Extended Regular Expressions: Succinctness and Decidability

Dominik D. Freydenberger

Institut für Informatik, Goethe Universität, Frankfurt am Main, Germany
freydenberger@em.uni-frankfurt.de

Abstract

Most modern implementations of regular expression engines allow the use of variables (also called back references). The resulting extended regular expressions (which, in the literature, are also called practical regular expressions, rewbr, or regex) are able to express non-regular languages.

The present paper demonstrates that extended regular expressions cannot be minimized effectively (neither with respect to length, nor number of variables), and that the tradeoff in size between extended and “classical” regular expressions is not bounded by any recursive function. In addition to this, we prove the undecidability of several decision problems (universality, equivalence, inclusion, regularity, and cofiniteness) for extended regular expressions. Furthermore, we show that all these results hold even if the extended regular expressions contain only a single variable.

1998 ACM Subject Classification F.4.3: Formal Languages

Keywords and phrases extended regular expressions, regex, decidability, non-recursive tradeoffs

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Since being introduced by Kleene [18] in 1956, regular expressions have developed into a central device of theoretical and applied computer science. On one side, research into the theoretical properties of regular expressions, in particular various aspects of their complexity, is still a very active area of investigation (see Holzer and Kutrib [16] for a survey with numerous recent references). On the other side, almost all modern programming languages offer regular expression matching in their standard libraries or application frameworks, and most text editors allow the use of regular expressions for search and replacement functionality.

But, due to practical considerations (cf. Friedl [13]), most modern matching engines have evolved to use an extension to regular expressions that allows the user to specify non-regular languages. In addition to the features of regular expressions as they are mostly studied in theory (which we, from now on, call proper regular expressions), and apart from the (regularity preserving) “syntactic sugar” that most implementations use, these extended regular expressions contain back references, also called variables, which specify repetitions that increase the expressive power beyond the class of regular languages. For example, the (non-regular) language \( L = \{ ww \mid w \in \{a, b\}^* \} \) is generated by the extended regular expression \( \alpha := (a \mid b)^*%x.x \).

This expression can be understood as follows (for a more formal treatment, see Definition 3): For any expression \( \beta \), \( (\beta)%x \) matches the same expression as \( \beta \), and binds the match to the variable \( x \). In the case of this example, the subexpression \( (a \mid b)^*%x \) can be matched to any word \( w \in \{a, b\}^* \), and when it is matched to \( w \), the variable \( x \) is assigned the value \( w \). Any further occurrence of \( x \) repeats \( w \), leading to the language of all words of
the form $ww$ with $w \in \{a, b\}^*$. Analogously, the expression\((a | b)^*\%x xx\) generates the language of all $www$ with $w \in \{a, b\}^*$.

Although this ability to specify repetitions is used in almost every modern matching engine (e.g., the programming languages PERL and Python), the implementations differ in various details, even between two versions of the same implementation of a programming language (for some examples, see Câmpeanu and Santean [6]). Nonetheless, there is a common core to these variants, which was first formalized by Aho [1], and later by Câmpeanu, Salomaa and Yu [5]. Still, theoretical investigation of extended regular expressions has been comparatively rare (in particular when compared to their more prominent subclass); see e.g., Larsen [20], Della Penna et al. [12], Câmpeanu and Santean [6], Carle and Narendran [8], and Reidenbach and Schmid [24].

In contrast to their widespread use in various applications, extended regular expressions have some undesirable properties. Most importantly, their membership problem (the question whether an expression matches a word) is NP-complete (cf. Aho [1]); the exponential part in the best known upper bounds depends on the number of different variables in the expression. Of course, this compares unfavorably to the efficiently decidable membership problem of proper regular expressions (cf. Aho [1]). On the other hand, there are cases where extended regular expressions express regular languages far more succinctly than proper regular expressions. Consider the following example:

