

# Existential length universality

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

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

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

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

December 13, 2018

## Abstract

We study the following natural variation on the classical universality problem: given an automaton  $M$  of some type (DFA/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 and deep construction, we prove that the problem is NEXPTIME-complete, and the smallest such  $\ell$  can be doubly exponential in the number of states. In the case of a DFA the problem is NP-complete, and there exist examples for which the smallest such  $\ell$  is of the form  $e^{\sqrt{n} \log n (1+o(1))}$ , which is best possible, where  $n$  is the number of states. In the case of a PDA this problem is recursively unsolvable, while the smallest such  $\ell$  cannot be bounded in the number of states by any computable function. Finally, we show that in all the cases the problem becomes computationally easier when the length  $\ell$  is also given in the input.

## 1 Introduction

In this paper we consider the computational complexity of the following natural decision problem:

Given  $M$ , an  $n$ -state machine of some type (DFA/NFA/PDA) over an input alphabet  $\Sigma$  of fixed size, does there exist an integer  $\ell \geq 0$  such that  $\Sigma^\ell \subseteq L(M)$ ?

We call this the *existential length universality* problem. It is analogous to the classical *universality* problem:

Given  $M$ , is  $L(M) = \Sigma^*$ ?

which has been widely studied (see, e.g., [1, 3, 4, 7, 8, 10, 11, 12, 13, 14, 17]).

Furthermore, if such an  $\ell$  exists, we are interested in how large it can be. The smallest such  $\ell$  is called the *minimal universality length*.

We also study the complexity of the question:

Given  $M$ , and an integer  $\ell$  (represented in binary), is  $\Sigma^\ell \subseteq L(M)$ ?

The outline of the paper is 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 examples for which the minimal universality length is of the form  $e^{\sqrt{n \log n}(1+o(1))}$ , which is best possible. Finally, the deepest case is covered in Section 4, where  $M$  is a nondeterministic finite automaton. Here, using a novel construction, we prove that existential length universality is NEXPTIME-complete, while the minimal universality length can be doubly exponential in the number of states of the NFA.

In all the cases, the existential length universality problem becomes easier when the length  $\ell$  (represented in binary) is also given: it is in EXPTIME for PDAs, polynomial for DFAs, and PSPACE-complete for NFAs.

The following straightforward fact implies that all our complexity results are valid when the input alphabet is binary:

**Lemma 1.** *Let  $M$  be a DFA/NFA/PDA with  $n$  states over an alphabet  $\Sigma$  of size  $k \geq 2$ , and let  $\ell$  be the minimal universality length of  $M$ . Then, there exists a DFA (NFA, PDA, respectively)  $M'$  over a binary alphabet with  $2^{\lceil \log_2 k \rceil - 1} n$  states, whose minimal universality length is equal to  $\ell \cdot \lceil \log_2 k \rceil$ .*

*Proof.* We can replace every state by a full binary tree of length  $\log_2 k$ , so that every letter  $a \in \Sigma$  acts in  $M$  as its binary representation acts in  $M'$  on the root states of the trees. The final states of  $M'$  are the roots of the trees corresponding to the final states of  $M$ .  $\square$

## 2 The case where $M$ is a PDA

**Theorem 2.** *Existential length universality is recursively unsolvable for PDA's.*

We mimic the usual proof [9, 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 other configuration is reversed. 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 (nondeterministically) 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.

*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.

We now 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 one configuration does not follow from the previous. If the last configuration is incomplete, and what is actually present does not violate any rules of  $T$ , however, our PDA *does not* accept.

First, 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  $n$  such that  $M$  accepts all strings of length  $n$ .

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

**Corollary 3.** *Fix an alphabet  $\Sigma$ . For a PDA  $M$ , define  $r = r(M)$  to be the least integer  $s$  such that  $M$  accepts all strings of length  $s$  (if it exists). Let  $f(n)$  be the maximum of  $r(M)$  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 any PDA  $M$  of size  $n$  by (1) computing  $r = g(n)$ ; (2) testing, by brute-force enumeration, whether  $M$  accepts all strings of length  $\ell$  for all  $\ell \leq n$ . (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 decision problem:

Given a PDA  $M$ , and an integer  $\ell$ , is  $\Sigma^\ell \subseteq L(M)$ ?

is solvable. Furthermore, it is solvable in exponential time, if  $\ell$  is represented in unary. We can convert a PDA in polynomial time to a CFG generating the same language. Then we can try each of the  $|\Sigma|^\ell$  strings of length  $\ell$  to see if they are generated (using, for example, the Cocke-Younger-Kasami dynamic programming solution).

**Open Question 4.** Is this decision problem EXPTIME-hard?

### 3 The case where $M$ is a DFA

**Theorem 5.** *Existential length universality 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 letter  $a$ . Now it is easy to see that  $M$  accepts all strings of length  $m$  if and only if every path of length  $m$  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  $m$  from state  $q_i$  to state  $q_j$  if and only if  $B^m$  has a 1 in row  $i$  and column  $j$ , where by  $B^m$  we mean the Boolean power of the matrix  $B$ .

So  $M$  accepts all strings of length  $m$  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  $m$ , we simply guess  $m$ , compute  $B$ , raise  $B$  to the  $m$ '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  $m$  is exponentially bounded in magnitude.

To see that  $m$  is exponentially bounded, we can argue that  $m \leq 2^{|Q|^2}$ . This follows trivially because our matrix is of dimension  $|Q| \times |Q|$ ; after we see  $2^{|Q|^2}$  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 [15] 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 6.** *Let  $M$  be a DFA with  $n$  states. If there exists  $\ell$  such that  $M$  accepts all strings of length  $m$ , then the minimal such  $\ell$  is  $\leq e^{\sqrt{n \log n}(1+o(1))}$ .*

The upper bound in the previous result is tight, at least if we are allowed to let the alphabet size grow with  $n$ .

**Theorem 7.** *For each sufficiently large  $n$ , there exists a DFA  $M$  with  $n$  states for which the minimal universality length  $\ell$  is  $\geq e^{\sqrt{n \log n}(1+o(1))}$ .*

*Proof.* The DFA's look like the following: 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 return to the original state in the cycle, which is accepting. This DFA accepts the language

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

For each length  $\ell < p_1 p_2 \cdots p_t - 1$ , there exists a prime  $p_i$ ,  $1 \leq i \leq t$ , such that  $\ell \not\equiv p_i - 1 \pmod{p_i}$ , so no string in  $a_i \Sigma^\ell$  is accepted. However, for  $\ell = p_1 p_2 \cdots p_t - 1$ , all strings of length  $\ell$  are accepted.

This DFA has  $p_1 + p_2 + \cdots + p_t + 1$  states and the least  $n$  for which all strings of length  $n$  is accepted is  $p_1 p_2 \cdots p_t - 1$ . From the prime number theorem we know that

$$p_1 + p_2 + \cdots + p_t + 1 \sim \frac{1}{2} t^2 \log t,$$

(see, for example, [2, p. 29]) and  $p_1 p_2 \cdots p_t - 1 \sim e^{t(1+o(1))}$ , from which the claim follows.  $\square$

**Theorem 8.** *Existential length universality is NP-hard for DFA's.*

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

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

Suppose there are  $t$  variables in  $\varphi$ , 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. If the integer is 1 mod  $p_i$ , then it corresponds to an assignment where  $v_i$  is set to 1; if the integer is 0 mod  $p_i$ , 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  $\Sigma$  consisting of the integers  $1, 2, \dots, s$ , where  $s$  is the number of clauses. The nonaccepting start state has a transition on each element of the alphabet to a cycle corresponding to the appropriate clause. Inside each cycle, transitions go to the next state on all elements of  $\Sigma$ . 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  $n$  if and only if  $n$  corresponds to a satisfying assignment for  $\varphi$ . To see this, note that if the DFA accepts all strings of length  $n$  for some  $n$ , 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  $n$  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 is bounded in magnitude by  $O(i \log i)$  (e.g., [16]), and each cycle is therefore of size at most  $O((t \log t)^3)$ , so the total number of states is  $O(s(t \log t)^3)$ .  $\square$

We now turn to the problem where  $\ell$  is also given.

