_{m-1}, p_m$ and, at each transition, to choose either v_1 or \bar{v}_1 to be active. Also, each v_i and \bar{v}_i are shifted to v_{i+1} and \bar{v}_{i+1} , respectively, and both v_m and \bar{v}_m are mapped to \emptyset , which ensures that the initial content of V is neglected. The transitions are defined as follows:

$$\begin{aligned} \delta(p_i, \alpha_1) &= \{p_{i+1}, v_1\} \text{ for } i = 0, \dots, m-1, \\ \delta(p_i, \alpha_2) &= \{p_{i+1}, v'_1\} \text{ for } i = 0, \dots, m-1, \\ \delta(p_m, \alpha_1) = \delta(p_m, \alpha_2) &= \{q_{\text{acc}}\}, \\ \delta(v_i, \alpha_1) = \delta(v_i = \alpha_2) &= \{v_{i+1}\} \text{ for } i = 1, \dots, m-1, \\ \delta(\bar{v}_i, \alpha_1) = \delta(\bar{v}_i = \alpha_2) &= \{\bar{v}_{i+1}\} \text{ for } i = 1, \dots, m-1, \\ \delta(v_m, \alpha_1) = \delta(\bar{v}_m = \alpha_2) &= \emptyset. \end{aligned}$$

The semantic properties are summarized in the following:

Lemma 14. *Let C be an initial configuration for the Selection Gadget $\text{SELECT}(V)$. For every value $c \in \{0, \dots, 2^m - 1\}$, there exists a complete computation in Σ^m from C to a*

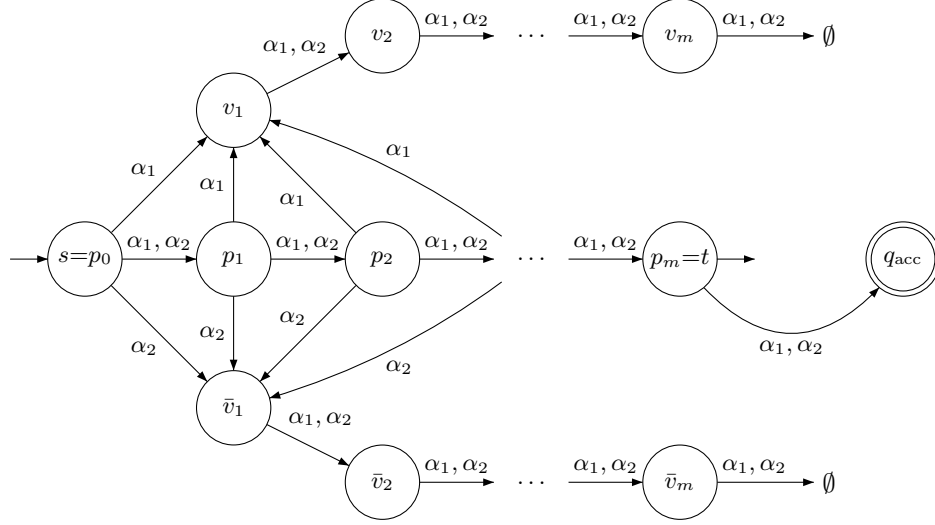


Figure 1: Selection Gadget.

configuration C' such that $V(C') = c$. Every complete computation has length m , and every longer computation is not proper.

Proof. This follows from the construction in a straightforward way. After reading a word $w \in \{\alpha_1, \alpha_2\}^m$ we get that t is active, and for every $i = 1, \dots, m$ either v_i or \bar{v}_i is active, depending on the i 'th letter. Thus, for every value of V there is a unique word $w \in \{\alpha_1, \alpha_2\}^m$ resulting in setting this value. If w is shorter than m , then t cannot become active, since the shortest path from s to t has length m . Longer computations are not proper since both letters map t to q_{acc} . \square

• Equality Gadget.

It is denoted by $U = V$, where U and V are two distinct variables. The gadget checks if the values of valid variables U and V are equal in the initial configuration. If so, the gadget admits a complete computation, which is of length m ; otherwise, every word of length at least m is a non-proper computation. We have the control flow states $P = \{s = p_0, p_1, \dots, p_m = t\}$, two variables U and V , and the letters $\Sigma = \{\alpha_1, \alpha_2\}$. The gadget is illustrated in Fig. 2.

The letters α_1 and α_2 allow mapping the active control flow state to the states $s = p_0, p_1, \dots, p_m = t$ and simultaneously checking corresponding positions of U and V to see if they agree. The transitions are defined in the same way for both variables, and they

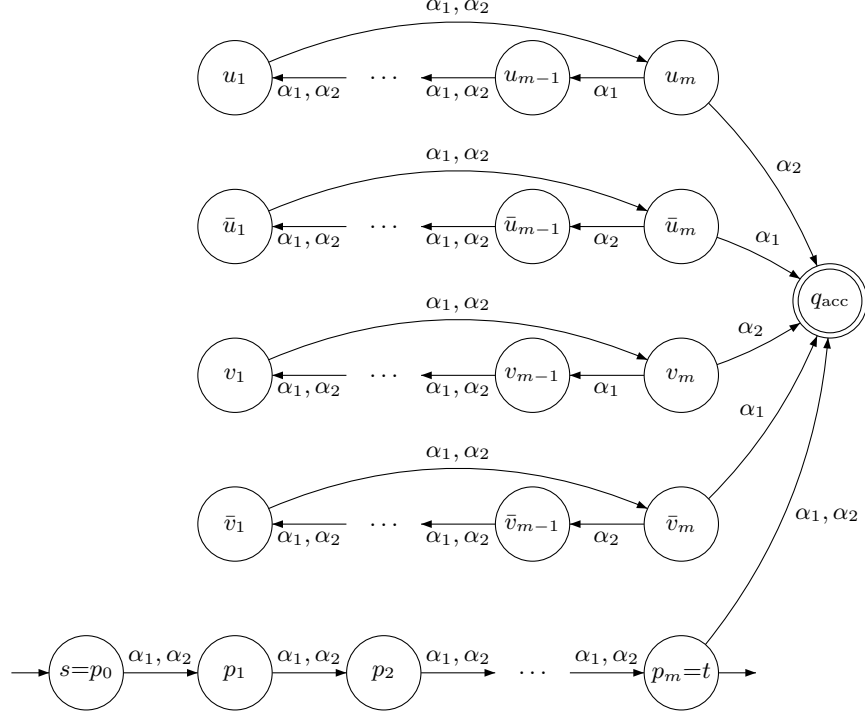


Figure 2: Equality Gadget.

cyclically shift their states.

$$\begin{aligned}
\delta(v_1, \alpha_1) &= \delta(v_1 = \alpha_2) = \{v_m\}, \\
\delta(v_i, \alpha_1) &= \delta(v_i = \alpha_2) = \{v_{i-1}\} \text{ for } i = 1, \dots, m-1, \\
\delta(v_m, \alpha_1) &= \{v_{m-1}\}, \\
\delta(v_m, \alpha_2) &= \{q_{acc}\}, \\
\delta(\bar{v}_1, \alpha_1) &= \delta(\bar{v}_1 = \alpha_2) = \{\bar{v}_m\}, \\
\delta(\bar{v}_i, \alpha_1) &= \delta(\bar{v}_i = \alpha_2) = \{\bar{v}_{i-1}\} \text{ for } i = 1, \dots, m-1, \\
\delta(\bar{v}_m, \alpha_1) &= \{q_{acc}\}, \\
\delta(\bar{v}_m, \alpha_2) &= \{\bar{v}_{m-1}\}, \\
\delta(u_1, \alpha_1) &= \delta(u_1 = \alpha_2) = \{u_m\}, \\
\delta(u_i, \alpha_1) &= \delta(u_i = \alpha_2) = \{u_{i-1}\} \text{ for } i = 1, \dots, m-1, \\
\delta(u_m, \alpha_1) &= \{u_{m-1}\}, \\
\delta(u_m, \alpha_2) &= \{q_{acc}\}, \\
\delta(\bar{u}_1, \alpha_1) &= \delta(\bar{u}_1 = \alpha_2) = \{\bar{u}_m\}, \\
\delta(\bar{u}_i, \alpha_1) &= \delta(\bar{u}_i = \alpha_2) = \{\bar{u}_{i-1}\} \text{ for } i = 1, \dots, m-1, \\
\delta(\bar{u}_m, \alpha_1) &= \{q_{acc}\}, \\
\delta(\bar{u}_m, \alpha_2) &= \{\bar{u}_{m-1}\}, \\
\delta(p_i, \alpha_1) &= \delta(p_i, \alpha_2) = \{p_{i+1}\} \text{ for } i = 0, \dots, m-1, \\
\delta(p_m, \alpha_1) &= \delta(p_m, \alpha_2) = \{q_{acc}\}.
\end{aligned}$$

Lemma 15. *Let C be an initial configuration with valid variables U and V for the Equality Gadget $U = V$. When $U(C) = V(C)$, there exists a complete computation in Σ^m from C to a configuration C' . Moreover, every complete computation has length m and is such that $U(C') = U(C)$ and $V(C') = V(C)$. Longer computations are not proper, and when $U(C) \neq V(C)$, every computation of length at least m is not proper.*

Proof. After reading a word $w \in \{\alpha_1, \alpha_2\}^m$ we get that t is active and no shorter word has this property. If $u_i \in C$ then the $(i+1)$ 'st letter of w (or first letter if $i = m$) must be α_1 . Similarly, if $\bar{u}_i \in C$ then the $(i+1)$ 'st letter of w (or first if $i = m$) must be α_2 . The same holds for the states of V . Thus, if $u_i \in C$ and $\bar{v}_i \in C$, or if $\bar{u}_i \in C$ and $v_i \in C$, then q_{acc} cannot be avoided by any such a word w . Otherwise, there exists a unique word w of length m such that t becomes active and q_{acc} does not. \square

• **Inequality Gadget.**

It is denoted by $U \neq V$, where U and V are two distinct variables. This gadget is similar to the Equality Gadget, and checks if the values of valid variables U and V are different. We have the control flow states $P = \{s = p_0, p_1, \dots, p_{2m} = t\}$, two variables U and V , and the letters $\Sigma = \{\alpha_1, \alpha_2, \alpha_s\}$. The gadget is illustrated in Fig. 3.