▶ Example 1. For $n \geq 1$, let $L_n := \{www \mid w \in \{a, b\}^+, |w| = n\}$. These languages $L_n$ are finite, and hence, regular. With some effort\(^1\), one can prove that every proper regular expression for $L_n$ is at least of length exponential in $n$. In contrast to this, $L_n$ is generated by the extended regular expression

$$\alpha_n := ((a | b) \cdots (a | b))\%x xx,$$

which is of a length that is linear in $n$.\(\Diamond\)

Due to the repetitive nature of the words of languages $L_n$ in Example 1, it is not surprising that the use of variables provides a shorter description of $L_n$. The following example might be considered less straightforward:

▶ Example 2. Consider the expression $\alpha := (a | b)^+((a | b)^+\%x x(a | b)^+)$. It is a well-known fact that every word $w \in \{a, b\}^*$ with $|w| \geq 4$ can be expressed in the form $w = uxxv$, with $u, v \in \{a, b\}^*$ and $x \in \{a, b\}^+$ (as is easily verified by examining all four letter words). Thus, the expression $\alpha$ matches all but finitely many words; hence, its language $L(\alpha)$ is regular.\(\Diamond\)

The phenomenon used in Example 2 is strongly related to the notion of *avoidable patterns* (cf. Cassaigne [9]), and involves some very hard combinatorial questions. We observe that extended regular expressions can be used to express regular languages more succinctly than proper regular expressions do, and that it might be hard to convert an extended regular expression into a proper regular expression for the same language.

The two central questions studied in the present paper are as follows: First, how hard is it to minimize extended regular expressions (both with respect to their length, and with respect to the number of variables they contain), and second, how succinctly can extended

\(^1\) One can show this by proving that every NFA for $L_n$ requires at least $O(2^n)$ states, e.g., by using the technique by Glaister and Shallit [14]. Due to the construction used in the proof of Theorem 2.3 in [17], this also gives a lower bound on the length of the regular expressions for $L_n$.\(\)
regular expressions describe regular languages? These natural questions are also motivated
by practical concerns: If a given application reuses an expression many times, it might pay
off to invest resources in the search for an expression that is shorter, or uses fewer variables,
and thus can be matched more efficiently.

We approach this question through related decidability problems (e.g., the universality
problem) and by studying lower bounds on the tradeoff between the size of extended regular
expressions and proper regular expressions.

The main contribution of the present paper is the proof that all these decision problems are
undecidable (some are not even semi-decidable), even for extended regular expressions that
use only a single variable. Thus, while bounding the number of variables in extended regular
expressions (or, more precisely, the number of variable bindings) reduces the complexity of the
membership problem from NP-complete to polynomial (cf. Aho [1]), we show that extending
proper regular expressions with only a single variable already results in undecidability of
various problems.

As a consequence, extended regular expressions cannot be minimized effectively, and the
tradeoff between extended and proper regular expressions is not bounded by any recursive
function (a so-called non-recursive tradeoff, cf. Kutrib [19]). Thus, although the use of the
“right” extended regular expression for a regular expression might offer arbitrary advantages
in size (and, hence, parsing speed), these optimal expressions cannot be found effectively.
These results highlight the power of the variable mechanism, and demonstrate that different
restrictions than the number of variables ought to be considered.

The structure of the further parts of the paper is as follows: In Section 2, we introduce
most of the technical details. Section 3 consists of Theorem 9 – the main undecidability
result – and its consequences, while Section 4 contains the proof of Theorem 9 and technical
groundwork needed for that proof. The paper is concluded by Section 5. Due to space
reasons, most technical details were moved to the Appendix.

2 Preliminaries

This paper is largely self-contained. Unexplained notions can be found in Hopcroft and
Ullman [17], Cutland [11], and Minsky [21].

2.1 Basic Definitions

Let \( \mathbb{N} \) be the set of natural numbers, including 0. The function \( \text{div} \) denotes integer division,
and \( \text{mod} \) denotes its remainder (e.g., \( 5 \text{ div } 3 = 1 \) and \( 5 \text{ mod } 3 = 2 \)). We denote the empty
string by \( \lambda \). For the concatenation of two strings \( w_1 \) and \( w_2 \), we write \( w_1 \cdot w_2 \) or simply
\( w_1w_2 \). We say a string \( v \in A^* \) is a factor of a string \( w \in A^* \) if there are \( u_1, u_2 \in A^* \) such
that \( w = u_1vu_2 \). The notation \( |K| \) stands for the size of a set \( K \) or the length of a string \( K \).