**Theorem 9.** *The decision problem:*

*Given a DFA  $M$ , and an integer  $\ell$  (represented in binary), is  $\Sigma^\ell \subseteq L(M)$ ?*

*is solvable in polynomial time (in the size of  $M$  and  $\log \ell$ ).*

*Proof.* The idea is just like the proof of Theorem 5 above. Given  $M$ , create the boolean matrix  $B$  just as in that proof. Then  $\Sigma^\ell \subseteq L(M)$  if and only if every 1 in row 0 of  $B^n$  occurs in a column corresponding to a final state of  $M'$ . We can compute  $B^\ell$  using the usual “binary method” or “doubling-up” trick, using only  $O(\log \ell)$  boolean matrix multiplications.  $\square$

## 4 The case where $M$ is an NFA

The ordinary universality problem for NFAs is known to be PSPACE-complete [1, Section 10.6]. Also, if the NFA does not accept  $\Sigma^*$ , then the length of the shortest non-accepted words has at most exponential length.

Here we show that the existential length variant is essentially harder in the case of an NFA. Deciding existential length universality 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 for the minimal universality length and the complexity of the existential problem.

**Proposition 10.** *Let  $M$  be an NFA with  $n$  states. If there exists  $\ell$  such that  $M$  accepts all strings of length  $m$ , then the smallest such  $\ell$  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 6 we get

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

□

**Proposition 11.** *Existential length universality for NFAs is in NEXPTIME.*

*Proof.* We determinize  $M$  to a DFA with at most  $2^n$  states. Then by Theorem 5 the exponential length is solvable in nondeterministic exponential time. □

Before we go into NEXPTIME-hardness, we state that the problem is easier when  $\ell$  is also given.

**Theorem 12.** *The decision problem*

*Given an NFA  $M$ , and an integer  $\ell$  (represented in binary), is  $\Sigma^\ell \subseteq L(M)$ ? is PSPACE-complete.*

*Proof.* To see that the problem is in nondeterministic linear space, note that to verify that  $\Sigma^\ell \subsetneq L(M)$ , all we need do is guess a string of length  $\ell$  that is *not* accepted and then simulate  $M$  on this string. Of course, we do not store the actual string, we just guess it symbol-by-symbol, meanwhile counting up to  $\ell$  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 usual proof that nonuniversality for NFA's in PSPACE-hard (see e.g. [6]). That proof takes a polynomial-space-bounded TM  $T$  and input  $x$ , and creates a polynomial-size NFA  $M$  (actually, a regular expression, but the conversion to NFA is trivial) that accepts 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. Using nondeterminism, an NFA can guess where an error occurs and verify it.

We now modify this construction as before in the proof of Theorem 2. 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 an NFA accepting invalid computations; it accepts strings that begin wrong or have syntax errors or fail to follow the rules of the TM, but fails to accept if something is a prefix of a possibly-accepting computation.

We set  $\ell$  to be the length of words describing computations of length longer than the polynomial space bound for  $T$ . So if  $T$  fails to accept an input  $x$ , for every length there is some prefix of a computation that is correct, and hence the NFA fails to accept. On the other hand, if  $T$  accepts  $x$ , then every string of length at least  $\ell$  is either syntactically incorrect, or has a bogus transition – including a transition out of the halting state – so every string of that length is accepted. □

Our final goal 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 nondeterministic 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 10.

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 can be translated into an NFA. A proper computation of a given program in this language will correspond to a word that is not accepted by the automaton, and vice versa.

Let our NFA be  $(Q, \Sigma, \delta, q_0, F)$  and let  $m \geq 1$  be a fixed integer. By a *configuration* we mean a subset  $C \subseteq Q$  of possible current states that the NFA could be in, after reading some word. We say that a state is *active* if it belongs to the current configuration. Our NFA has a distinguished state  $q_{\text{acc}} \in F$ , which is such that  $\delta(q_{\text{acc}}, a) = q_{\text{acc}}$  for every  $a \in \Sigma$ . We say that a configuration  $C$  is *proper* if it does not contain  $q_{\text{acc}}$ . The other states belong to certain gadgets that we will define.

A *variable*  $V$  is a subset of states  $\{v_1, \dots, v_m, v'_1, \dots, v'_m\}$  in the NFA. We say that  $V$  is *valid* in some configuration  $C \subseteq Q$  of the NFA if  $v_i \in C$  if and only if  $v'_i \notin C$ . In other words, the states  $v'_1, \dots, v'_m$  are the complements of the states  $v_1, \dots, v_m$ . If  $V$  is valid in a configuration  $C$ , then it can be viewed as storing some positive integer, namely,

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

Therefore, a variable can store any integer from 0 to  $2^m - 1$ .

Every gadget  $G$  is a triple  $(Q^G, \Sigma^G, \delta^G)$ , where  $Q^G$  is a set of states,  $\Sigma^G$  is an alphabet, and  $\delta^G: Q^G \rightarrow 2^{Q^G} \cup \{q_{\text{acc}}\}$  is the transition function. The sets of states and the alphabets are disjoint in every gadget, and the membership is specified by the superscript  $G$ . When the context is clear we omit the superscript. The set  $Q^G$  consists of *control flow* states  $P^G$  and states of *input*, *output*, or *internal* variables. We treat a gadget as a separate part of our NFA, and show that it possesses some properties for a configuration in which  $s = p_0 \in P^G$  is active and the input variables are valid. The control flow states  $P^G$  represent the current position during computation of the program and possess the property that in a proper configuration exactly one of them is active. There are two special control flow states: the *start* state  $s$  and the *target* state  $t$ ; they are used to join gadgets together in the whole NFA.

### 4.1.1 Basic gadgets

#### • Selection Gadget $S$ .

The goal of this gadget is to allow a nondeterministic selection of an arbitrary value of one



variable. There are no input variables and there is one output variable  $V$ . We have the states  $Q^S = P \cup V$  and the letters  $\Sigma^S = \{\alpha_1, \alpha_2\}$ , where  $P = \{s = p_0, p_1, \dots, p_{m-1}, p_m = t\}$ . The gadget is illustrated in Fig. 1.

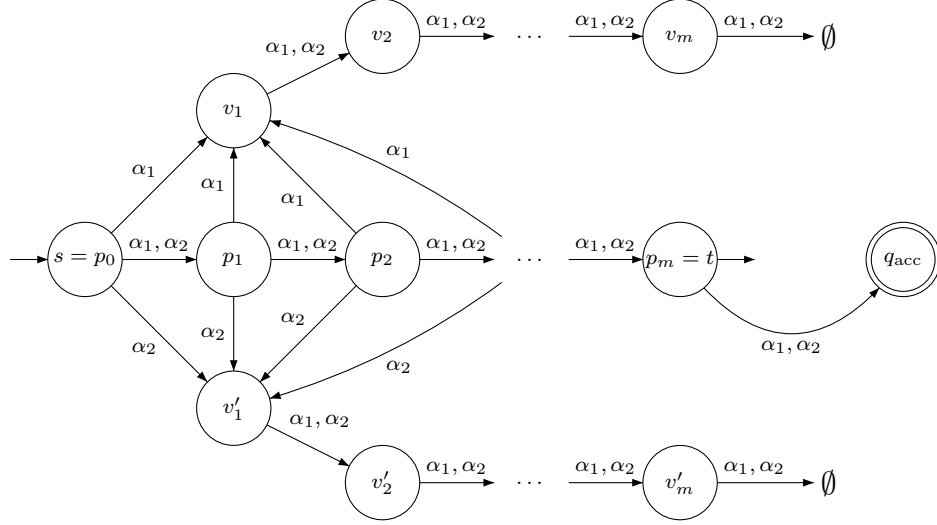


Figure 1: Selection Gadget.