The letter α_s allows mapping the active control flow state to the states p_0, p_1, \dots, p_m , and it cyclically shifts the states of both variables. Its transition function is defined as follows:

$$\begin{aligned}
\delta(u_1, \alpha_s) &= \{u_m\}, \\
\delta(u_i, \alpha_s) &= \{u_{i-1}\} \text{ for } i = 1, \dots, m, \\
\delta(\bar{u}_1, \alpha_s) &= \{\bar{u}_m\}, \\
\delta(\bar{u}_i, \alpha_s) &= \{\bar{u}_{i-1}\} \text{ for } i = 1, \dots, m, \\
\delta(v_1, \alpha_s) &= \{v_m\}, \\
\delta(v_i, \alpha_s) &= \{v_{i-1}\} \text{ for } i = 1, \dots, m, \\
\delta(\bar{v}_1, \alpha_s) &= \{\bar{v}_m\}, \\
\delta(\bar{v}_i, \alpha_s) &= \{\bar{v}_{i-1}\} \text{ for } i = 1, \dots, m, \\
\delta(p_i, \alpha_s) &= \{p_{i+1}\} \text{ for } i = 0, \dots, m-1, \\
\delta(p_{m+i}, \alpha_s) &= \{p_{m+i+1}\} \text{ for } i = 1, \dots, m-1,
\end{aligned}$$

At some point, at the position where the variables differ, either α_1 or α_2 can be applied, which also switches the active control flow state from p_i to the corresponding p_{m+i} . Their

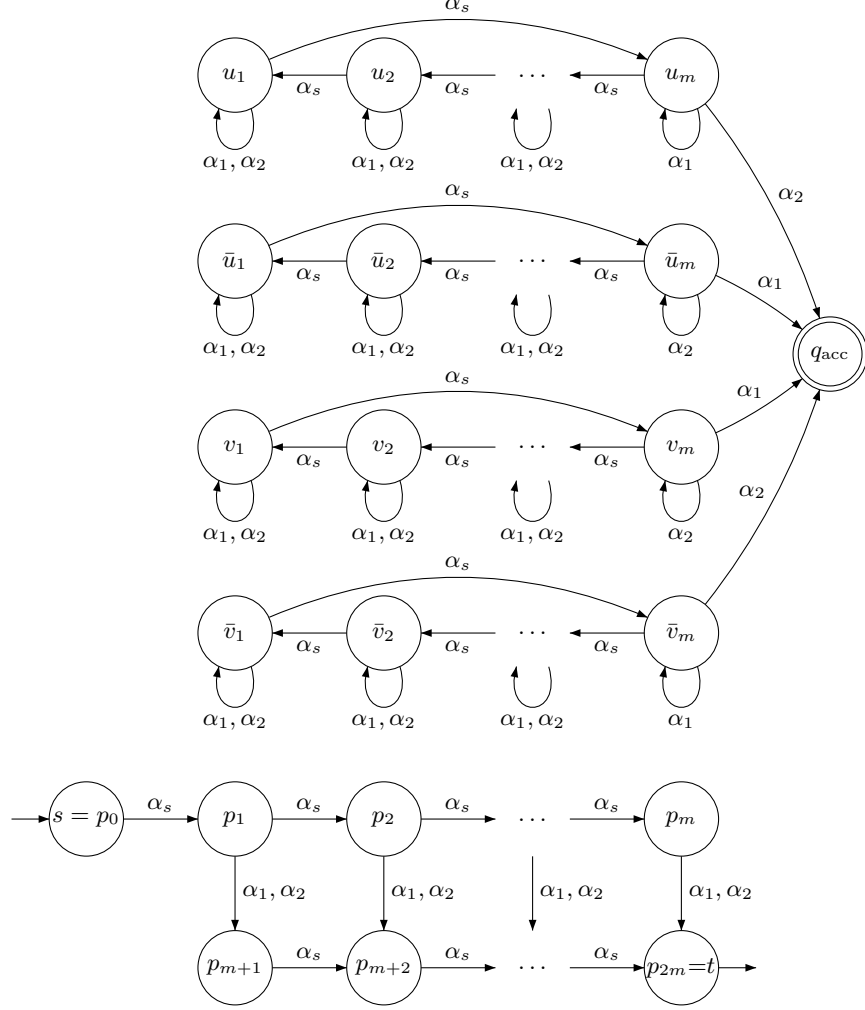


Figure 3: Inequality Gadget. All omitted transitions go to q_{acc} .

transitions are defined as follows:

$$\begin{aligned}
\delta(u_i, \alpha_1) = \delta(u_i, \alpha_2) &= \{u_i\} \text{ for } i = 1, \dots, m-1, \\
\delta(\bar{u}_i, \alpha_1) = \delta(\bar{u}_i, \alpha_2) &= \{\bar{u}_i\} \text{ for } i = 1, \dots, m-1, \\
\delta(v_i, \alpha_1) = \delta(v_i, \alpha_2) &= \{v_i\} \text{ for } i = 1, \dots, m-1, \\
\delta(\bar{v}_i, \alpha_1) = \delta(\bar{v}_i, \alpha_2) &= \{\bar{v}_i\} \text{ for } i = 1, \dots, m-1, \\
\delta(u_m, \alpha_1) &= \{u_m\}, \\
\delta(\bar{v}_m, \alpha_1) &= \{\bar{v}_m\}, \\
\delta(\bar{u}_m, \alpha_2) &= \{\bar{u}_m\}, \\
\delta(v_m, \alpha_2) &= \{v_m\}, \\
\delta(\bar{u}_m, \alpha_1) = \delta(v_m, \alpha_1) = \delta(u_m, \alpha_2) = \delta(\bar{v}_m, \alpha_2) &= \{q_{\text{acc}}\}, \\
\delta(p_i, \alpha_1) = \delta(p_i, \alpha_2) &= \{p_{m+i}\} \text{ for } i = 1, \dots, m.
\end{aligned}$$

Lemma 16. *Let C be an initial configuration with valid variables U and V for the Inequality Gadget $U \neq V$. When $U(C) \neq V(C)$, there exists a complete computation in Σ^{m+1} from C to a configuration C' . Moreover, every complete computation has length $m + 1$ and is such that $U(C') = U(C)$ and $V(C') = V(C)$. Longer computations are not proper, and when $U(C) = V(C)$, every computation of length at least $m + 1$ is not proper.*

Proof. Consider a word w of length $m + 1$. If q_{acc} does not become active, then by the structure of control flow states, t must become active and w contains exactly m occurrences of α_s and one occurrence of either α_1 or α_2 . If α_1 is the i 'th letter of w ($2 \leq i \leq m + 1$), then it must be that $u_{i-1} \in C$ and $v_{i-1} \notin C$. This is dual for α_2 . Therefore, there must exist a position at which the variables differ.

Conversely, if the variables differ at an i 'th position, then either the word $\alpha_s^i \alpha_1 \alpha_s^{m-i}$ or $\alpha_s^i \alpha_2 \alpha_s^{m-i}$ does the job. \square

• **Incrementation Gadget.**

It is denoted by $V++$, where V is a variable. The gadget increases the value of valid variable V by 1. If the value of V is the largest possible ($2^m - 1$), then the gadget does not allow to obtain a proper configuration by a word of at least a certain length ($m + 1$). Variable V must be valid in an initial configuration. We have the control flow states $P = \{s = p_0, p_1, \dots, p_{2m} = t\}$, one variable V , and the letters $\Sigma = \{\alpha_a, \alpha_c, \alpha_p\}$. The gadget is illustrated in Fig. 4.

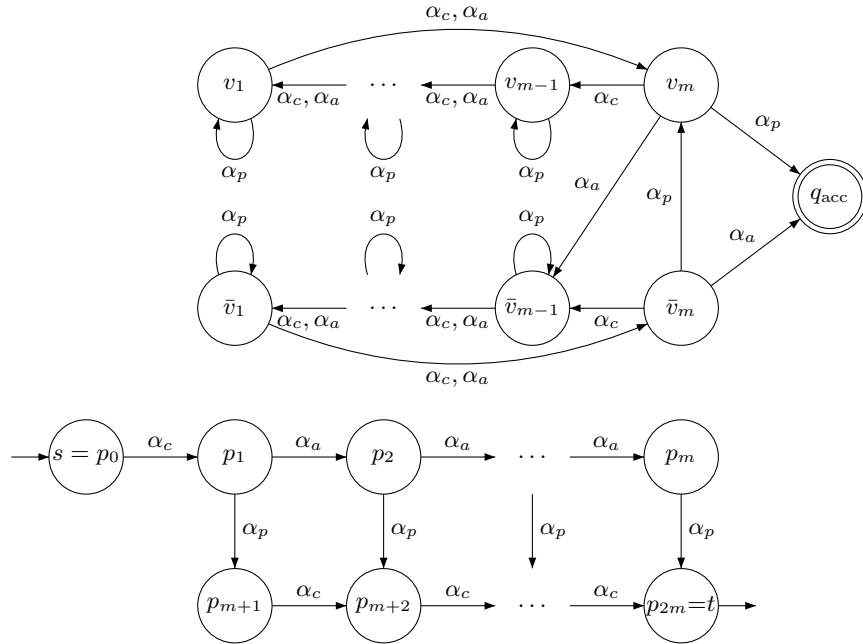


Figure 4: Incrementation Gadget. All omitted transitions go to q_{acc} .

The incrementation gadget performs the written addition of one to the value of V interpreted in binary. First, the letter α_c begins the process: this is the only letter that can be

applied when p_0 is active. Its transitions are defined as follows:

$$\begin{aligned}
\delta(p_0, \alpha_c) &= \{p_1\}, \\
\delta(v_1, \alpha_c) &= \{v_m\}, \\
\delta(v_i, \alpha_c) &= \{v_{i-1}\} \text{ for } i = 2, \dots, m, \\
\delta(\bar{v}_1, \alpha_c) &= \{\bar{v}_m\}, \\
\delta(\bar{v}_i, \alpha_c) &= \{\bar{v}_{i-1}\} \text{ for } i = 2, \dots, m, \\
\delta(p_{m+i}, \alpha_c) &= \{p_{m+i+1}\} \text{ for } i = 1, \dots, m-1.
\end{aligned}$$

Then, the letter α_a cyclically shifts the states of V , and must be applied until the m 'th position in V is empty. Every time when it is applied, the m 'th position becomes cleared. Its transitions are defined as follows:

$$\begin{aligned}
\delta(v_1, \alpha_a) &= \{v_m\}, \\
\delta(v_i, \alpha_a) &= \{v_{i-1}\} \text{ for } i = 2, \dots, m-1, \\
\delta(v_m, \alpha_a) &= \{\bar{v}_{m-1}\}, \\
\delta(\bar{v}_1, \alpha_a) &= \{\bar{v}_m\}, \\
\delta(\bar{v}_i, \alpha_a) &= \{\bar{v}_{i-1}\} \text{ for } i = 2, \dots, m-1, \\
\delta(\bar{v}_m, \alpha_a) &= \{q_{\text{acc}}\}, \\
\delta(p_i, \alpha_a) &= \{p_{i+1}\} \text{ for } i = 1, \dots, m-1.
\end{aligned}$$

Then, the letter α_p must be applied, which fills the m 'th position in V , and the active state p_i is moved to the corresponding state p_{m+i} . Its transition are as follows:

$$\begin{aligned}
\delta(v_m, \alpha_p) &= \{q_{\text{acc}}\}, \\
\delta(\bar{v}_m, \alpha_p) &= \{v_m\}, \\
\delta(v_i, \alpha_p) &= v_i \text{ for } i = 1, \dots, m-1, \\
\delta(\bar{v}_i, \alpha_p) &= \bar{v}_i \text{ for } i = 1, \dots, m-1, \\
\delta(p_i, \alpha_p) &= \{p_{m+i}\} \text{ for } i = 1, \dots, m.
\end{aligned}$$

Finally, the letter α_c finishes the cyclic shifting of V , while moving the active control flow state over p_{m+i}, \dots, p_{2m} .