If \( A \) is an alphabet, a (one-sided) infinite word over \( A \) is an infinite sequence \( w = (w_i)_{i=0}^{\infty} \)
with \( w_i \in A \) for every \( i \geq 0 \). We denote the set of all one-sided infinite words over \( A \) by
\( A^\omega \). For a string \( v \in A^* \), let \( a^v \) denote the word \( w = (w_i)_{i=0}^{\infty} \) with \( w_i = a \) for every \( i \geq 0 \).
We shall only deal with infinite words \( w \in A^\omega \) that have the form \( w = u a^v \) with \( u \in A^\ast \)
and \( a \in A \). Concatenation of words and infinite words is defined canonically: For every
\( u \in A^* \) and every \( v \in A^\omega \) with \( v = (v_i)_{i=0}^{\infty} \), \( u \cdot v := w \in A^\omega \), where \( w_0 \cdot \ldots \cdot w_{|a|-1} = u \) and
\( w_{|a|+i} = v_i \) for every \( i \geq 0 \), while \( vu \) is undefined. In particular, note that \( a a^v = a^v \) for
every \( a \in A \).
2.2 Extended Regular Expressions

We now introduce syntax and semantics of extended regular expressions. Apart from some changes in terminology, this formalization is due to Aho [1]:

- **Definition 3.** Let $\Sigma$ be an infinite set of terminals, let $X$ be an infinite set of variables, and let the set of metacharacters consist of $\lambda$, $( )$, $|$, $\ast$, and $\%$, where all three sets are pairwise disjoint. We define extended regular expressions inductively as follows:

1. Each $a \in \Sigma \cup \{\lambda\}$ is an extended regular expression that matches the word $a$.
2. Each $x \in X$ is an extended regular expression that matches the word to which $x$ is bound.
3. If $\alpha_1$ and $\alpha_2$ are extended regular expressions, then $(\alpha_1 \mid \alpha_2)$ is an extended regular expression that matches any word matched by $\alpha_1$ or by $\alpha_2$.
4. If $\alpha_1$ and $\alpha_2$ are extended regular expressions, then $(\alpha_1 \alpha_2)$ is an extended regular expression that matches any word of the form $vw$, where $v$ matches $\alpha_1$ and $w$ matches $\alpha_2$.
5. If $\alpha$ is an extended regular expression, then $(\alpha)^+$ is an extended regular expression that matches any word of the form $w_1 \cdots w_n$ with $n \geq 0$, where $\alpha$ matches each $w_i$ with $1 \leq i \leq n$.
6. If $\alpha$ is an extended regular expression that matches a word $w$, and $x \in X$, then $(\alpha)\%x$ is an extended regular expression that matches the word $w$, and $x$ is bound to the value $w$.

We denote the set of all extended regular expressions by RegEx. For every extended regular expression $\alpha$, we use $L(\alpha)$ to denote the set of all words that are matched by $\alpha$, and call $L(\alpha)$ the language generated by $\alpha$. A proper regular expression is an extended regular expression that contains neither $\%$, nor any variable.

Note that, as in [1], some peculiarities of the semantics of extended regular expressions are not addressed in this definition (some examples are mentioned further down). Câmpeanu et al. [5] offer an alternative definition that explicitly deals with some technical peculiarities that are omitted in Aho’s definition, and is closer to the syntax of the programming language PERL. The proofs presented in this paper are not affected by these differences and can be easily adapted to the definition of Câmpeanu et al., or any similar definition, like the models proposed by Bordihn et al. [3], Câmpeanu and Yu [7].

We shall use the notation $(\alpha)^+$ as a shorthand for $\alpha(\alpha)^+$, and freely omit parentheses whenever the meaning remains unambiguous. When doing this, we assume that there is a precedence on the order of the applications of operations, with $\ast$ and $\%$ ranking over concatenation ranking over the alternation operator $|$.