The letters  $\alpha_1$  and  $\alpha_2$  allow transitions through the states  $p_0, p_1, \dots, p_{m-1}, p_m$  and, at each transition, a choice of either  $v_1$  or  $v'_1$  to be active. Also, each  $v_i$  and  $v'_i$  is shifted to  $v_{i+1}$  and  $v'_{i+1}$ , respectively, and  $v_m$  and  $v'_m$  are mapped to  $\emptyset$ , which ensures that the initial contents of  $V$  is neglected.

The transitions are defined as follows:

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

**Lemma 13.** *Let the current configuration  $C$  contain only the state  $s$  and possibly some states of  $V$ . After reading a word of length exactly  $m$  with letters from  $\Sigma^S$  such that the obtained configuration is proper,  $t$  is active and  $V$  is valid. There exists such a word for every possible value of  $V$ . When reading a word shorter than  $m$ ,  $t$  cannot become active (unless the configuration is non-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  $v'_i$  is active,

depending on the  $i$ 'th letter. Thus for every value of  $V$  there is the unique word  $w\{\alpha_1, \alpha_2\}^m$  resulting in setting this value. Observe that if  $w$  is shorter than  $m$ , then  $t$  cannot become active, since the shortest path from  $s$  to  $t$  has length  $m$ .  $\square$

• **Equality Gadget.**

The goal of this gadget is to take two variables and continue computation (permitting to obtain a proper configuration) if and only if their values are equal. There are two input variables  $U$  and  $V$ , and there are no output variables. We have the states  $Q^{\text{EQ}} = P \cup U \cup V$  and the letters  $\Sigma^{\text{EQ}} = \{\alpha_1, \alpha_2\}$ , where  $P = \{s = p_0, p_1, \dots, p_{m-1}, p_m = t\}$ . The gadget is illustrated in Fig. 2.

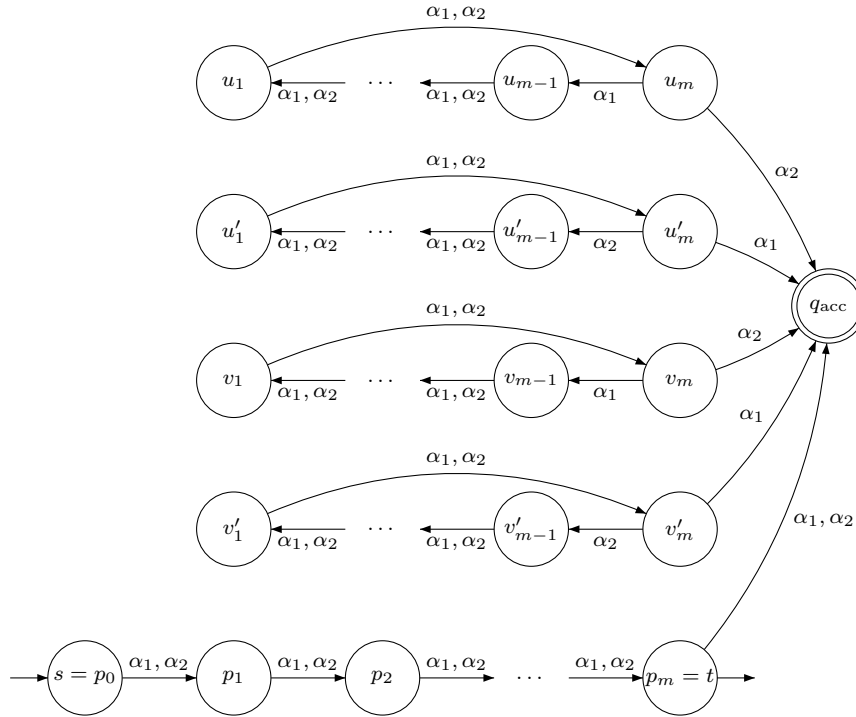


Figure 2: Equality Gadget.

The letters  $\alpha_1$  and  $\alpha_2$  permit transitions through the states  $s = p_0, p_1, \dots, p_m = t$ , while checking every position of  $U$  and  $V$  to see if they agree. The transitions are defined in the

same way for both the variables, and they cyclically shift their states:

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

**Lemma 14.** *Let the current configuration  $C$  contain only the state  $s$  and valid variables  $U$  and  $V$ . After reading a word of length exactly  $m$  with letters from  $\Sigma^{\text{EQ}}$  such that the obtained configuration is proper, it must be that  $u_i \in C$  if and only if  $v_i \in C$  for all  $i = 1, \dots, m$ , and  $t$  becomes active. There exists such a word if the values of  $U$  and  $V$  in  $C$  are equal. When reading a shorter word,  $t$  cannot become active (unless the configuration is non-proper).*

*Proof.* After reading a word  $w \in \{\alpha_1, \alpha_2\}^m$  we get that  $t$  is active. If  $u_i \in C$  then the  $(i+1)$ 'st letter of  $w$  (or first letter if  $i = m$ ) must be  $\alpha_1$ . Similarly if  $u'_i \in C$  then the  $(i+1)$ 'st letter of  $w$  (or first if  $i = m$ ) must be  $\alpha_2$ . The same holds for the states of  $V$ . Thus if  $u_i \in C$  and  $v'_i \in C$ , or if  $u'_i \in C$  and  $v_i \in C$ , then  $q_{\text{acc}}$  cannot be avoided by any such  $w$ . Otherwise there exists a unique word  $w$  such that  $t$  becomes active and  $q_{\text{acc}}$  does not.  $\square$

### • Inequality Gadget.

This gadget is similar to the equality gadget. Its goal is to take two variables and continue computation if and only if their values are different. There are two input variables  $U$  and  $V$ , and there are no output variables. We have the states  $Q^{\text{INEQ}} = P \cup U \cup V$  and the letters  $\Sigma^{\text{INEQ}} = \{\alpha_1, \alpha_2, \alpha_s\}$ , where  $P = \{s = p_0, p_1, \dots, p_{m-1}, p_m = t\}$ . The gadget is illustrated in Fig. 3.

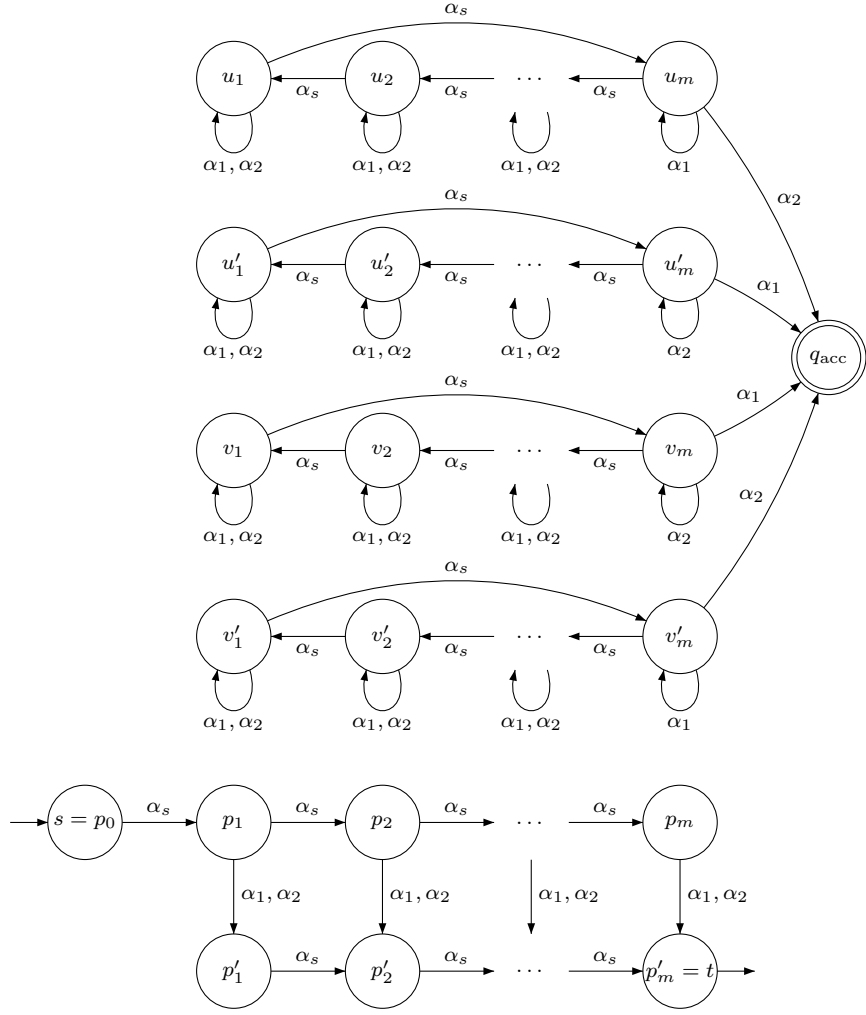


Figure 3: Inequality Gadget. All the omitted transitions go to  $q_{acc}$ .