Lemma 17. *Let C be an initial configuration with valid variable V for the Incrementation Gadget $V++$. If $V(C) < 2^m - 1$, then there exists a complete computation in Σ^{m+1} from C to a configuration C' . Moreover, every complete computation has length $m+1$ and is such that $V(C') = V(C) + 1$. Longer computations are not proper, and if $V(C) = 2^m - 1$, then every computation of length at least $m+1$ is not proper.*

Proof. Consider a word w of length $m+1$ and suppose that the obtained configuration C' is proper. So $t \in C'$ because of moving the active state in P from s to t . Then w must have the following form:

$$w = \alpha_c \alpha_a^i \alpha_p \alpha_c^{m-1-i},$$

for some $i \in \{0, \dots, m-1\}$. Consider the j 'th occurrence of α_a ($1 \leq j \leq i$). If after reading it the current configuration is still proper, then it must be that $\bar{v}_j \notin C$ and so $v_j \in C$, since v_j and \bar{v}_j were mapped to v_m and \bar{v}_m by $\alpha_c \alpha_a^{j-1}$. Next, it must be that $v_i \notin C$, because of the application of α_p . Since w cyclically shifts V exactly m times, we end up with C' such that:

- $v_j \in C'$ and $\bar{v}_j \notin C'$ for $j < i$,
- $v_i \in C'$,
- $v_j \in C'$ if and only if $v_j \in C$, and $\bar{v}_j \in C'$ if and only if $\bar{v}_j \in C$, for $j > i$.

Thus the value $V(C')$ is $V(C) - 2^1 - \dots - 2^{i-1} + 2^i = V(C) + 1$.

If $V(C) < 2^m - 1$, then there exists a smallest $i \leq m$ such that $v_i \notin C$, and there exists the unique word w of that form which does the job. \square

• Assignment Gadget.

It is denoted either by $U \leftarrow c$ or by $U \leftarrow V$, where $c \in \{0, \dots, 2^m - 1\}$ and U and V are two distinct variables. The gadget assigns to U either the fixed constant c or the value from the other variable V . Variable V must be valid in an initial configuration, but U does not have to be. We have two control flow states $P = \{s = p_0, p_1 = t\}$, a variable U or two variables U, V , and the unary alphabet $\Sigma = \{\alpha\}$.

The transition of α maps s to t , and additionally maps either s to the states of U encoding value c or the states of V to the corresponding states of U . The transitions are defined as follows:

$$\delta(u_i, \alpha) = \delta(\bar{u}_i, \alpha) = \emptyset.$$

If it assigns a fixed value c then:

$$\begin{aligned} \delta(s, \alpha) &= \{t\} \cup \{v_i \mid i\text{'th least bit of } c \text{ in binary is } 1\} \\ &\cup \{\bar{v}_i \mid i\text{'th least bit of } c \text{ in binary is } 0\}. \end{aligned}$$

If it assigns the value of V then:

$$\begin{aligned} \delta(s, \alpha) &= \{t\}, \\ \delta(v_i, \alpha) &= \{u_i, v_i\}, \\ \delta(\bar{v}_i, \alpha) &= \{\bar{u}_i, \bar{v}_i\}. \end{aligned}$$

• Waiting Gadget.

It is denoted by $\text{WAIT}(D)$, where D is a fixed positive integer. This is a very simple gadget which just delays the computation for D number of letters. It means that there is exactly one complete computation, which has length D , and longer computations are not proper. There are the states $Q = \{s = p_0, p_1, \dots, p_{D-1}, p_D = t\}$ and the unary alphabet $\Sigma = \{\alpha\}$. The transitions of α are defined as follows:

$$\delta(p_i, \alpha) = \{p_{i+1}\} \text{ for } i = 0, \dots, D-1.$$

4.1.2 Joining gadgets together

The general scheme for creating a compound gadget by joining gadgets G_1, \dots, G_k operating on variables from the sets $\mathcal{V}_1, \dots, \mathcal{V}_k$ (all of width m), respectively, is as follows:

1. There are fresh (unique) control flow states of the gadgets, and there are the variables from $\mathcal{V}_1 \cup \dots \cup \mathcal{V}_k$. Thus when gadgets operate on the same variable, its states are shared.
2. The alphabet contains fresh (unique) copies of the letters of the gadgets.
3. Final states in the gadgets are also final in the compound gadget.
4. The transitions are defined as in the gadgets, whereas the transitions of a letter from a gadget G_i map every control flow state that does not belong to G_i to $\{q_{\text{acc}}\}$, and fix the states of the variables on which G_i does not operate.
5. Particular definitions of compound gadgets may additionally identify some of the start and target states of the gadgets, and may add more control flow states and letters.

In our constructions, the control flow states with their transitions will form a directed graph, where the out-degree at every state of every letter is one (except the Parallel Gadget, defined later, which is an exception from the above scheme) – it either maps a control flow state to another one or to q_{acc} . This will ensure that during every proper computation from an initial configuration, exactly one control flow state is active. The active control flow state will determine which letters can be used by a proper computation, i.e. the letters from the gadget owning this state (but in which it is not the target state).

Moreover, we will take care about that whenever a proper computation activates the start state of an internal gadget G_i , the current configuration restricted to the states of G_i is an initial configuration for G_i – this boils down to assure that the variables required to be valid have been already initialized (by a Selection Gadget or an Assignment Gadget). Hence, complete computations for the compound gadget will contain complete computations for the internal gadgets, and the semantic properties of the compound gadget are defined in a natural way from the properties of the internal gadgets.

Now we define basic ways to join gadgets together. Let G_1, \dots, G_k be some gadgets with start states s^{G_1}, \dots, s^{G_k} and target states t^{G_1}, \dots, t^{G_k} , respectively.

• Sequence Gadget.

For each $i = 1, \dots, k-1$, we identify the target state t^{G_i} with the start state $s^{G_{i+1}}$. Then s^{G_1} and t^{G_k} are respectively the start and target states of the Sequence Gadget. We represent this construction by writing $I_1 \dots I_k$.

Complete computations for this gadget are concatenations of complete computations for the internal gadgets.

For example, for $k = 3$ and $m = 3$, the Sequence Gadget $\text{SELECT}(U) \text{ SELECT}(V) U = V$ is shown in Fig. 5. It has the property that every complete computation has length $3m = 9$ and is a concatenation of complete computations for the three gadgets, and the final configuration

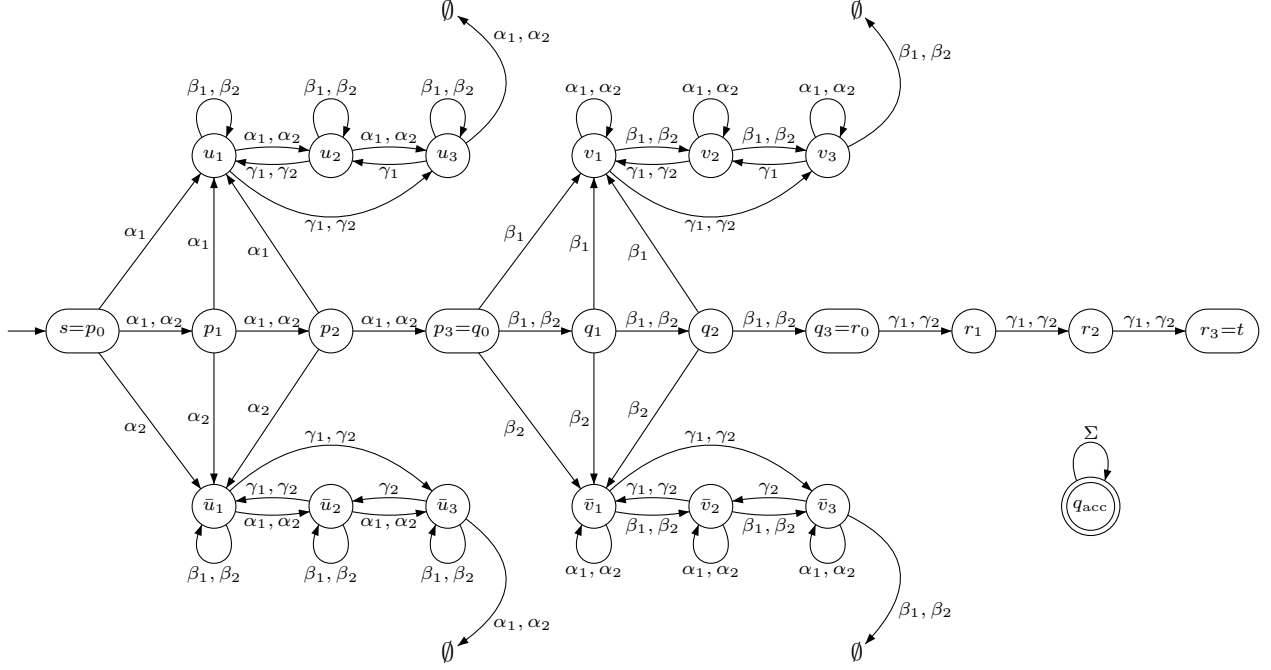


Figure 5: The complete NFA of the Sequence Gadget $\text{SELECT}(U) \text{ SELECT}(V) U = V$. All omitted transitions go to q_{acc} . The states p_i, q_i, r_i and letters $\alpha_i, \beta_i, \gamma_i$ belong to the three gadgets, respectively, that is: $p_i = p_i^{\text{SELECT}(U)}$, $\alpha_i = \alpha_i^{\text{SELECT}(U)}$, $q_i = p_i^{\text{SELECT}(V)}$, $\beta_i = \alpha_i^{\text{SELECT}(V)}$, $r_i = p_i^{U=V}$, $\gamma_i = \alpha_i^{U=V}$.

C' is such that $U(C') = V(C')$. There exists a complete computation for every possible value of both variables, and longer computations are not proper.

• **Choose Gadget.**

This gadget allows to select one of the given gadgets non-deterministically and perform a computation for it. We add a fresh start state s and k unique letters $\alpha_1, \dots, \alpha_k$. The letter α_i maps s to $\{s^{G_i}\}$, maps control flow states from the gadgets to $\{q_{\text{acc}}\}$, and fixes variables states. All target states t^{G_i} are identified into the target state t of the Choose Gadget. We represent this construction by:

choose:

I_1

or:

\dots

or:

I_k

end choose

Note that for this gadget there may exist complete computations of different lengths, even for the same initial configuration. Nevertheless, there exists an upper bound on the length such that every computation longer than this bound is not proper (which is 1 plus the maximum from the bounds for the internal gadgets).