We illustrate the intended semantics of extended regular expressions using the following examples in addition to the examples in Section 1:

- **Example 4.** Consider the following extended regular expressions:

  $$\alpha_1 := ((a \mid b)^\ast) \%x xx ((a \mid b)^\ast) \%x x, \quad \alpha_2 := ((a \mid b)^\ast) \%x x)^\ast.$$ 

  These expressions generate the following languages:

  $$L(\alpha_1) = \{vvww \mid v, w \in \{a, b\}^\ast\}, \quad L(\alpha_2) = \{w_1w_2 \cdots w_nw_n \mid n \geq 1, w_i \in \{a, b\}^\ast\}.$$ 

  Note that both expressions rely on the fact that variables can be bound multiple times, and implicitly assume that we parse from left to right. $\diamond$
From a formal point of view, the fact that variables have a scope and the possibility to rebind variables (as in Example 4) can cause unexpected side effects and would normally require a more formal definition of semantics, instead of our “definition by example”. Furthermore, Aho’s definition does not deal with pathological cases in which some variables might be unbound, e.g., like \((a\%)x \mid b\ x\). Although the definition by Câmpeanu et al. [5] addresses these problems, we still use Aho’s notation, because it is more convenient for the proof of Theorem 19, the main technical tool of the present paper. As mentioned above, all proofs can be easily adapted to the notation and semantics of Câmpeanu et al.

In general, the membership problem for RegEx is NP-complete, as shown in Theorem 6.2 in Aho [1]. As explained in that proof, this problem is solvable in polynomial-time if the number of different variables is bounded. It is not clear how (or if) Aho’s reasoning applies in Aho [1]. As explained in that proof, this problem is solvable in polynomial-time if the number of different variables is bounded. It is not clear how (or if) Aho’s reasoning applies in Aho [1].

**Definition 5.** For \(k \geq 0\), let \(\text{RegEx}(k)\) denote the class of all extended regular expressions \(\alpha\) that satisfy the following properties:

1. \(\alpha\) contains at most \(k\) occurrences of the metacharacter \(\%\),
2. if \(\alpha\) contains a subexpression \((\beta)^*\), then the metacharacter \(\%\) does not occur in \(\beta\),
3. for every \(x \in \Sigma\) that occurs in \(\alpha\), \(\alpha\) contains exactly one occurrence of \(\%x\).

Intuitively, these restrictions on extended regular expressions in \(\text{RegEx}(k)\) limit not only the number of different variables, but also the total number of possible variable bindings, to at most \(k\).

Note that \(\text{RegEx}(0)\) is equivalent to the class of proper regular expressions; furthermore, observe that \(\text{RegEx}(k) \subseteq \text{RegEx}(k + 1)\) for every \(k \geq 0\).

Referring to the extended regular expressions given in Example 4, we observe that, as \(\%x\) occurs twice in \(\alpha_1\), \(\alpha_1\) is not element of any \(\text{RegEx}(k)\) with \(k \geq 0\), but the extended regular expression \(\alpha_1' := ((a \mid b)^\%x xx ((a \mid b)^\%) \%y y\) generates the same language as \(\alpha_1\), and \(\alpha_1' \in (\text{RegEx}(2) \setminus \text{RegEx}(1))\). In contrast to this, \(\alpha_2 \notin \text{RegEx}(k)\) for all \(k \geq 0\), as \(\%\) occurs inside a \((^\ast\) subexpression (as we defined \(^\ast\) through \(^\ast\)).

For any \(k \geq 0\), we say that a language \(L\) is a \(\text{RegEx}(k)\)-language if there is some \(\alpha \in \text{RegEx}(k)\) with \(L(\alpha) = L\).