The letter  $\alpha_s$  permits transitions through the states  $p_0, p_1, \dots, p_m$ , and it cyclically shifts

both the variables. Its transition function is defined as follows:

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

At some point, when the position at which the variables differ, either the letter  $\alpha_1$  or the letter  $\alpha_2$  may be applied, which also switches a state  $r_i$  to the state  $r'_i$ . Their transitions are defined as follows:

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

**Lemma 15.** *Let the current configuration  $C$  contain only the state  $s$  and the states of the valid variables  $U$  and  $V$ . Then after reading a word of length exactly  $m+1$  with letters from  $\Sigma^{\text{INEQ}}$  such that the obtained configuration is proper, then it must be that  $u_i \in C$  and  $v_i \notin C$  or  $u_i \notin C$  and  $v_i \in C$  for some  $i$ , and  $t$  becomes active. There exists such a word if the values of  $U$  and  $V$  are different. When reading a shorter word,  $t$  cannot become active (unless the configuration is non-proper).*

*Proof.* Consider a word  $w$  of length  $m+1$ . If  $q_{\text{acc}}$  does not become active, then  $t$  becomes active and  $w$  contains exactly  $m$  occurrences of  $\alpha_s$  and one occurrence of either  $\alpha_1$  or  $\alpha_2$ . If  $\alpha_1$  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  $\alpha_2$ . 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.**

The goal of this gadget is to take a variable and increase its value by 1. There is one input variable  $V$ , which is also the single output variable. We have the states  $Q^I = P \cup V$  and the letters  $\Sigma^I = \{\alpha_a, \alpha_c, \alpha_p\}$ , where  $P = \{s = p_0, p_1, \dots, p_m, p'_1, \dots, p'_m = t\}$ . The gadget is illustrated in Fig. 4.

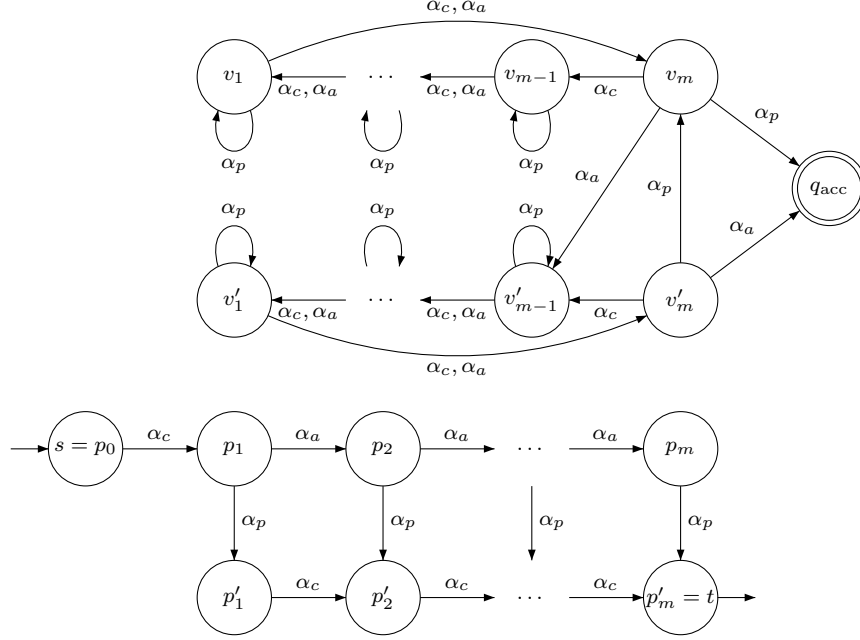


Figure 4: Incrementation Gadget. All the omitted transitions go to  $q_{acc}$ .

The incrementation gadget performs the written addition of one for the value of variable  $V$  interpreted in binary. First, the letter  $\alpha_c$  begins the process: this is the only letter that can be applied when  $p_0$  is active. Its transitions are as follows:

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

Then, the letter  $\alpha_a$  cyclically shifts the states of  $V$ , and must be applied until the  $m$ 'th position in  $V$  is empty. Every time this happens the  $m$ 'th position becomes cleared. Its

transitions are as follows:

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

Then, the letter  $\alpha_p$  must be applied, which fills the  $m$ 'th position in  $V$ , and the active state  $p_i$  is moved to the corresponding state  $p'_i$ . Its transition are as follows:

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

Finally, the letter  $\alpha_c$  finishes the cyclic shifting of  $V$ , while moving the active state over  $p'_i, \dots, p'_m$ .

**Lemma 16.** *Let the current configuration  $C$  contain only the state  $s$  and the valid variable  $V$ . Then after reading exactly  $m+1$  letters from  $\Sigma^1$ , if the new obtained configuration  $C'$  is proper, then only  $t$  is active, and  $V$  is valid and its value in  $C'$  is larger by 1 than in  $C$ . There exists such a word for every value of  $V$  in  $C$  except  $2^m - 1$ . When reading a shorter word,  $t$  cannot become active (unless the configuration is non-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  $\alpha_a$  ( $1 \leq j \leq i$ ). If after reading it the current configuration is still proper, then it must be that  $v'_j \notin C$  and so  $v_j \in C$ , since  $v_j$  and  $v'_j$  were mapped to  $v_m$  and  $v'_m$  by  $\alpha_c \alpha_a^{j-1}$ . Next, it must be that  $v_i \notin C$ , because of the application of  $\alpha_p$ . Since  $w$  cyclically shifts  $V$  exactly  $m$  times, we end up with  $C'$  such that:

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

- **Waiting Gadget.**

This is a very simple gadget which just delays the computation for a fixed number of  $D \geq 1$  letters. There are the states  $Q^W = P = \{s = p_0, p_1, \dots, p_{D-1}, p_D = t\}$  and the unary alphabet  $\Sigma^W = \{\alpha\}$ . The transitions of  $\alpha$  are defined as follows:

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

Clearly, when  $p_0$  becomes active, then  $\alpha^D$  must be read to avoid  $q_{\text{acc}}$ , which makes  $t$  active.

#### 4.1.2 Instructions

In a program, every gadget in a particular context is represented by an *instruction*. So the Equality Gadget for existing variables  $V_1$  and  $V_2$  is represented by  $V_1 = V_2$  and the Inequality Gadget is represented by  $V_1 \neq V_2$ , The Incrementation Gadget is represented by  $V++$ , the Selection Gadget is represented by  $\text{SELECT}(V)$ , and the Waiting Gadget is represented by  $\text{WAIT}(D)$ , where  $D$  is a fixed number.

Now we can write a program, which is just a sequence of instructions joined together in some way. For every instruction we add new (unique) copies of states and letters of their gadgets. The transitions for the states that are not explicitly defined map these states to  $q_{\text{acc}}$  if they are control flow states, and they fix the states if they are states of variables. In particular, applying the transition of a letter from an instruction, while having an active control flow state from another instruction, results in mapping this state to  $q_{\text{acc}}$ .