- **If-Else Gadget.**

It joins three gadgets, where G_1 is either an Equality Gadget or an Inequality Gadget, and G_2 and G_3 are any gadgets. This construction is represented by:

if G_1 **then** G_2 **else** G_3 **end if**

The third gadget G_3 may be empty, and then we omit the *else* part. The gadget is implemented as follows:

choose:

G_1

G_2

or:

$\neg G_1$

G_3

end choose

where by $\neg G_1$ we represent the negated version of G_1 , i.e. the corresponding Inequality Gadget if G_1 is an Equality Gadget, and the corresponding Equality Gadget if G_1 is an Inequality Gadget. Thus, a complete computation contains first a non-deterministic guess whether the variables in G_1 are equal, which is then verified, and then there is a complete computation for one of the two gadgets.

- **While Gadget.**

This is easily constructed using *If-Else*, where G_3 is empty, and the target state of G_2 is identified with the start state of the If-Else Gadget, which also becomes the start state of the While Gadget. The target state t of the While Gadget is the target state of $\neg G_1$.

- **While-True Gadget.**

A special variant of the previous one is with **true** condition instead of G_1 . It therefore degenerates to the gadget G_2 with the target and the start states identified. This is the only gadget that has outgoing transitions from its target state. In contrast to the other constructions, there may exist infinite number of complete computations for this gadget (which are concatenations of shorter complete computations – iterations of the loop). We will be using it only as the last gadget in a Sequence Gadget, so there will be no possibility to leave this gadget.

4.1.3 Compound gadgets as programs

Now we define the rest of the required gadgets by writing suitable programs, which represent these gadgets. In each program we also define *external* and *internal* variables. Internal variables are always fresh and unique and only this gadget operates on them, whereas the external variables may be shared.

- **Addition Gadget.**

The addition of two variables is denoted by $W \leftarrow U + V$ and defined by Alg. 1. It is basically a Sequence Gadget with two Assignment Gadgets and a While Gadget. It operates on four different variables. Variables U and V must be valid in an initial configuration, while W and X do not have to be.

The semantic properties are such that, for an initial configuration C , if $U(C) + V(C) <$

$2^m - 1$ then there exists a complete computation to a final configuration C' such that $W(C') = U(C) + V(C)$. If the result $U(C) + V(C)$ is larger than $2^m - 1$, then the gadget does not admit a complete computation, because at some point we would have to go through the Incrementation Gadget in line 5 when the value of U is equal to $2^m - 1$. Furthermore, every computation long enough is not proper.

Algorithm 1 Addition Gadget $W \leftarrow U + V$.

External variables: W, U, V

Internal variable: X

1: $W \leftarrow U$	▷ Assignment Gadget
2: $X \leftarrow 0$	▷ Assignment Gadget
3: while $X \neq V$ do	▷ While Gadget with Inequality Gadget
4: $X++$	▷ Incrementation Gadget
5: $W++$	▷ Incrementation Gadget
6: end while	

• **Multiplication Gadget.**

The multiplication of two variables is denoted by $W \leftarrow U \cdot V$ and defined by Alg. 2. It uses the Addition Gadget to add $V(C)$ times the value $U(C)$ to the output variable W , where C is an initial configuration. As in the Addition Gadget, if the result $U(C) \cdot V(C)$ is larger than $2^m - 1$, then this gadget does not admit a complete computation.

Algorithm 2 Multiplication Gadget $U \cdot V$.

External variables: U, V, W

Internal variables: X, W'

1: $W \leftarrow 0$	
2: $X \leftarrow 0$	
3: while $X \neq V$ do	
4: $X++$	
5: $W' \leftarrow W + U$	▷ Addition Gadget
6: $W \leftarrow W'$	
7: end while	

• **Primality Gadget.**

The primality testing of the value of a variable P is defined by Alg. 3. A complete computation is possible if and only if P is prime. We test whether P is prime by enumerating all pairs of integers $2 \leq X, Y < P$ and checking whether $X \cdot Y = P$.

We will also need the negated version of this gadget (testing if P is not prime), which can be implemented by selecting arbitrary values for two integers X, Y , and verifying that the values are not from $\{0, 1, P\}$ and that $X \cdot Y = P$. From now, the Primality Gadget or its negated version can be also used in an If-Else or a While gadget.

• **Prime Number Gadget.**

This gadget assigns the $U(C)$ 'th prime number to a variable P , where C is an initial configuration. It enumerates all integers $P = 2, 3, 4, \dots$, checks which are prime, and counts

Algorithm 3 Primality Gadget.

External variable: P **Internal variables:** X, Y, Z

```
1:  $X \leftarrow 2$ 
2: while  $X \neq P$  do
3:    $Y \leftarrow 2$ 
4:   while  $Y \neq P$  do
5:      $Y++$ 
6:      $Z \leftarrow X \cdot Y$ 
7:      $Z \neq P$ 
8:   end while
9:    $X++$ 
10: end while
```

the prime ones in a separate variable X . When P becomes the $U(C)$ 'th prime number, the computation ends. The gadget does not admit a complete computation when the $U(C)$ 'th prime number exceeds $2^m - 1$, or when $U(C) = 0$.

Algorithm 4 Prime Number Gadget $P \leftarrow U$ 'th prime number.

External variables: P, U **Internal variable:** X

```
1:  $P \leftarrow 2$ 
2:  $X \leftarrow 1$ 
3: while  $X \neq U$  do
4:    $P++$ 
5:   if  $P$  is prime then  $X++$  end if
6: end while
```

The number of states in our NFAs of the gadgets is in $O(dm)$, where d is the length of the program measured by the number of basic gadgets instantiated plus the number of variables, and its alphabet has size $\Theta(d)$. For a given program (and integer m), we can easily construct the corresponding NFA in polynomial time.

For every of the defined gadgets, except the While-True Gadget (or gadgets containing it), there exists an upper bound on the length of any complete computation, and every longer computation is not proper. This bound is at most exponential in the size of the gadget, i.e. $O(2^{dm})$, because proper computations cannot repeat the same configuration.

4.2 An NFA with a large minimal universality length

Our first application is to show a lower bound on the maximum minimal universality length. Its idea will be further extended to show the lower bound for computational complexity.

Alg. 5 gives the program encoding our NFA, and Fig. 6 shows the control flow states of this NFA. The numbers in the brackets [] at the right denote the length of a word of

Algorithm 5 Large minimal universality length.

Variables: X, Y

1: SELECT(Y)	$\triangleright [m]$
2: $X \leftarrow 0$	$\triangleright [1]$
3: while true do	
4: choose:	$\triangleright [1]$
5: $X = Y$	$\triangleright [m]$
6: $X \leftarrow 0$	$\triangleright [1]$
7: [final state] WAIT($m + 1$)	$\triangleright [m + 1]$
8: or:	
9: $X \neq Y$	$\triangleright [m + 1]$
10: $X++$	$\triangleright [m + 1]$
11: end choose	
12: end while	

a complete computation for the gadget (or a part of it) in the current line. These are the lengths of paths from the start state to the target state corresponding to this line. In line 7, the annotation [final state] indicates that the start control flow state of this Waiting Gadget is final, so the NFA has two final states.

The idea of the program is as follows: At the beginning we choose an arbitrary value for Y , and then in an infinite loop we increment X modulo $Y + 1$. Every iteration (complete computation of the Choose Gadget) of the loop takes the same number of letters ($2m + 3$), hence given a length we know that we must perform $d - 1$ complete iterations and end in the d 'th iteration, for some d . A proper computation of this length can avoid the final state in line 7 only in the iterations where the value of X does not equal the value of Y . This is the case for every length smaller than $\text{lcm}(1, 2, \dots, 2^m) \cdot O(m)$, as we can always select Y such that $Y + 1$ does not divide $d + 1$.

Now we have a very technical part, which is to calculate the size of the NFA of Alg. 5 and ensure that the lengths are correct (see also Fig. 6).

The lengths of proper computations. First, in line 1 any value for the variable Y can be chosen, and this takes exactly m letters of the Selection Gadget. In line 2 the Assignment Gadget takes 1 letter. The While-True Gadget only means that the start and target states of the internal Choose Gadget are identified. In line 4 a non-deterministic branch either to line 5 or 9 is performed, which takes 1 letter. Line 5 contains an Equality Gadget, which takes m letters. In line 6 we have the second Assignment Gadget, which takes 1 letter. In line 7 there is a Waiting Gadget that takes $m + 1$ letters; there is also indicated that the start control flow state, which is also the target state of the second Assignment Gadget, is final. In line 9 the Inequality Gadget takes $m + 1$ letters, and the Incrementation Gadget in line 10 also takes $m + 1$ letters. Summarizing, each complete iteration of the while loop (thus a complete computation of the Choose Gadget) takes exactly $2m + 3$ letters, regardless of the non-deterministic choice.

The size of the NFA. We count the number of states and the number of letters in the NFA. We have two variables with $2m$ states each. The Selection Gadget in line 1 has m

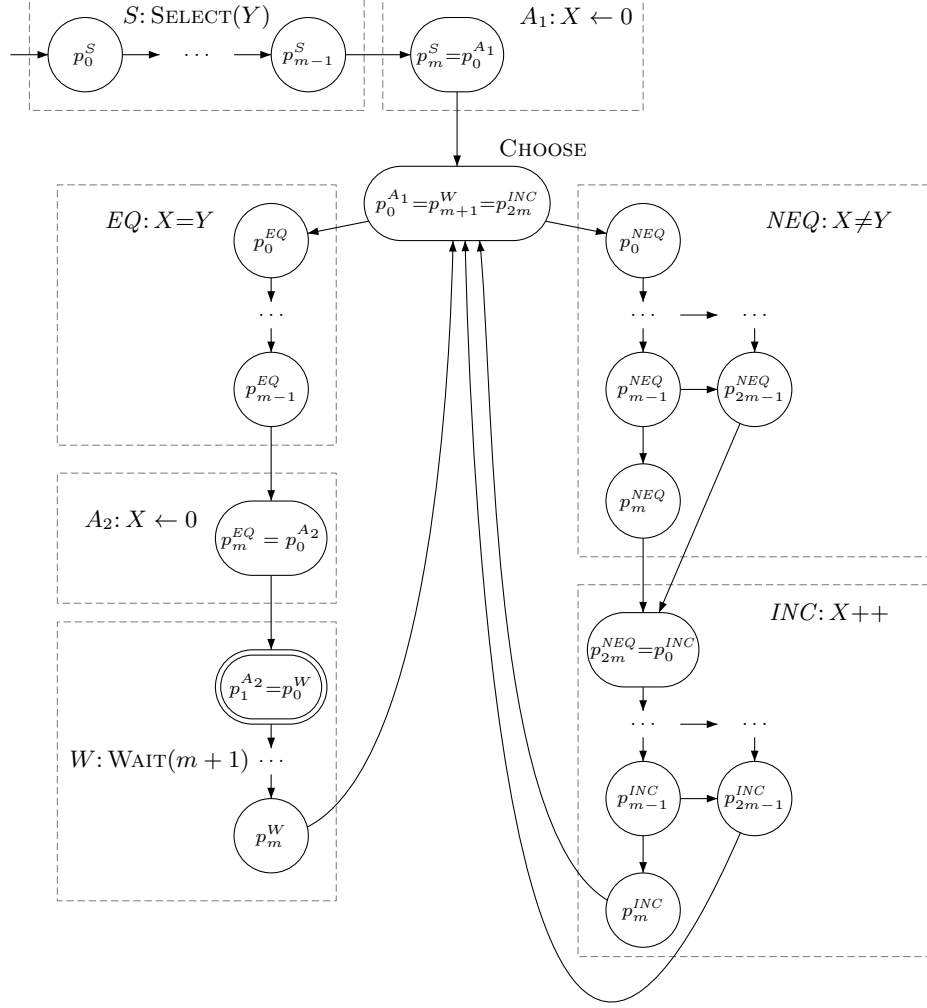


Figure 6: The control flow states in the NFA of Alg. 5 with their membership in particular gadgets. Omitted transitions go to q_{acc} .