We also consider the class \(\text{FRegEx}\) of all extended regular expressions that do not use the operator \(^*\) (or \(^\ast\)), and its subclasses \(\text{FRegEx}(k) := \text{FRegEx} \cap \text{RegEx}(k)\) for \(k \geq 0\). Thus, \(\text{FRegEx}\) contains exactly those expressions that generate finite (and, hence, regular) languages. Analogously, for every \(k \geq 0\), we define a class \(\text{CoFRegEx}(k)\) as the class of all \(\alpha \in \text{RegEx}(k)\) such that \(L(\alpha)\) is cofinite. Unlike the classes \(\text{FRegEx}(k)\), these classes have no straightforward syntactic definition – as we shall prove in Theorem 9, cofiniteness is not semi-decidable for \(\text{RegEx}(k)\) (if \(k \geq 1\)).

### 2.3 Decision Problems and Descriptive Complexity

Most of the technical reasoning in the present paper is centered around the following decision problems:

**Definition 6.** Let \(\Sigma\) denote a fixed terminal alphabet. For all \(k, l \geq 0\), we define the following decision problems for \(\text{RegEx}(k)\):

- **Universality** Given \(\alpha \in \text{RegEx}(k)\), is \(L(\alpha) = \Sigma^*\)?
- **Cofiniteness** Given \(\alpha \in \text{RegEx}(k)\), is \(\Sigma^* \setminus L(\alpha)\) finite?
- **RegEx\((l)\)-ity** Given \(\alpha \in \text{RegEx}(k)\), is there a \(\beta \in \text{RegEx}(l)\) with \(L(\alpha) = L(\beta)\)?
As we shall see, Theorem 9 – one of our main technical results – states that these problems are undecidable (to various degrees). We use the undecidability of the universality problem to show that there is no effective procedure that minimizes extended regular expressions with respect to their length, and the undecidability of RegEx(l)-ity to conclude the same for minimization with respect to the number of variables. Furthermore, cofiniteness and RegEx(l)-ity help us to obtain various results on the relative succinctness of proper and extended regular expressions.

By definition, RegEx(l)-ity holds trivially for all RegEx(k) with $k \leq l$. If $l = 0$, we mostly use the more convenient term regularity (for RegEx(k)), instead of RegEx(0)-ity. Note that, even for RegEx(0), universality is already PSPACE-complete (see Aho et al. [2]).

In order to examine the relative succinctness of RegEx(1) in comparison to RegEx(0), we use the following notion of complexity measures:

▶ Definition 7. Let $\mathcal{R}$ be a class of extended regular expressions. A complexity measure for $\mathcal{R}$ is a total recursive function $c: \mathcal{R} \rightarrow \mathbb{N}$ such that, for every alphabet $\Sigma$, the set of all $\alpha \in \mathcal{R}$ with $L(\alpha) \subseteq \Sigma^*$ can be effectively enumerated in order of increasing $c(\alpha)$, and 2. does not contain infinitely many extended regular expressions with the same value $c(\alpha)$.

This definition includes the canonical concept of the length, as well as most of its natural extensions (for example, in our context, one could define a complexity measure that gives additional weight to the number or distance of occurrences of variables, or their nesting level). Kutrib [19] provides more details on (and an extensive motivation of) complexity measures. Using this definition, we are able to define the notion of tradeoffs between classes of extended regular expressions:

▶ Definition 8. Let $k > l \geq 0$ and let $c$ be a complexity measure for RegEx(k) (and thereby also for RegEx(l)). A recursive function $f_c: \mathbb{N} \rightarrow \mathbb{N}$ is said to be a recursive upper bound for the tradeoff between RegEx(k) and RegEx(l) if, for all those $\alpha \in$ RegEx(k) for which $L(\alpha)$ is a RegEx(l)-language, there is a $\beta \in$ RegEx(l) with $L(\beta) = L(\alpha)$ and $c(\beta) \leq f_c(c(\alpha))$.

If no recursive upper bound for the tradeoff between RegEx(k) and RegEx(l) exists, we say that the tradeoff between RegEx(k) and RegEx(l) is non-recursive.

There is a considerable amount of literature on a wide range of non-recursive tradeoffs between various description mechanisms; for a survey, see Kutrib [19].