Given a proper configuration, we say that the word represents a *proper computation* if the obtained configuration after reading this word is proper. In our construction the control flow states form a directed graph, where the out-degree at every state of every letter is at most one. So the construction ensures that during every proper computation exactly one control flow state is active.

- **Sequence** First, given a sequence of consecutive instructions  $I_1, \dots, I_k$  we identify the target states  $t^i$  with the start states  $s^{i-1}$ , for  $i = 1, \dots, k-1$ . Then  $s_1$  and  $t_k$  are the start and target states of the obtained gadget (a compound instruction). We represent this construction by writing

$$\begin{array}{l} I_1 \\ \dots \\ I_k \end{array}$$

For example, if  $k = 2$ ,  $I_1 = V++$ , and  $I_2 = V++$ , then the resulting sequence forms a gadget whose proper computation of length  $2m + 2$  increases the value in  $V$  by 2.

For the instructions below we assume that  $s$  and  $t$  are new start and target states.

- **Choose Instruction.**

Given a set of instructions  $I_1, \dots, I_k$  we can allow choosing one of them nondeterministically, that is, every proper computation must go through exactly one of the instructions.



Let  $s_i$  and  $t_i$  be the start and target state of  $I_i$ . We add  $k$  unique letters  $\alpha_i$  whose action maps  $s$  to  $s_i$ . Also, all states  $t_i$  are identified with  $t$ .

This construction is represented by writing

**choose:**

$I_1$

**or:**

$\dots$

**or:**

$I_k$

**end choose**

• **Checking Instruction.**

This verifies whether a given fixed logical formula  $\varphi(V_1, \dots, V_k)$  with  $\wedge$ ,  $\vee$ , and  $\neg$ , and whose simple propositions are of the forms  $(V_i = V_j)$  or  $(V_i \neq V_j)$  for some  $i, j$ , is satisfied in the current context.

We transform  $\varphi$  by De Morgan's laws and by replacing  $(V_i = V_j)$  with  $(V_i \neq V_j)$  into the equivalent  $\varphi'$  without negations. The gadget is defined recursively. For  $(V_i = V_j)$  or  $(V_i \neq V_j)$  we use the Equality Gadget or the Inequality Gadget, respectively. If  $\varphi = \varphi_1 \wedge \dots \wedge \varphi_h$ , then we join them using the Checking Instruction for the  $h$  subformulas, as a sequence of instructions. If  $\varphi = \varphi_1 \vee \dots \vee \varphi_h$ , then we join the Checking Instruction for  $h$  subformulas as in the Choose Instruction.

It follows that for a given configuration containing  $s$  and valid variables  $V_1, \dots, V_k$ , there is a word leading to a configuration  $C'$  such that  $q_{\text{acc}} \notin C'$  and  $t \in C'$  if and only if  $\varphi'$  and so  $\varphi$  is satisfied. Note that this word may have different length depending on the current configuration  $C$ .

We write  $\text{CHECK}(\varphi(V_1, \dots, V_k))$  to denote this construction. Later, we will extend the formula to handle additional simple propositions than (in)equalities.

• **Assignment Instruction.**

An assignment is denoted by  $V = E(V_1, \dots, V_k)$ , where  $V$  is a variable and  $E$  is a gadget that takes  $k$  input variables  $V_1^E, \dots, V_k^E$  which take the values of variables  $V_1, \dots, V_k$  from the context, and outputs one variable  $W^E$ . We assume that  $E$  does not initially assume anything about  $W^E$  and that it eventually does not modify the input variables; hence we use neither the Selection Gadget nor the Incrementation Gadget here. For example, an allowed gadget is such that it copies the states of  $V_1$  to  $W^E$  and applies the Incrementation Gadget to  $W^E$ .

First, there is a letter whose action maps  $s$  into the start state  $s^E$  of  $E$ , and the corresponding states of the variables  $V_1, \dots, V_k$  to the states of the input variables  $V_1^E, \dots, V_k^E$  of  $E$ , whereas the states of  $V_1^E, \dots, V_k^E$  are mapped to  $\emptyset$ . We add another letter whose action maps the target state  $t^E$  of  $E$  to  $t$ , and the corresponding states of  $W^E$  to  $V$ .

Note that, since the states of input variables are mapped to  $\emptyset$  and the gadgets used in assignments do not assume that the state of the output variable are inactive, the same assignment can be processed multiple times (e.g., in a loop).

A fixed constant  $c$  ( $0 \leq c \leq 2^m - 1$ ) or a variable  $U$  from the context may also play the role of  $E$ . In these cases we add just one letter, which maps  $s$  to  $t$  together with the states

of  $V$  that encode the value  $c$ , or copy the corresponding states of  $U$  to  $V$ .

• **If-Else Instruction.**

It contains a Checking Instruction with  $\varphi$  and two internal instructions  $I_1$  and  $I_2$ . This construction is represented as follows:

```

if  $T$  then
   $I_1$ 
else
   $I_2$ 
end if

```

The second instruction  $I_2$  may be empty, and then we omit the *else* part. The gadget is implemented as follows:

```

choose:
  CHECK( $\varphi$ )
   $I_1$ 
or:
  CHECK( $\neg\varphi$ )
   $I_2$ 
end choose

```

Thus, there is a nondeterministic choice for the value of  $\varphi$ , which is then verified by the Checking Instruction.

• **While Instruction.**

This is easily constructed using *If-Else*, where  $I_2$  is empty, and the target state of  $I_1$  is identified with  $s$ . The target state  $t$  is identified with the target state of the Checking Instruction in the *else* part. A special variant of this is with **true** condition, which just does not involve the Choose Instruction nor the Checking Instructions.

### 4.1.3 Compound gadgets.

Now we define the rest of the required gadgets as programs. They may be then used in an Assignment Instruction if they have exactly one output variable, or in a Checking Instruction if they do not have output variables.

• **Addition Gadget.**

The addition of two variables  $U + V$  is defined by Alg. 1.

---

**Algorithm 1** Addition Gadget  $U + V$ .

---

**Input variables:**  $U, V$

**Output variable:**  $U$

**Internal variable:**  $X$

```

1:  $X \leftarrow 0$                                  $\triangleright$  Assignment of a fixed value to  $X$ 
2: while  $X \neq V$  do                             $\triangleright$  While with Checking Instruction with Inequality Gadget
3:    $X++$                                            $\triangleright$  Incrementation Gadget
4:    $U++$                                            $\triangleright$  Incrementation Gadget
5: end while

```

---

Note that if the result  $U + V$  is larger than  $2^m - 1$ , then this gadget does not allow a proper (and long enough) computation, because at some point we have to go through the Incrementation Gadget in line 4 when  $U(C) = 2^{m-1}$ .

• **Multiplication Gadget.**

The multiplication of two variables  $U \cdot V$  is defined by Alg. 2. This uses the Addition Gadget to add the number  $U$  to the output variable  $W$  a total of  $V$  times.

---

**Algorithm 2** Multiplication Gadget  $U \cdot V$ .

---

**Input variables:**  $U, V$

**Output variable:**  $W$

**Internal variable:**  $X$

```

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

```

---

As before, if the result  $U + V$  is larger than  $2^m - 1$ , then this gadget does not allow a proper (and long enough) computation.

• **Primality Gadget.**

The primality testing of a variable  $P$  is defined by Alg. 3. This does not have output variables, and a proper computation is possible if and only if  $P$  is prime.

We test whether  $P$  is prime by enumerating all numbers  $2 \leq X, Y < P$  and checking whether  $X \cdot Y = P$ . In the latter case, there occurs an *acc* instruction, which just makes  $q_{acc}$  active (the last target state is identified with  $q_{acc}$ ), and so the computation fails.