states (excluding the target state) and it has 2 letters. The Assignment Gadget in line 2 adds just 1 control flow state and 1 letter. The while loop does not introduce any new states nor letters, but just the target and the start states of the internal Choose Gadget are identified. In line 4 the Choose Gadget adds 1 state and 2 letters for branching. In line 5 the Equality Gadget adds m control flow states and 2 letters. In line 6 the Assignment Gadget adds 1 state and 1 letter. In line 7 the Waiting Gadget adds $m + 1$ states and 1 letter. In line 9 the Inequality Gadget adds $2m$ control flow states and 3 letters. Also, in line 10 the Incrementation Gadget adds $2m$ control flow states and 3 letters. The target state of the Waiting Gadget and of the Incrementation Gadget were identified with the start state of the Choose Gadget, thus we have already counted it. Finally, there is the state q_{acc} . Summarizing, the NFA has $4m + m + 1 + 1 + m + 1 + (m + 1) + 2m + 2m + 1 = 11m + 5$ states and $2 + 1 + 2 + 2 + 1 + 1 + 3 + 3 = 15$ letters.

Now we prove that the NFA has a long minimal universality length.

Lemma 18. *For a given m , the NFA of Alg. 5 has minimal universality length*

$$\text{lcm}(1, 2, \dots, 2^m)(2m + 3).$$

The number of states of this NFA is $11m + 5$ and the size of its alphabet is 15.

Proof. There are two final states in the NFA: q_{acc} and the start state of the Waiting Gadget indicated in line 7. Thus, every non-accepted word must be a proper computation and such that the current configuration does not contain this final state from line 7.

First we show that there exists a non-accepted word for every length smaller than $\text{lcm}(1, 2, \dots, 2^m)(2m + 3)$. Observe that for every length there exists a word w being a proper computation of the program. Moreover, there exists such a word w for every value of Y selected in line 1 (if $|w| \geq m$). Consider such a word w for some value of Y . From the beginning of the program to (a configuration with) the final state a proper computation takes at least $2m + 3$ letters, so if $|w| < 2m + 3$ then w is not accepted. Since every iteration of the while loop takes exactly $2m + 3$ letters, if $|w|$ is not divisible by $2m + 3$, then the final state in line 7 cannot be active after reading the whole w . Suppose that $|w|$ of length at least $2m + 3$ is divisible by $2m + 3$ and let $d = |w|/(2m + 3) \geq 1$. Then the computation of w performs exactly $d - 1$ iterations of the while loop and ends in the d 'th iteration. Depending on the value of Y , after reading w the active control flow state may be either the final state in line 7 or the start state of the Incrementation Gadget in line 10. If $|w| < \text{lcm}(1, 2, \dots, 2^m)(2m + 3)$, then $d < \text{lcm}(1, 2, \dots, 2^m)$. So we can find a value for Y ($1 \leq Y \leq 2^m - 1$) such that d is not divisible by $Y + 1$. The computation w is such that there are Y iterations in which X is incremented and one iteration in which X is reset to 0, which is repeated during the whole computation. So at the beginning of an i 'th iteration the value of X is equal to $(i - 1) \bmod (Y + 1)$. Since d is not divisible by $Y + 1$, at the beginning of the d 'th iteration we have $X = (d - 1) \bmod (Y + 1) \neq Y$, and so the computation finishes at the first state of the Incrementation Gadget in line 10.

It remains to show that every word w of length $\text{lcm}(1, 2, \dots, 2^m)(2m + 3)$ is accepted. Suppose, contrary to what we want, that w is not accepted, which means by definition that it must be a proper computation. Hence, at the beginning it chooses some value for Y , and then performs iterations in the while loop. Since every iteration takes exactly $2m + 3$ letters, exactly $\text{lcm}(1, 2, \dots, 2^m) - 1$ complete iterations must be performed, and the computation ends in the $\text{lcm}(1, 2, \dots, 2^m)$ 'th iteration. Regardless of the selected value for Y , $\text{lcm}(1, 2, \dots, 2^m)$ is divisible by $Y + 1$. Hence, at the beginning of the last iteration we have $X = Y$, and there remain $m + 2$ letters to read. Now, if w chooses the second branch, the computation cannot pass the test in line 9, since for this Inequality Gadget there is no complete computation and every computation longer than $m + 1$ is not proper. So w must choose the first branch, which after $m + 2$ letters results in a configuration with the final state in line 7. \square

It remains to calculate the bound in terms of the number of states.

Theorem 19. *For a 15-letter alphabet, the minimal universality length can be as large as*

$$e^{2^{n/11}(1+o(1))}.$$

Proof. We have

$$\text{lcm}(1, 2, \dots, 2^m)(2m + 3) = (2m + 3) \cdot \exp(2^m(1 + o(1))) = \exp(2^m(1 + o(1))).$$

For a sufficiently large number of states n we can construct the NFA from Alg. 5 for $m = \lfloor (n - 5)/11 \rfloor$, and add $n - m$ unused states. Then we obtain

$$\exp(2^m(1 + o(1))) = \exp(2^{(n-5)/11}(1 + o(1))) = \exp(2^{n/11}(1 + o(1))).$$

□

Remark 20. The constants in Lemma 18 and Theorem 19 are not optimal and could be further optimized. The witness NFA was obtained directly from the construction, whereas, for example, some control flow states could be shared and the number of letters reduced.

4.3 Checking the properties of the computation length

In Subsection 4.2 we have constructed an NFA for which every word encoding a proper computation must be accepted or can be not accepted depending on its length, namely, it is always accepted if the length is divisible by $\text{lcm}(1, 2, \dots, 2^m)(2m + 3)$. We generalize this idea, so that we will be able to express more complex properties about the length for which all words must be accepted.

• Delaying Gadget.

The first new ingredient is the Delaying Gadget $\text{DELAY}(D)$, where D is a fixed integer ≥ 0 . This is a stronger version of the Waiting Gadget. It has the advantage that it can wait for an exponential number of letters in the size of the gadget.

Let $D \geq 1$ be a fixed integer and let $m' \geq 1$ be an integer such that $D \leq 2^{m'} - 1$. The program uses two variables X and Y , both of width m' . The width m' can be different from m , which is used for all variables in all other gadgets, but this is allowed, since the variables X, Y are always internal and will never be shared. The Delaying Gadget is defined by Alg. 6 and denoted by $\text{DELAY}(D)$.

The length of every complete computation is precisely $2 + D(1 + 2(m' + 1)) + 1 + m'$, and every longer computation is not proper. Let $T(D)$ denote this exact length when we take m' to be the minimum possible, so $m' = \lceil \log_2(D + 1) \rceil$. Thus, $T(D) \in \Theta(D \log D)$.

• Parallel Gadget.

As we noted, because of the Choose Gadgets, complete computations may have different lengths. This is an obstacle that would make very difficult or impossible to further rely on the exact length $|w|$ of a proper computation, basing on which we would like to force whether w must be accepted or not. Therefore, we need a possibility to ensure that all complete computations have a fixed known length, and furthermore, that there are no longer proper computations.

Algorithm 6 Delaying Gadget.

Internal variables: X, Y of width m'

```
1:  $X \leftarrow 0$   $\triangleright [1]$ 
2:  $Y \leftarrow D$   $\triangleright [1]$ 
3: while  $X \neq Y$  do  $\triangleright [1 + (m' + 1)]$ 
4:    $X++$   $\triangleright [m' + 1]$ 
5: end while  $\triangleright [m']$  (here is the Equality Gadget)
```

The idea for doing this is to implement computation in parallel. A given gadget for which there may exist complete computations of different lengths is computed in parallel with a Delaying Gadget. When the computation is completed for the given gadget, it still must wait in its target state until the computation is also finished for the Delaying Gadget. In this way, as long as complete computations for the Delaying Gadget are always longer than those for the given gadget (we can ensure this by choosing D), complete computations for the joint construction will have a fixed length.

We construct the Parallel Gadget as follows. Let G_1 be a given gadget for which there exists an upper bound L such that every longer computation is not proper. We assume that all outgoing transitions from its target state go to q_{acc} (only the While-True Gadget violates this). The second gadget G_2 will be the Delaying Gadget with a D such that $T(D) \geq L$. The Parallel Gadget is illustrated in Fig. 7 and defined as follows.

- The start states s^{G_1} and s^{G_2} are identified with the start state s of the Parallel Gadget. The other states from both gadgets are separate. The target state t of the Parallel Gadget is a fresh state.
- The alphabet consists of the following:
 - For every pair of letters $\alpha_i^{G_1}$ from G_1 and $\alpha_j^{G_2}$ from G_2 , there is the letter $\beta_{i,j}$ that acts on the states from G_1 as $\alpha_i^{G_1}$ and on the states from G_2 as $\alpha_j^{G_2}$.
 - For every letter $\alpha_j^{G_2}$ from G_2 , there is the letter γ_j that acts on the states from G_2 as $\alpha_j^{G_2}$, fixes t^{G_1} and the variable states in G_1 , and maps all the other control flow states of G_1 to q_{acc} .
 - There is the letter τ that maps both t^{G_1} and t^{G_2} to $\{t\}$, fixes all variable states, and maps all the other control flow states to q_{acc} .
 - All letters map t to q_{acc} .