3 Main Results

As mentioned in Section 1, the central questions of the present paper are whether we can minimize extended regular expressions (under any complexity measure as defined in Definition 7, or with respect to the number variables), and whether there is a recursive upper bound on the tradeoff between extended and proper regular expressions. We approach these questions by proving various degrees of undecidability for the decision problems given in Definition 6, as shown in the main theorem of this section:

▶ Theorem 9. For RegEx(1), universality is not semi-decidable; and regularity and cofiniteness are neither semi-decidable, nor co-semi-decidable.

The proof of Theorem 9 requires considerable technical preparation and can be found in Section 4.

Of course, all these undecidability results also hold for every RegEx(k) with $k \geq 2$, and for the whole class RegEx of extended regular expressions (as RegEx(1) is contained in all
these classes). Theorem 9 also demonstrates that inclusion and equivalence are undecidable for RegEx(1) (and, hence, all of RegEx). We also see, as an immediate consequence to Theorem 9, that there is no algorithm that minimizes the number of variables in an extended regular expression, as such an algorithm could be used to decide regularity.

Note that in the proof of Theorem 9, the single variable $x$ is bound only to words that match the expression $0^*$. This shows that the “negative” properties of extended regular expressions we derive from Theorem 9 hold even if we restrict RegEx(1) by requiring that the variable can only be bound to a very restricted proper regular expression. Furthermore, the proof also applies to the extension of proper regular expressions through numerical parameters that is proposed in Della Penna et al. [12]. In addition to this, the construction from Theorem 19 (which we shall use to prove Theorem 9, and consequently, all other results in the present paper) can be refined to also include bounds on the number of occurrences of the single variable – see Section A.11 in the Appendix.

From the undecidability of universality, we can immediately conclude that RegEx(1) cannot be minimized effectively (for details, see Section A.1 in the Appendix):

▶ Corollary 10. Let $c$ be a complexity measure for RegEx(1). Then there is no recursive function $m_c$ that, given an expression $\alpha \in \text{RegEx}(1)$, returns an expression $m_c(\alpha) \in \text{RegEx}(1)$ with 1. $L(m_c(\alpha)) = L(\alpha)$, and 2. $c(\beta) \geq c(m_c(\alpha))$ for every $\beta \in \text{RegEx}(1)$ with $L(\beta) = L(\alpha)$.

Following the classic proof method of Hartmanis [15] (cf. Kutrib [19]), we can use the fact that non-regularity is not semi-decidable to obtain a result on the relative succinctness of extended and proper regular expressions (see Section A.2 in the Appendix for details):

▶ Corollary 11. There are non-recursive tradeoffs between RegEx(1) and RegEx(0). This holds even if we consider only the tradeoffs between CoFRegEx(1) and CoFRegEx(0), using a complexity measure for RegEx(1).

Thus, no matter which complexity measure and which computable upper bound we assume for the tradeoff, there is always a regular language $L$ that can be described by an extended regular expression from RegEx(1) so much more succinctly that every proper regular expression for $L$ has to break that bound. Obviously, this has also implications for the complexity of matching regular expressions: Although membership is “easier” for proper regular expressions than for extended regular expressions, there are regular languages that can be expressed far more efficiently through extended regular expressions than through proper regular expressions.

Recall Example 1, where we consider extended regular expressions that describe finite languages. In this restricted case, there exists an effective conversion procedure – hence, the tradeoffs are recursive (cf. Section A.3 in the Appendix):

▶ Lemma 12. For every $k \geq 1$, the tradeoff between $F\text{RegEx}(k)$ and $F\text{RegEx}(0)$ is recursive (even when considering complexity measures for RegEx($k$) instead of $F\text{RegEx}(k)$).