---

**Algorithm 3** Primality Gadget.

---

**Input variable:**  $P$

**Ensure:**  $P$  is prime.

**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:     if  $Z = P$  then acc end if
8:   end while
9:    $X++$ 
10: end while

```

---

To be able to use this in the Checking Instruction we also need the negated version. This

can be done by replacing line 7, where the last target state is identified with the target state of the gadget instead of  $q_{acc}$ , and  $q_{acc}$  is added at the end of the program.

• **Prime Number Gadget.**

This gadget takes the input variable  $U$  and outputs the variable  $P$ , whose value is the  $U$ 'th prime number. It enumerates all numbers  $P = 2, 3, 4, \dots$ , checks which are prime and counts them. When  $P$  becomes the  $U$ 'th prime number, the computation is done.

---

**Algorithm 4** Prime Number Gadget.

---

**Input variable:**  $U$

**Output variable:**  $P$

**Ensure:**  $P$  is the  $U$ 'th prime number.

**Internal variable:**  $X$

```

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

```

---

From now on we can translate a program into an NFA, which does not end in  $q_{acc}$  and ends in the last target state  $t$  whenever a word encoding a proper computation is read. Then the values of the variables in the program are correctly computed. The number of states in the NFA constructed in this way is  $O(cm)$ , where  $c$  is the length of the program measured, e.g., by the number of instructions and variables, and its alphabet has size  $\Theta(c)$ . Also, for a given program (and  $m$ ) we can construct the corresponding NFA in polynomial time.

## 4.2 An NFA with a long minimal universality length

Our first application is to show a lower bound on the maximum minimal universality length.

Alg. 5 gives the program encoding our NFA. The numbers in the brackets [ ] at the right denote the length of a word of a proper computation in the current line, which is the distance between the start and the target state of the instruction in this line; thus, this is the number of control flow states in this line excluding the last target state. The If-Else instruction in lines 4–11 is unrolled to present the lengths more visibly.

**The lengths of a proper computation.** Let us explain the construction and analyze that these lengths are correct. First, in line 1 any value for the variable  $Y$  can be chosen, and this takes exactly  $m$  letters of the Selection Gadget (Lemma 13). In line 2 the assignment to  $X$  of constant 0 is performed, which takes 1 letter. The While Instruction has the **true** condition, and therefore does not involve any test, so only the start and target states of the internal sequence are identified. In line 4 a nondeterministic branch either to line 5 or 9 is performed, which takes 1 letter. Checking in line 5 involves just one Equality Gadget, which takes  $m$  letters (Lemma 14). In line 6 we again have an assignment to  $X$  of 0, which takes 1 letter. In line 7 there is a Waiting Gadget that takes  $m + 1$  letters; there is also indicated

---

**Algorithm 5** Long minimal universality length.

---

**Internal variables:**  $X, Y$ 

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

---

that the start control flow state, which is also the target state from line 6, is final. In line 9 the Inequality Gadget takes  $m + 1$  letters (Lemma 15). In line 10 the Incrementation Gadget also takes  $m + 1$  letters. Summarizing, in a proper computation, each complete iteration of the while loop takes exactly  $2m + 3$  letters.

**The size of the NFA.** We count the number of states and the number of letters in the NFA of Alg. 5. We have two internal variables with  $2m$  states each. The Selection Gadget in line 1 has  $m$  states, since it operates directly on the internal variable  $Y$ , and it has 2 letters. The assignment in line 2 has just 1 control flow state and 1 letter. The while loop does not introduce any new states, but the target state is identified with the start state of the internal block. In line 4 we have 1 control flow state  $s$  (identified with  $t$  by the while loop) and 2 letters for branching. The Checking Instruction in line 5 has  $m$  control flow states of the Equality Gadget, which operates directly on the states of  $X$  and  $Y$ , and it has 2 letters from this gadget. In line 6 we have 1 control flow state and 1 letter, and in line 7 we have  $m + 1$  states and 1 letter. In line 9 the Checking Instruction has  $2m$  control flow states of the Inequality Gadget and it has 3 letters from this gadget. In line 10 the Incrementation Gadget has  $2m$  control flow states and 3 letters. Since the target states of line 7 and 10 are identified by the while loop with the start state of line 4, we have already counted them. Summarizing, the NFA has  $4m + m + 1 + 1 + m + 1 + (m + 1) + 2m + 2m = 11m + 4$  states and  $2 + 1 + 2 + 2 + 1 + 1 + 3 + 3 = 15$  letters.

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

**Lemma 17.** *For a given  $m$ , the NFA of Alg. 5 has minimal universality length*

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

*The number of states of this NFA is  $11m + 4$  and the size of its alphabet is 15.*

*Proof.* There are two final states in the NFA:  $q_{\text{acc}}$  and the final state indicated in line 7.

First we show that there exists a non-accepted word for every length smaller than  $(1 + \text{lcm}(1, \dots, 2^m))(2m + 3)$ . Observe that for every length there is a word  $w$  encoding a proper computation of the program; consider such a word  $w$ . Moreover, for every selection of the value of  $Y$  there exists a word  $w$  encoding a proper computation with this value. During the first  $m + 1$  letters it cannot activate any final state. In an iteration of the while loop, from the beginning of the loop to the final state  $f$  it takes exactly  $m + 2$  letters. So if  $|w| < 2m + 3$  then  $w$  is not accepted. Since every iteration 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  $w$ . So suppose that  $|w|$  is divisible by  $2m + 3$  and let  $d = |w|/(2m + 3) - 1$ . Then, 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  $|w| < (1 + \text{lcm}(1, \dots, 2^m))(2m + 3)$ , then  $d < \text{lcm}(1, \dots, 2^m)$ . So we may find  $Y$  ( $1 \leq Y \leq 2^m - 1$ ) such that  $d$  is not divisible by  $Y + 1$  (and we use  $Y = 1$  if  $d = 0$ ). The proper computation of  $w$  is such that there are  $Y$  iterations in which  $X$  is incremented and one iteration in which  $X$  is reset to 0, and this is repeated during the whole computation. Since  $d$  is not divisible by  $Y + 1$ , at the beginning of the  $d$ 'th iteration we have  $X \neq 0$ , 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  $(1 + \text{lcm}(1, \dots, 2^m))(2m + 3)$  is accepted. Suppose, contrary to what we want, that  $w$  is not accepted, which means by definition that it represents 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  $\text{lcm}(1, \dots, 2^m)$  complete such iterations must be performed. This means that, regardless of the selected value of  $Y$ , at the beginning of the last iteration we have  $X = 0$ , and there remain  $m + 2$  letters to read. Now, if  $w$  chooses the second branch, it cannot pass the test in line 9, since the test takes only  $m + 1$  letters. So  $w$  must choose the first branch, which after  $m + 2$  letters results in the final state in line 7.  $\square$

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

**Theorem 18.** *Under a 15-letter alphabet, the minimal universality length can be as large as*

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

*Proof.* We have

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

For a sufficiently large number of states  $n$  we can construct the NFA from Alg 5 for  $m = \lfloor (n - 4)/11 \rfloor$ , and add  $n - m$  unused states. Thus we obtain

$$\exp(2^m(1 + o(1))) = \exp(2^{(n-4)/11}(1 + o(1))).$$

$\square$

**Remark 19.** The constants in Lemma 17 and Theorem 18 are not optimal and can 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 Verifying the length

We are going to verify whether the length  $|w|$  satisfies some properties that provide criteria for the acceptance of  $w$ .

#### 4.3.1 Delaying

In an arbitrary program encoding our NFA whose proper computation may finish, the length of a proper computation may vary, depending on nondeterministic choices, which makes it very difficult or impossible to rely on the exact length  $|w|$ . So first we need to be able to ensure that the computation is finished after reading a known number of letters.

Consider an arbitrary program whose proper computation cannot be infinite, and so must finish in the target state  $t$  after some number of letters, though this number may be different depending on the particular word  $w$ . We encode a simple delaying program that finishes after a fixed number of letters, and create a combined program where both actual and delaying program are computed in parallel. In a proper computation, first the actual program must be finished and then the delaying program may be finished, while the actual program waits in its target state  $t$ . Hence, the combined program ends up after a fixed known number of letters (independent on  $w$ ).

Alg. 6 shows the delaying program for a fixed integer  $D$ . Clearly, a proper computation of this program takes  $O(D \cdot m)$  letters.

---

**Algorithm 6** Delaying program for a fixed number  $D$ .

---