The Parallel Gadget works as follows: First, letters $\beta_{i,j}$ must be used, which encodes proper computations for both gadgets. When t^{G_1} becomes active, the computation for the G_1 is completed and $\beta_{i,j}$ cannot be used anymore, but the letters γ_i can be applied. When letters γ_i are applied, G_1 waits until the Delaying Gadget has finished the computation. Finally, when and only when both t^{G_1} and t^{G_2} are active, we can and must apply τ , which finishes a complete computation of the Parallel Gadget and the target state becomes active.

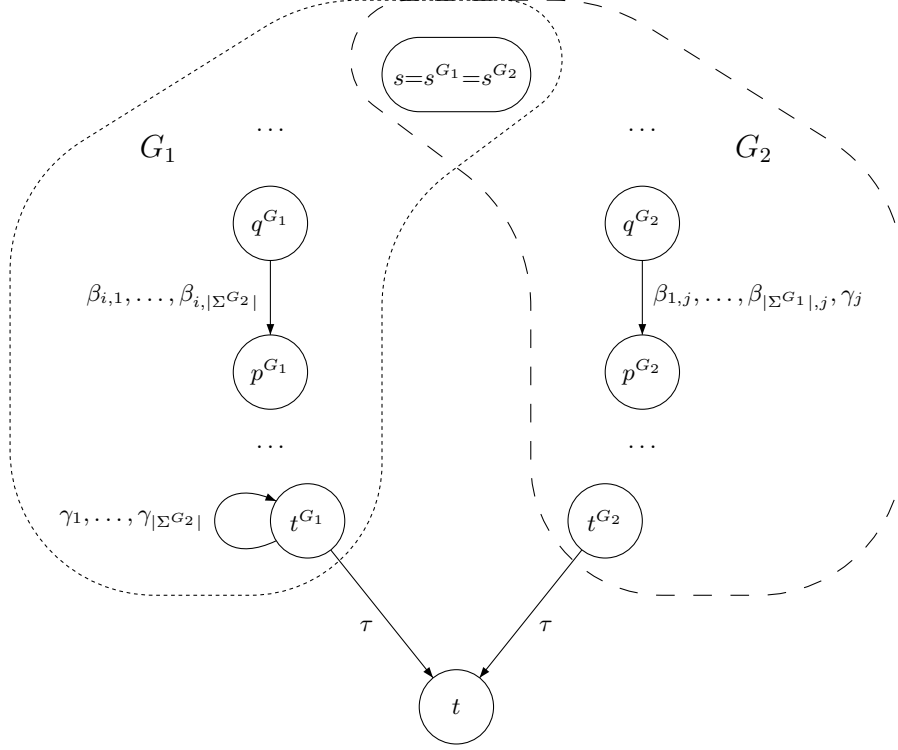


Figure 7: The Parallel Gadget, where in G_1 a letter $\alpha_i^{G_1}$ maps a state q^{G_1} to p^{G_1} , and in G_2 a letter $\alpha_j^{G_2}$ maps a state q^{G_2} to p^{G_2} .

The length of all complete computations of the Parallel Gadget is therefore equal to $T(D) + 1$ (the length for the Delaying Gadget plus the letter τ).

Note that we can construct the Parallel Gadget in polynomial time. If n is the number of states in G_1 , we know that its complete computations cannot be longer than 2^n (so $L \leq 2^n$). Thus we can simply set $D = 2^n$ and $m' = n + 1$, which for sure is such that $T(D) > L$. The number of states and letters of the Parallel Gadget is still polynomial in the size of G_1 .

4.3.1 Divisibility

We are going to test whether $|w|$ satisfies some properties, in particular, whether (a function of it) is divisible by some integers. This extends the idea from Alg. 5, which just verifies whether $|w|/r$, for some constant r , is not divisible by some integer from 2 to 2^m .

We define the *divisibility program* shown in Alg. 7. It is constructed for given numbers k and m , and a *verifying procedure*, which is any gadget satisfying the following properties:

- Its every complete computation does not exceed a bound L (exponential in its size), and longer computations are not proper. All outgoing transitions from its target state go to q_{acc} .
- It may use, but does not modify the variables $X_1, \dots, X_k, X'_1, \dots, X'_k$, i.e. after its

Algorithm 7 Divisibility program.

Variables: $X_1, \dots, X_k, X'_1, \dots, X'_k$

```
1: SELECT( $X_1$ ), ..., SELECT( $X_k$ ).  $\triangleright [km]$ 
2:  $X'_1 \leftarrow 0, \dots, X'_k \leftarrow 0$   $\triangleright [k]$ 
3: while true do
4:   choose:  $\triangleright [1]$ 
5:      $\left\{ \begin{array}{l} \text{execute} \\ \text{in parallel} \end{array} \right. \left\{ \begin{array}{l} \text{DELAY}(D) \\ \text{Verifying procedure} \end{array} \right.$   $\triangleright [T(D) + 1]$ 
6:   or:
7:     DELAY( $D$ )  $\triangleright [T(D)]$ 
8:     [final state] WAIT(1)  $\triangleright [1]$ 
9:   end choose
10:  for  $i = 1, \dots, k$  do
11:     $X'_i++$   $\triangleright [m + 1]$ 
12:    if  $X'_i = X_i$  then  $\triangleright [m + 1 \text{ if } X'_i = X_i, \text{ and } m + 2 \text{ otherwise}]$ 
13:       $X'_i \leftarrow 0$   $\triangleright [1]$ 
14:    end if
15:  end for
16: end while
```

every complete computation their values in the final configuration are the same as in the initial configuration.

- It may have internal variables, which do not have to be valid in an initial configuration and the existence of complete computations do not depend on their setting in an initial configuration.
- It does not contain final states.

As before, the numbers in the brackets $[]$ at the right denote the lengths of a word of a complete computation for the current line. We have an infinite while loop, which consists of two parts. In the first part, a non-deterministic choice is made (line 4): either to run the verifying procedure or to wait. For the verifying procedure (line 5) we use the Parallel Gadget; this ensures that this part finishes after exactly $T(D) + 1 > L$ letters. In the waiting case (line 7) we use the Delaying Gadget with the same value of D as that in the Parallel Gadget. Then there is a single final state (line 8). In the second part (lines 10–15) every variable X'_i counts the number of iterations of the while loop modulo X_i . The *for* loop denotes that the body is instantiated for every i (so it is a Sequence Gadget). Every complete computation of the second part (lines 10–15) has exactly $(2m + 3)k$ letters.

The idea is that, for certain lengths, every proper computation must end with a configuration with the non-final control flow states in line 5 (these are precisely the two target states, of the verifying procedure and of the Delaying Gadget) or in the final state in line 8. However, for the first option it must first succeed with the verifying procedure when $X'_i = \ell' \bmod X_i$. In other words, for some selection of the values for X_1, \dots, X_k , there must exist a complete

computation of the verifying procedure from an initial configuration with these values for X_i and $X'_i = \ell' \bmod X_i$. In this way the verifying procedure can check divisibility properties of ℓ by X_i .

Lemma 21. *Consider Alg. 7 for some k, m , and a verifying procedure. There exist integers $r_1 \geq 1$ and $r_2 \geq 1$ such that Alg. 7 accepts all words of length ℓ if and only if there exist an integer ℓ' such that:*

- $\ell = r_1 \cdot \ell' + r_2$, and
- for every initial configuration C of the verifying procedure where variables X_1, \dots, X_k are valid and $X'_i(C) = \ell' \bmod X_i(C)$ for all $1 \leq i \leq k$, there does not exist a complete computation for the verifying procedure.

Proof. Let $r_1 = 1 + (T(D) + 1) + (2m + 3)k$, which is the length complete computations of one iteration of the while loop (lines 4–15), and let $r_2 = km + k + 1 + T(D)$, which is the length of a proper computation from a configuration with the initial state (start state in line 1) to a configuration with the final state in line 8 that does not contain a complete computation of the while loop. These values depend only on the algorithm, so on k, m , and the verifying procedure.

Consider a length ℓ that is not expressible as $r_1 \cdot \ell' + r_2$. There exists a proper computation w of length ℓ that every time chooses the second branch (lines 7–8). It is not accepted, since the obtained configuration is proper and cannot contain the final state in line 8.

Consider a word w of length $r_1 \cdot \ell' + r_2$. If w is not accepted, then it must encode a proper computation. Then it must perform exactly ℓ' complete iterations of the while loop, and the last control flow state is either the final state in line 8 or the non-final states in line 5. We know that at the beginning of the last (incomplete) iteration we have $X'_i = \ell' \bmod X_i$ for all i . Now, if there exists a selection of the values for X_1, \dots, X_k such that there exists a complete computation of the verifying procedure for the initial configurations with these values of X_i and X'_i , then w can select these values of X_i and choose the first branch in the last iteration. Thus there exists a proper computation w that ends with a configuration with the non-final control states in line 5. Otherwise, since in the last iteration entering the verifying procedure results in a non-proper configuration after $T(D)$ letters, regardless of the choice for X_1, \dots, X_k , w must choose the second branch in the last iteration of the while loop, which results in the final state in line 8 in the last configuration. \square

• Verifying procedure.

We develop a method for verifying the properties of the length of a word w in a flexible way. Consider a logical formula $\varphi(\ell')$ of the following form:

$$\exists_{X_1, \dots, X_k \in \{0, \dots, 2^m - 1\}} \psi(X_1, \dots, X_k).$$

Formula ψ is an arbitrary logical formula that uses \wedge, \vee , and whose simple propositions are of the following possible forms:

1. $(X_i = c)$, where $c \in \{0, \dots, 2^m - 1\}$,

2. $(X_h = X_i + X_j)$,
3. $(X_h = X_i \cdot X_j)$,
4. X_i is prime,
5. X_i is the X_j 'th prime number,
6. $(X_i \mid \ell')$ or $(X_i \nmid \ell')$,

where X_i, X_j, X_h are some variables from $\{X_1, \dots, X_k\}$.¹

We construct the gadget $\text{VERIFY}(\psi)$ for verifying ψ that will be our verifying procedure. It is built using Sequence Gadgets for conjunctions, Choose Gadgets for disjunctions, and other appropriate gadgets for (1)–(6). In more detail, $\text{VERIFY}(\psi)$ is defined recursively as follows.