Although the class of RegEx-languages is not closed under complementation (Lemma 2 in Câmpeanu et al. [5]), there are languages $L$ such that both $L$ and its complement $\Sigma^* \setminus L$ are RegEx-languages (e.g., all regular languages). Combining Lemma 12 and Corollary 11, we can straightforwardly conclude that there are cases where it is far more efficient to describe the complement of a RegEx(1)-language, as opposed to the language itself (cf. Section A.4 in the Appendix):

2 Note that cofiniteness for extended regular expressions is a more general case of the question whether a pattern is avoidable over a fixed terminal alphabet, an important open problem in pattern avoidance (cf. Currie [10]). Example 2 illustrates this connection for the pattern $xx$ over a binary alphabet.
Extended Regular Expressions: Succinctness and Decidability

**Corollary 13.** Let $\Sigma$ be a finite alphabet. Let $c$ be a complexity measure for $\text{RegEx}(1)$. For any recursive function $f_c : \mathbb{N} \to \mathbb{N}$, there exists an $\alpha \in \text{RegEx}(1)$ such that $\Sigma^* \setminus L(\alpha)$ is a $\text{RegEx}(1)$-language, and for every $\beta \in \text{RegEx}(1)$ with $L(\beta) = \Sigma^* \setminus L(\alpha)$, $c(\beta) \geq f_c(c(\alpha))$.

With some additional technical effort, we can extend the previous results on undecidability of $\text{RegEx}(l)$-ity and on tradeoffs between $\text{RegEx}(k)$ and $\text{RegEx}(0)$ to arbitrary levels of the hierarchy of $\text{RegEx}(k)$-languages:

**Lemma 14.** Let $k \geq 1$. For $\text{RegEx}(k + 1)$, both $\text{RegEx}(k)$-ity and its complement are not semi-decidable.

In this proof, we concatenate the languages from the proof of Theorem 9 with languages that are $\text{RegEx}(k + 1)$-languages, but not $\text{RegEx}(k)$-languages; see Section A.5 in the Appendix. Non-recursive tradeoffs between $\text{RegEx}(k + 1)$ and $\text{RegEx}(k)$ for every $k \geq 1$ follow immediately, using Hartmanis’ proof technique as in the proof of Corollary 11.

## 4 Proof of Theorem 9

On a superficial level, we prove Theorem 9 by using Theorem 19 (which we introduce further down in the present section) to reduce various undecidable decision problems for Turing machines to appropriate problems for extended regular expressions (the problems from Definition 6). This is done by giving an effective procedure that, given a Turing machine $\mathcal{M}$, returns an extended regular expression that generates the complement of a language that encodes all accepting runs of $\mathcal{M}$.

On a less superficial level, this approach needs to deal with certain technical peculiarities that make it preferable to study a variation of the Turing machine model. An extended Turing machine is a 3-tuple $\mathcal{X} = (Q, q_1, \delta)$, where $Q$ and $q_1$ denote the state set and the initial state. All extended Turing machines operate on the tape alphabet $\Gamma := \{0, 1\}$ and use 0 as the blank letter. The transition function $\delta$ is a function $\delta : \Gamma \times Q \rightarrow (\Gamma \times \{L, R\} \times Q) \cup \{\text{HALT}\} \cup (\{\text{CHECK}_R\} \times Q)$. The movement instructions $L$ and $R$ and the HALT-instruction are interpreted canonically – if $\delta(a, q) = (b, M, p)$ for some $M \in \{L, R\}$ (and $a, b \in \Gamma$, $p, q \in Q$), the machine replaces the symbol under the head ($a$) with $b$, moves the head to the left if $M = L$ (or to the right if $M = R$), and enters state $p$. If $\delta(a, q) = \text{HALT}$, the machine halts and accepts.

The command CHECKR works as follows: If $\delta(a, q) = (\text{CHECK}_R, p)$ for some $p \in Q$, $\mathcal{X}$ immediately checks (without moving the head) whether the right side of the tape (i.e., the part of the tape that starts immediately to the right of the head) contains only the blank symbol 0. If this is the case, $\mathcal{X}$ enters state $p$; but if the right side of the tape contains any occurrence of 1, $\mathcal{X}$ stays in $q$. As the tape is never changed during a CHECKR-instruction, this leads $\mathcal{X}$ into an infinite loop, as it will always read $a$ in $q$, and will neither halt, nor change its state, head symbol, or head position. Although it might seem counterintuitive to include an instruction that allows our machines to search the whole infinite side of a tape in a single step and without moving the head, this command is expressible in the construction we use in the proof of Theorem 19, and it is needed for the intended behavior.