```

1:  $X \leftarrow 0$ 
2: while  $X \neq D$  do
3:    $X++$ 
4: end while

```

---

Computation in parallel is done as follows. Let  $s_1$  and  $t_1$  be the start and the target control flow states of the actual program, and let  $s_2$  and  $t_2$  be these states of the delaying program. To the actual program we add the new letter  $\alpha_d$ , whose transition fixes  $t_1$  and the states of all variables, and maps everything else to  $q_{acc}$ . We identify  $s_1 = s_2$  and make it the start state  $s$  of the combined program. The other states of both programs are disjoint. For every pair of letters  $\alpha$  from the actual program and  $\beta$  from the delaying program, we create the letter  $\gamma_{\alpha\beta}$  that acts as  $\delta(q, \alpha) \cup \delta(q, \beta)$  for every state  $q$ , where  $\delta(q, \alpha) = \emptyset$  if  $q$  is not a state of the actual program, and similarly  $\delta(q, \beta) = \emptyset$  if  $q$  is not a state of the delaying program. We add the final target state  $t$  and the letter  $\gamma_t$ , whose transition maps both  $t_1$  and  $t_2$  to  $\{t\}$ , fixes all the variables, and maps everything else to  $q_{acc}$ .

The combined program works as follows: First both programs are computed in parallel which is encoded by letters  $\gamma_{\alpha\beta}$ . Whenever the actual program finishes in  $t_1$  while the delaying program is not yet finished in  $t_2$ , the letters  $\gamma_{\alpha_d\beta}$  (everything else maps a state to  $q_{acc}$ ) must be used until the delaying program finishes in  $t_2$ . Then we may apply  $\gamma_t$ , which results in  $t$ .

The length of a proper computation of the delaying program is  $O(m'D)$ , where  $m'$  is the used size for  $X$  and  $D$ . Note that the length a proper computation of the combined program is larger by 1 than the delaying program, because of the letter  $\gamma_t$  applied at the end.

It remains to ensure that the delaying program could always run longer than the actual program. Note that, since a proper computation of the program cannot be infinite, the length of every proper computation of the actual program is bounded by  $2^n$  (no subset of states is repeated), where  $n$  is the number of states in the NFA encoding it. Since  $n \in O(cm)$ , where  $c$  is the length of the actual program, we know that the length of every proper computation is at most  $2^{O(cm)}$ . So we can find a suitable  $D \leq 2^{O(cm)}$  such that the length of the delaying program will be at least  $2^{O(cm)}$ . Then we can find  $m' \in O(cm)$  for the size of the variable  $X$  in Alg. 6 such that  $D \leq 2^{m'} - 1$ . Thus,  $m'$  and so the number of states in the combined program is still  $O(cm)$ . Since the number of letters in the delaying program is fixed, we also have an alphabet of size  $\Theta(c)$  in the combined program.

Finally, for every given program we can construct the delayed NFA so that every proper computation finishes after a fixed number of  $D$  letters. This can be done in polynomial time, since we can always set  $D = 2^n$  and  $m' = n + 1$ , where  $n$  is the number of states of the NFA of the actual program.

#### 4.3.2 Divisibility

We are going to test whether  $|w|$  is divisible by some integers  $X_1, \dots, X_k$ . This extends the idea of Alg. 5, which just verifies whether  $(|w| - r_1)/r_2$ , for some constants  $r_1$  and  $r_2$ , is not divisible by some integer from 2 to  $2^m - 1$ .

---

**Algorithm 7** Divisibility template program.

---

**Internal variables:**  $X_1, \dots, X_k, X'_1, \dots, X'_k$ , and variable of the verifying procedure

---