- For (1), when ψ is $(X_i = c)$:
We add a fresh unique variable C . First we use (in a Sequence Gadget) the Assignment Gadget $C \leftarrow c$, and then we use either the Equality Gadget $X_i = c$.
- For (2,3), when ψ is $(X_i + X_j = X_h)$ or $(X_i \cdot X_j = X_h)$:
We add a fresh unique variable C . First we assign $C \leftarrow X_i + X_j$ by the Addition Gadget or $C \leftarrow X_i \cdot X_j$ by the Multiplication Gadget, and then we use the Equality Gadget.
- For (4), when ψ is $(X_i \text{ is prime})$:
We use the Primality Gadget.
- For (5), when ψ is $(X_i \text{ is the } X_j \text{'th prime number})$:
We add a fresh unique variable C . First we compute the X_j 'th prime number using the Prime Number Gadget, and then we use the Equality Gadget.
- For (6), when ψ is $(X_i \mid \ell')$ or $(X_i \nmid \ell')$: We use either the Equality Gadget $X'_i = 0$ or the Inequality Gadget $X'_i \neq 0$.
- When $\psi = \psi_1 \wedge \dots \wedge \psi_h$:
It is the Sequence Gadget joining the verifying gadgets for the subformulas, i.e.
 $\text{VERIFY}(\psi_1) \dots \text{VERIFY}(\psi_h)$
- When $\psi = \psi_1 \vee \dots \vee \psi_h$:
It is the Choose Gadget joining the verifying gadgets for the subformulas, i.e.
choose: $\text{VERIFY}(\psi_1)$ **or:** \dots **or:** $\text{VERIFY}(\psi_h)$ **end choose**

¹We do not require that, but this form could be extended for example by negations. In this case we could transform ψ by De Morgan's laws into an equivalent formula with negations before simple propositions, and increase precision for proper verification of inequalities that may appear from (2), (3), and (5).

Note that for (2), (3), and (5) if the value of the operation exceeds $2^m - 1$ then for sure this simple proposition is false, and a complete computation does not exist for the gadget computing this value.

Alg. 8 shows the program describing $\text{VERIFY}(\psi)$ for an example formula ψ .

Algorithm 8 The verifying procedure for $k = 3$ and

$$\psi = (X_1 = X_2 + X_3) \vee ((X_1 \text{ is the } X_2\text{'th prime number}) \wedge (X_2 \nmid \ell')).$$

External variables: $X_1, \dots, X_3, X'_1, \dots, X'_3$

Internal variables: C_1, C_2

```

1: choose:
2:    $C_1 \leftarrow X_2 + X_3$ 
3:    $X_1 = C_1$ 
4: or:
5:    $C_2 \leftarrow X_2\text{'th prime number}$ 
6:    $X_1 = C_2$ 
7:    $X'_2 \neq 0$ 
8: end choose

```

It follows that, for an initial configuration C for $\text{VERIFY}(\psi)$ with valid variables X_1, \dots, X_k and where $X'_i(C) = \ell' \bmod X_i(C)$, there exists a complete computation if and only if $\psi(\ell')$ is satisfied. Furthermore, the values of the variables $X_1, \dots, X_k, X'_1, \dots, X'_k$ remain the same in the final configuration. There exists an exponential upper bound L for the length of all complete computations (although they may have different lengths due to disjunctions), and longer computations are always not proper. The internal variables are always initialized, so they can be arbitrary in an initial configuration and complete computations do not depend on their setting. Hence, $\text{VERIFY}(\psi)$ satisfies the requirements for being the verifying procedure and we will use it in Alg. 7.

By Lemma 21, for every length ℓ , there exists a word w of length ℓ that is not accepted by the NFA if and only if $\ell = r_1 \cdot \ell' + r_2$ and $\varphi(\ell')$ is satisfied. Since the NFA of Alg. 7 and the NFA of the verifying procedure can be constructed in polynomial time, the final NFA also can be. Its size is polynomial and the constants r_1 and r_2 are at most exponential.

4.4 Reduction

We reduce from the canonical NEXPTIME-complete problem: given a non-deterministic Turing machine N with s states, does it accept the empty input after at most 2^s steps? Without loss of generality, N is a one-tape machine using the binary alphabet $\{0, 1\}$. Furthermore, N can be modified so that, upon accepting, it clears the tape, moves the head to the leftmost cell, and waits there while being still in the (unique) accepting state q_f . Because we are interested in executing at most 2^s steps, we can also bound the length of the tape. Therefore, we can assume that the tape of length 2^s is initially filled with 2^s zeroes, the head is on the leftmost cell and the machine is in the unique initial state q_0 . The goal is

then to check if there exists a computation such that, after exactly 2^s steps, the head is on the leftmost cells and the machine is in the unique accepting state. We work with such a formulation from now on.

A computation of N can be represented by an $2^s \times 2^s$ table, where the i 'th row of the table describes the content of the tape after i steps of the computation. Every cell (r, c) ($0 \leq r, c \leq 2^s - 1$) of the table stores a symbol from $\{0, 1\}$ and, possibly, a state of N . Therefore, to check if there exists an accepting computation we need to check if it is possible to fill the table so that it represents subsequent steps of such a computation. We will construct an NFA M such that every choice of what is stored in the cells of the table corresponds to a number $\ell' \leq 2^{2^{\text{poly}(s)}}$ and there exists a word $w \in \Sigma^\ell$ not accepted by M if and only if ℓ' does not describe an accepting computation.

Let Q be the set of states of N , and $q_0, q_f \in Q$ be its starting and accepting states, respectively. We want to check if it is possible to choose an element $t(r, c) \in (Q \cup \{\text{nil}\}) \times \{0, 1\}$ for every cell (r, c) of an $2^s \times 2^s$ table, so that the whole table describes an accepting computation of M . The elements are identified with integers by a function $f: (Q \cup \{\text{nil}\}) \times \{0, 1\} \rightarrow \{0, \dots, z-1\}$, where $z = 2s+2$. Therefore, we want to choose a number $t(r, c)$ from $\{0, \dots, z-1\}$ for every (r, c) . We encode all these choices in one non-negative integer ℓ' as follows. Let $p(k)$ denote the k 'th prime number. For every (r, c) , we reserve z prime numbers $p((2^s r + c)z + 1), \dots, p((2^s r + c)z + z)$. Each of these primes represents a possible choice for $t(r, c)$. Then, we select the remainder of ℓ' modulo $p((2^s r + c)z + i)$ to be zero if $t(r, c) = i$; otherwise, we select any non-zero remainder. Then, by the Chinese remainder theorem, any choice of all the elements can be represented by a non-negative integer $\ell' \leq p(1) \cdots p(2^{2^s} z)$. In the other direction, every non-negative integer ℓ' represents such a choice as long as, for all (r, c) , ℓ' is divisible by exactly one of the z primes reserved for (r, c) . Hence, we now focus on constructing a logical formula $\varphi(\ell')$ that can be used to check if indeed ℓ' has such a property and, if so, whether the represented choice describes an accepting computation of N .

We construct $\varphi(\ell')$ so that it is satisfied exactly when at least one of the following situations occur:

1. For some (r, c) and $1 \leq i < j \leq z$, ℓ' is divisible by both $p((2^s r + c)z + i)$ and $p((2^s r + c)z + j)$.
2. For some (r, c) , ℓ' is not divisible by $p((2^s r + c)z + i)$, for every $i = 1, \dots, z$.
3. ℓ' is not divisible by $p(f(q_0, 0))$.
4. For some $c \in \{1, \dots, 2^s - 1\}$, ℓ' is not divisible by $p(c \cdot z + f(\text{nil}, 0))$.
5. ℓ' is not divisible by $p(2^s(2^s - 1)z + f(q_f, 0))$.
6. For some $r \in \{1, \dots, 2^s - 1\}$ and $c \in \{2, \dots, 2^s - 1\}$, the 2×3 window of cells with the lower-right corner at (r, c) is not legal for the transitions of N .

Before describing how to construct such a formula, we elaborate on the last condition. A 2×3 window of cells is legal if it is consistent with the transition function of N . We avoid giving a tedious precise definition and only specify that $W \subseteq \{0, \dots, z-1\}^6$ is the set of six-tuples of numbers corresponding to elements chosen for the cells of such a legal 2×3 window; W can be constructed in polynomial time given the transition function of N . Therefore, the last condition can be written in more detail as follows.

- 6'. For some $r \in \{1, \dots, 2^s - 1\}$ and $c \in \{2, \dots, 2^s - 1\}$ and $(i_{1,1}, i_{1,2}, \dots, i_{2,3}) \notin W$, ℓ' is divisible by $p((2^s(r-1+x) + c-2+y)z + i_{x,y})$ for every $x = 0, 1$ and $y = 0, 1, 2$.

The table encoding an accepting computation of N with a six-tuple (r, c) is illustrated in Fig. 8.

	0	...	$c-2$	$c-1$	c	...	2^s-1
0	$(q_0, 0)$...	$(\text{nil}, 0)$	$(\text{nil}, 0)$	$(\text{nil}, 0)$...	$(\text{nil}, 0)$
\vdots							
$r-1$			$t_{r-1,c-2}$	$t_{r-1,c-1}$	$t_{r-1,c}$		
\vdots							
r			$t_{r,c-2}$	$t_{r,c-1}$	$t_{r,c}$		
\vdots							
2^s-1	$(q_f, 0)$...	$(\text{nil}, 0)$	$(\text{nil}, 0)$	$(\text{nil}, 0)$...	$(\text{nil}, 0)$

Figure 8: The table of tape configurations encoded by ℓ' with a six-tuple at (r, c) .

The final formula $\varphi(\ell')$ is the disjunction of sub-formulas $\varphi_1(\ell')$, $\varphi_2(\ell')$, $\varphi_3(\ell')$, $\varphi_4(\ell')$, $\varphi_5(\ell')$, $\varphi_{6'}^{(i_{1,1}, i_{1,2}, \dots, i_{2,3})}(\ell')$, respectively for each of the six conditions. Then we simply return

$$\varphi(\ell') := \bigvee_{1 \leq i < j \leq z} \varphi_1^{i,j}(\ell') \vee \varphi_2(\ell') \vee \varphi_3(\ell') \vee \varphi_4(\ell') \vee \varphi_5(\ell') \bigvee_{(i_{1,1}, i_{1,2}, \dots, i_{2,3}) \notin W} \varphi_{6'}^{(i_{1,1}, i_{1,2}, \dots, i_{2,3})}(\ell').$$

Each of the constructed formulas is in the required form for the verifying procedure from Subsection 4.3.1, i.e. it is $\exists_{X_1, \dots, X_k \in \{0, \dots, 2^m-1\}} \psi(X_1, \dots, X_k)$, where ψ uses conjunctions and disjunctions and simple propositions (1)–(6), and where the value of m depends only on s (but the number k of existentially quantified variables might be different in different formulas). A disjunction of all constructed formulas can be easily rewritten to also have such form.

We only describe in detail how to construct the formula $\varphi_1^{i,j}(\ell')$ that is satisfied when, for some (r, c) , ℓ' is divisible by both $p((2^s r + c)z + i)$ and $p((2^s r + c)z + j)$. Formulas $\varphi_2(\ell')$, $\varphi_3(\ell')$, $\varphi_4(\ell')$, $\varphi_5(\ell')$, $\varphi_{6'}^{(i_{1,1}, i_{1,2}, \dots, i_{2,3})}(\ell')$ are constructed using a similar reasoning.

To construct $\varphi_1^{i,j}(\ell')$ we need to bound the primes used to represent the choice.

Lemma 22. $p(2^{2s}z) \leq 2^{11s}$.

Proof. A well-known bound [21] states that $p(n) < n(\log n + \log \log n)$ for $n \geq 6$. Therefore, for all $n \geq 1$ we have $p(n) \leq 2n^2$. Then, $p(2^{2s}z) = p(2^{2s}(2s+2)) \leq p(2^{2s} \cdot 2^{2s+1}) = p(2^{4s+1}) \leq 2^{8s+3} \leq 2^{11s}$. \square

Therefore, we set $m = 11s$ and quantify variables over $\{0, \dots, 2^{11s} - 1\}$. The expression $\varphi_1^{i,j}(\ell')$ is of the form $\exists_{X_1, \dots, X_k \in \{0, \dots, 2^{11s} - 1\}} \psi_1^{i,j}(X_1, \dots, X_k)$, where $\psi_1^{i,j}(X_1, \dots, X_k)$ is a conjunction of simple propositions. For clarity, we construct it incrementally.

Recall that we are looking for $r, c \in \{0, \dots, 2^s - 1\}$. We start with quantifying over $r, c, r', c', \Delta \in \{0, \dots, 2^{11s} - 1\}$ and including $(\Delta = 2^{10s}) \wedge (r \cdot \Delta = r') \wedge (c \cdot \Delta = c')$ in the conjunction. This guarantees that indeed $r, c \in \{0, \dots, 2^s - 1\}$.

Then we quantify over $z_r, z_c, m_r, m_c, m_s \in \{0, \dots, 2^{11s} - 1\}$ and include the following in the conjunction:

$$(z_r = 2^s \cdot z) \wedge (z_c = z) \wedge (z_r \cdot r = m_r) \wedge (z_c \cdot c = m_c) \wedge (m_s = m_r + m_c).$$

This ensures that $m_s = (2^s \cdot r + c)z$. Additionally, we quantify over $m_i, m_j, m'_i, m'_j \in \{0, \dots, 2^{11s} - 1\}$ and add:

$$(m_i = i) \wedge (m_j = j) \wedge (m_s + m_i = m'_i) \wedge (m_s + m_j = m'_j)$$

to the conjunction, which ensures that $m'_i = (2^s \cdot r + c)z + i$ and $m'_j = (2^s \cdot r + c)z + j$. Finally, we quantify over $p_i, p_j \in \{0, \dots, 2^{11s} - 1\}$ and include:

$$(p_i \text{ is the } m'_i\text{'th prime number}) \wedge (p_j \text{ is the } m'_j\text{'th prime number}) \wedge (p_i \mid \ell') \wedge (p_j \mid \ell')$$

in the conjunction. By construction, the obtained $\varphi_1^{i,j}(\ell')$ is satisfied when, for some (r, c) , ℓ' is divisible by both $p((2^s r + c)z + i)$ and $p((2^s r + c)z + j)$.

The other formulas are constructed using the same principle, and then we rewrite their disjunction to have the required form and obtain $\varphi(\ell')$. Formula $\varphi(\ell')$ is satisfied exactly when ℓ' does not correspond to an accepting computation of N . As was described in Subsection 4.3.1, we construct an NFA M , such that for some constants r_1, r_2 depending only on $\varphi(\ell')$, M accepts all words of length ℓ if and only if $\ell = r_1 \cdot \ell' + r_2$ and $\varphi(\ell')$ is not satisfied. Therefore, checking if M accepts all words of length ℓ , for some ℓ , is equivalent to checking if there exists an accepting computation of N . We state our final result:

Theorem 23. *Existential length universality (Problem 1) for NFAs is NEXPTIME-hard.*

5 The case where M is a regular expression

Theorem 24. *Given length universality (Problem 2) for regular expressions and NFAs is PSPACE-complete, and existential length universality (Problem 1) for regular expressions is PSPACE-hard.*

Proof. We transform a regular expression M into an NFA N with linear number of states in the length of M . To see that the problem is in non-deterministic linear space, note that to verify that $\Sigma^\ell \subsetneq L(M)$, all we need do is guess a string of length ℓ that is *not* accepted and then simulate N on this string. Of course, we do not store the actual string, we just guess it symbol-by-symbol, meanwhile counting up to ℓ to make sure the length is correct. The counter can be achieved in $O(\log \ell)$ space. By Savitch's theorem, it follows that the problem is in PSPACE.

To see that the problem is PSPACE-hard, we model our proof after the proof that non-universality for NFA's is PSPACE-hard (see e.g. [1, Section 10.6]). That proof takes a polynomial-space-bounded deterministic TM T and input x , and creates a polynomial length regular expression M that describes all strings which do not correspond to accepting computations. So $L(M) \neq \Sigma^*$ if and only if T accepts x . Strings can fail to correspond to accepting computations because they begin wrong, end wrong, have a syntax error, have an intermediate step that exceeds the polynomial-space bound, or have a transition that does not correspond to a rule of the TM. Our regular expression is a sum of these cases.

We now modify this construction as before in the proof of Theorem 4. First, we assume our TM always has a next move, except out of the halting state h , where there is no valid next move. Thus on any input we either halt or loop forever. Next, we create a regular expression M describing the language of invalid computations; its language contains all strings that begin wrong, have syntax errors, or fail to follow the rules of the TM, but no strings that are a prefix of a possibly-accepting computation. This shows that existential length universality is PSPACE-hard.

For hardness of given length universality, we set ℓ to be the length of words describing computations of length longer than the one following from the polynomial space bound for T , that is, $2^{|x|^c} + 1$. So if T fails to accept an input x , for every length there is some prefix of a computation that is correct, which is not in $L(M)$. On the other hand, if T accepts x , then every string of length at least ℓ is either syntactically incorrect, or has a bogus transition – including a transition out of the halting state – so every string of that length is in the language. \square

The problem about the exact complexity of existential length universality in the case of a regular expression remains open, as well as the bounds for the minimal universality length.

Open Question 25. What is the complexity of the existential length universality (Problem 1) when M is a regular expression? What is the largest possible minimal universality length in terms of the length of M ?

If we could show that the minimal universality length is at most exponential, then it would immediately imply that the problem is PSPACE-complete. Probably, the constructions from Subsection 4.1 cannot be encoded in regular expressions. The best example that we have found has exponential minimal universality length.

Theorem 26. *There exists a binary regular expression with n input symbols for which the minimal universality length is $2^{\Omega(n)}$.*

Proof. We already know that there is class of regular expressions such that the shortest not specified string looks like the binary expansions of $0, 1, \dots, 2^m - 1$ separated by delimiters. This can be found, for example, in [8, Section 5]. More precisely, the shortest not specified string looks like (for $m = 3$):

```
#000#100#010#110#001#101#011#
 100 010 110 001 101 011 111
```

where columns denote composite symbols and the bottom row is one more than the top row and numbers are written least-significant-bit-first.

Now we can modify this construction so not only does it not accept this string, but it also fails to accept every prefix of this string. This is easy, as it just involves deleting the conditions about ending properly (so condition (2) from [8, Section 5]).

So our regular expression is a sum of two parts. One specifies all strings except those that are prefixes of the string above; the other one specifies all strings that end in the proper string, which in for $m = 3$ would be

```
#011#
 111
```

Since no prefix ends with this except the total string itself, this regular expression will fail to specify all strings of every length until it gets to the length of the total string $((m + 1)(2^m - 1))$. As in [8], the regular expression can be converted to a binary one with n input symbols, where n is linear in m . Thus, its minimal universality length is $2^{\Omega(n)}$. \square

References

- [1] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1974.
- [2] R. Alur and P. Madhusudan. Visibly Pushdown Languages. In *Proc. of the 36th Annual ACM Symposium on Theory of Computing*, STOC 2004, pages 202–211. ACM, 2004.
- [3] E. Bach and J. Shallit. *Algorithmic Number Theory*. MIT Press, 1996.
- [4] P. Barceló, L. Libkin, and J. L. Reutter. Querying regular graph patterns. *J. ACM*, 61(1):8:1–8:54, 2014.
- [5] N. Bertrand, P. Bouyer, T. Brihaye, and A. Stainer. Emptiness and Universality Problems in Timed Automata with Positive Frequency. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP 2011, pages 246–257. Springer, 2011.
- [6] S. Cho and D. T. Huynh. The parallel complexity of finite-state automata problems. *Information and Computation*, 97(1):1–22, 1992.

- [7] M. De Wulf, L. Doyen, T.A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In Thomas Ball and Robert B. Jones, editors, *CAV*, pages 17–30. Springer, 2006.
- [8] K. Ellul, B. Krawetz, J. Shallit, and M.-w. Wang. Regular expressions: New results and open problems. *J. Autom. Lang. Comb.*, 10(4):407–437, 2005.
- [9] M. Holzer. On emptiness and counting for alternating finite automata. In *DLT 1996*, pages 88–97. World Scientific, 1996.
- [10] M. Holzer and M. Kutrib. Descriptive and computational complexity of finite automata – a survey. *Information and Computation*, 209(3):456–470, 2011.
- [11] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [12] H. B. Hunt, III and D. J. Rosenkrantz. On equivalence and containment problems for formal languages. *J. ACM*, 24(3):387–396, 1977.
- [13] H. B. Hunt, III and D. J. Rosenkrantz. Computational parallels between the regular and context-free languages. *SIAM Journal on Computing*, 7(1):99–114, 1978.
- [14] H. B. Hunt, III, D. J. Rosenkrantz, and T. G. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences*, 12(2):222–268, 1976.
- [15] N. D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975.
- [16] J.-K. Kao, N. Rampersad, and J. Shallit. On nfas where all states are final, initial, or both. *Theoretical Computer Science*, 410(47):5010–5021, 2009.
- [17] M. Krötzsch, T. Masopust, and M. Thomazo. On the Complexity of Universality for Partially Ordered NFAs. In *MFCS*, pages 61:1–61:14, 2016.
- [18] A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *Symposium on Switching and Automata Theory (SWAT 1972)*, pages 125–129. IEEE Computer Society, 1972.
- [19] N. Rampersad, J. Shallit, and Z. Xu. The computational complexity of universality problems for prefixes, suffixes, factors, and subwords of regular languages. *Fundamenta Informaticae*, 116(1–4):223–236, 2012.
- [20] D. Rosenblatt. On the graphs and asymptotic forms of finite Boolean relation matrices. *Naval Research Quart.*, 4:151–167, 1957.
- [21] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Ill. J. Math.*, 6:64–94, 1962.

- [22] J. Shallit. Open problems in automata theory: An idiosyncratic view. <https://cs.uwaterloo.ca/~shallit/Talks/bc4.pdf>.
- [23] G. Stefanoni, B. Motik, M. Krötzsch, and S. Rudolph. The Complexity of Answering Conjunctive and Navigational Queries over OWL 2 EL Knowledge Bases. *Journal of Artificial Intelligence Research*, 51:645–705, 2014.
- [24] L. J. Stockmeyer and A. R. Meyer. Word Problems Requiring Exponential Time. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC, pages 1–9, 1973.