```

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:
5:      $\begin{cases} \text{Delay with } D \\ \text{Verifying procedure with } X'_1, \dots, X'_k \end{cases}$   $\triangleright [T]$ 
6:   or:
7:     Delay with  $D$   $\triangleright [T - 1]$ 
8:     [final state] WAIT(1)  $\triangleright [1]$ 
9:   end choose
10:  for  $i = 1, \dots, k$  do
11:     $X \leftarrow X + 1$   $\triangleright [m + 1]$ 
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

```

---



We define a template program that is used with an arbitrary verifying procedure, whose computation passes depending on the integers  $X_1, \dots, X_k$  and the divisibility of  $(|w| - r_1)/r_2$  by these numbers. Alg. 7 shows the program. As in Alg. 5, the numbers in the brackets [ ] at the right denote the lengths of a word of a proper computation in the current line. We assume a proper computation of the verifying procedure cannot be infinite, that it does not have final states, and that it does not modify the variables  $X_1, \dots, X_k, X'_1, \dots, X'_k$ .

At the beginning (line 1) arbitrary values for the variables  $X_1, \dots, X_k$  are selected. Then the variables  $X'_1, \dots, X'_k$  are initialized to 0. Next we have the infinite while loop, which consists of two parts. First, a nondeterministic choice is made (line 4): either to run the verifying procedure or to wait. For the verifying procedure (line 5) we use the delaying trick; the verifying procedure is computed in parallel with the delaying program, which ensures that this part finishes after exactly  $T$  letters. The value of  $T$  depends on our choice for  $D$  for the delaying program, and so it depends on the verifying procedure and is at most exponential on its length. In the waiting case (line 7) we also use the delaying program with the same value of  $D$ , which finishes after  $T - 1$  of letters by the construction (see 4.3.1). Then there is a 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 does not represent an instruction, but it denotes that the body is instantiated for every  $i$  as a sequence of instructions. It can be verified that a proper computation of this part, if long enough, always takes exactly  $(2m + 3)k$  letters.

For certain lengths, every proper computation must end with either a non-final state in line 5 or in the final state in line 8. However, to end with the non-final state it must first pass the verifying procedure with  $X'_i = 0$  whenever  $X_i$  divides the number of iterations. We formalize this claim:

**Lemma 20.** *Alg. 7 accepts all words of length  $\ell$  if and only if:*

- $\ell = r_1 \cdot \ell' + r_2$ , for some constants  $r_1$  and  $r_2$ , and
- for any selection of  $X_1, \dots, X_k$ , there exists no proper computation of the verifying procedure with  $X'_i = 0$  if and only if  $X_i$  divides  $\ell'$  for all  $i$ .

*Proof.* Let  $r_1 = T + (2m + 3)k$ , which is the total length of the body in the while loop (lines 4–15), and let  $r_2 = (m + 1)k + T - 1$ , which is the length from the beginning to the final state in line 8. These values depend only on the algorithm, so on  $k$ ,  $m$ , and the verifying procedure.

Consider a length  $\ell$  that is not expressible as  $r_1 \cdot \ell' + r_2$  for any  $\ell'$ . Then a word  $w$  encoding a proper computation with the iterations of the while loop choosing every time the second branch (lines 7–8), is not accepted, since it does not finish with the final state in line 8.

Consider a word  $w$  of length  $r_1 \cdot \ell' + r_2$ . If  $w$  encodes a proper computation, then exactly  $\ell'$  complete iterations of the while loop are performed, and the last control flow state is either the final state in line 8 or a non-final state in line 5. We know that at the beginning of the last (incomplete) iteration  $X'_i = 0$  if and only if  $X_i \mid \ell'$ . So if the values for  $X_i$  can be chosen so that the verifying procedure may pass, then the computation of  $w$  may end with

a non-final state in line 5. Otherwise, it must choose the second branch in the last iteration, which results in the final state in line 8.  $\square$

### 4.3.3 Verifying the length's properties

Now we can verify the properties of the length of the read 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  $\psi$  is an arbitrary logical formula that uses  $\neg$ ,  $\wedge$ ,  $\vee$ , and whose simple propositions are of the following possible forms:

- $(X_i = c)$ , where  $c \in \{0, \dots, 2^m - 1\}$ ,
- $(X_i + X_j = X_h)$ ,
- $(X_i \cdot X_j = X_h)$ ,
- $X_i$  is prime,
- $X_i$  is the  $X_j$ 'th prime number,
- $(X_i \mid \ell')$ ,

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

We construct an NFA such that there are some constants  $r_1, r_2$  and the NFA accepts all words  $w$  of length  $\ell$  if and only if  $\ell = r_1 \cdot \ell' + r_2$  and  $\varphi(\ell')$  is not satisfied under the arithmetic bounded to integers from  $\{0, \dots, 2^m - 1\}$ , that is,  $\psi$  always evaluates to false if one of  $X_i + X_j$  or  $X_i \cdot X_j$  exceeds  $2^m - 1$ .

We use Alg. 7 with the verifying procedure checking formula  $\psi$  as follows: First, every constant  $c$  and all expressions of the forms  $X_i + X_j$ ,  $X_i \cdot X_j$ , and  $X_j$ 'th prime number that appear in  $\psi$  are computed into dedicated local variables. Note that a proper computation of this part is not possible if  $X_i + X_j$  or  $X_i \cdot X_j$  exceeds  $2^m - 1$ . Then, by the Checking Instruction, the formula  $\psi$  is verified, where we use these local variables for the corresponding expressions and where each  $(X_i \mid \ell')$  is replaced with  $(X'_i = 0)$ . Hence, by Lemma 20, there exists a word  $w$  of length  $\ell$  that is not accepted if and only if  $\psi$  is true for some values of  $X_1, \dots, X_k$  and the value of the expressions are in  $\{0, \dots, 2^m - 1\}$ .

Since the NFA for Alg. 7 and the NFA for the verifying procedure can be constructed in polynomial time, the NFA for  $\varphi(\ell')$  also can be. The size of this NFA is polynomial and the constants  $r_1$  and  $r_2$  are at most exponential in  $m$  and in the length of  $\varphi$ .

## 4.4 Reduction

We reduce from the canonical NEXPTIME-complete problem: given a nondeterministic Turing machine  $N$  on  $s$  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 still in the (unique) accepting state. 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. 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  $(r, c)$  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 computation. We 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  $\ell$  does not describe an accepting computation.

Let  $Q$  be the set of states of  $N$ , and  $q_{\text{start}}, q_{\text{acc}} \in Q$  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 numbers 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 from  $\{0, \dots, z-1\}$  for every  $(r, c)$ . We encode all these choices in one non-negative integer  $\ell'$  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  $\ell'$  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  $\ell' \leq p(1) \cdots p(2^{2s} z)$ . In the other direction, every non-negative integer  $\ell'$  represents such a choice as long as, for all  $(r, c)$ ,  $\ell'$  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  $\ell'$  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 occurs:

1. For some  $(r, c)$  and  $1 \leq i < j \leq z$ ,  $\ell'$  is divisible by both  $p((2^s r + c)z + i)$  and  $p((2^s r + c)z + j)$ .
2. For some  $(r, c)$ ,  $\ell'$  is not divisible by  $p((2^s r + c)z + i)$ , for every  $i = 1, \dots, z$ .
3.  $\ell'$  is not divisible by  $p(f(q_{\text{start}}, 0))$ .

4. For some  $c > 0$ ,  $\ell'$  is not divisible by  $p(c \cdot z + f(\text{nil}, 0))$ .
5.  $\ell'$  is not divisible by  $p((2^s(2^s - 1)z + f(q_{\text{acc}}, 0)))$ .
6. For some  $r > 0$  and  $c > 1$ , the  $2 \times 3$  window of cells with the lower-right corner at  $(r, c)$  is not legal.

Before describing how to construct such a formula, we elaborate on the last condition. As in the standard proof of NP-hardness of SAT, a  $2 \times 3$  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  $2 \times 3$  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  $r > 0$  and  $c > 1$  and  $(i_{1,1}, i_{1,2}, \dots, i_{2,3}) \notin W$ ,  $\ell'$  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. 5.

	1	$\dots$	$c-2$	$c-1$	$c$	$\dots$	$2^s$
1	$(q_{\text{start}}, 0)$	$\dots$	$(\text{nil}, 0)$	$(\text{nil}, 0)$	$(\text{nil}, 0)$	$\dots$	$(\text{nil}, 0)$
$\vdots$							
$r-1$			$t_{r-1, c-2}$	$t_{r-1, c-1}$	$t_{r-1, c}$		
$\vdots$							
$r$			$t_{r, c-2}$	$t_{r, c-1}$	$t_{r, c}$		
$\vdots$							
$2^s$	$(q_{\text{acc}}, 0)$	$\dots$	$(\text{nil}, 0)$	$(\text{nil}, 0)$	$(\text{nil}, 0)$	$\dots$	$(\text{nil}, 0)$

Figure 5: The table of tape configurations encoded by  $\ell'$  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 of the form  $\exists_{X_1, \dots, X_k \in \{0, \dots, 2^m - 1\}} \psi(X_1, \dots, X_k)$ , 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  $(r, c)$ ,  $\ell'$  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 similar reasoning.

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

**Lemma 21.**  $p(2^{2s}z) \leq 2^{11s}$ .

*Proof.* A well-known bound [16] 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  $(r, c)$ ,  $\ell'$  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  $\ell'$  does not correspond to an accepting computation of  $N$ . As was described in 4.3.3, 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  $\ell$  if and only if  $\ell = r_1 \cdots \ell' + r_2$  and  $\varphi(\ell')$  is not satisfied. Therefore, checking if  $M$  accepts all words of length  $\ell$ , for some  $\ell$ , is equivalent to checking if there exists an accepting computation of  $M$ . We state our final

**Theorem 22.** *Existential length universality for NFAs is NEXPTIME-hard.*

## References

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974, §10.6.
- [2] E. Bach and J. Shallit. *Algorithmic Number Theory*. MIT Press, 1996.
- [3] S. Cho and D. T. Huynh. The parallel complexity of finite-state automata problems. *Inform. Comput.* **97** (1992) 1–22.
- [4] 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, eds., *Computer Aided Verification, 18th International Conference, CAV 2006*, Lecture Notes in Computer Science, Vol. 4144, Springer, 2006, pp. 17–30.
- [5] K. Ellul, B. Krawetz, J. Shallit, and M.-w. Wang. Regular expressions: new results and open problems. *J. Automata, Languages and Combinatorics* **10** (2005), 407–437.
- [6] Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman. 1979.
- [7] M. Holzer. On emptiness and counting for alternating finite automata. In *Developments in Language Theory (DLT 1996)*, World Scientific, 1996, pp. 88–97.
- [8] M. Holzer and M. Kutrib. Descriptive and computational complexity of finite automata — a survey. *Inform. Comput.* **209** (2011), 456–470.
- [9] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [10] H. B. Hunt III and D. J. Rosenkrantz. On equivalence and containment problems for formal languages. *J. ACM* **24** (1977) 387–396.
- [11] H. B. Hunt III and D. J. Rosenkrantz. Computational parallels between the regular and context-free languages. *SIAM J. Comput.* **7** (1978) 99–114.
- [12] 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. *J. Comput. System Sci.* **12** (1976), 222–268.
- [13] J. Y. Kao, N. Rampersad, and J. Shallit. On NFA’s where all states are final, initial, or both. *Theoret. Comput. Sci.* **410** (2009), 5010–5021.
- [14] 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)*, IEEE Press, 1972, pp. 125–129.

- [15] D. Rosenblatt. On the graphs and asymptotic forms of finite Boolean relation matrices. *Naval Research Quart.* **4** (1957), 151–167.
- [16] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Ill. J. Math.* **6** (1962), 64–94.
- [17] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. *Symposium on Theory of Computing (STOC 1973)*, ACM Press, 1973, pp. 1–9.