We partition the tape of an extended Turing machine $\mathcal{X}$ into three disjoint areas: The head symbol, which is (naturally) the tape symbol at the position of the head, the right tape side, which contains the tape word that starts immediately to the right of the head symbol and extends rightward into infinity, and the left tape side, which starts immediately left to the head symbol and extends infinitely to the left. When speaking of a configuration, we denote the head symbol by $a$ and refer to the contents of the left or right tape side as the left tape...
word $t_L$ or the right tape word $t_R$, respectively. For an illustration and further explanations, see Figure 2 (Section A.6 in the Appendix). A configuration of an extended Turing machine $\mathcal{X} = (Q, q_1, \delta)$ is a tuple $(t_L, t_R, a, q)$, where $t_L, t_R \in \Gamma^*0^\omega$ are the left and right tape word, $a \in \Gamma$ is the head symbol, and $q \in Q$ denotes the current state. The symbol $\Gamma_X$ denotes the successor relation on configurations of $\mathcal{X}$, i.e., $C \vdash_X C'$ if $\mathcal{X}$ enters $C'$ immediately after $C$.

We define $\text{dom}_X(\mathcal{X})$, the domain of an extended Turing machine $\mathcal{X} = (Q, q_1, \delta)$, to be the set of all tape words $t_R \in \Gamma^*0^\omega$ such that $\mathcal{X}$, if started in the configuration $(0^\omega, t_R, 0, q_1)$, halts after finitely many steps.

The definition of $\text{dom}_X$ is motivated by the properties of the encoding that we shall use. Usually, definitions of the domain of a Turing machine rely on the fact that the end of the input is marked by a special letter $\$ or an encoding thereof (cf. Minsky [21]). As we shall see, our use of extended regular expressions does not allow us to express the fact that every input is ended by exactly one $\$ symbol. Without the CHECKR-instruction in an extended Turing machine $\mathcal{X}$, we then would have to deal with the unfortunate side effect that a nonempty $\text{dom}_X(\mathcal{X})$ could never be finite: Assume $v \in \Gamma^*$ such that $v0^\omega \in \text{dom}_X(\mathcal{X})$. The machine can only see a finite part of the right side of the tape before accepting. Thus, there is a $v \in \Gamma^*$ such that both $v10^\omega \in \text{dom}_X(\mathcal{X})$ and $v00^\omega \in \text{dom}_X(\mathcal{X})$, as $\mathcal{X}$ will not reach the part where $v1$ and $v0$ differ. This observation leads to $v10^\omega \in \text{dom}_X(\mathcal{X})$ for every $x \in \Gamma^*$, and applies to various other extensions of the Turing machine model. As Lemma 18 – and thereby most of the main results in Section 3 – crucially depends on the fact that there are extended Turing machines with a finite domain, we use CHECKR to allow our machines to perform additional sanity checks on the input and to overcome the limitations that arise from the lack of the input markers $\$ and $\$.

Using a classical coding technique for two-symbol Turing machines (see Minsky [21]) and the correspond undecidability results, we establish the following negative results on decision problems for extended Turing machines (for details, see Section A.7 in the Appendix):

**Lemma 15.** Consider the following decision problems for extended Turing machines:

**Emptiness** Given an extended Turing machine $\mathcal{X}$, is $\text{dom}_X(\mathcal{X})$ empty?

**Finiteness** Given an extended Turing machine $\mathcal{X}$, is $\text{dom}_X(\mathcal{X})$ finite?

Then emptiness is not semi-decidable, and finiteness is neither semi-decidable, nor co-semi-decidable.

In order to simplify some technical aspects of our further proofs below, we adopt the following convention on extended Turing machines:

**Convention 16.** Every extended Turing machine

1. has the state set $Q = \{q_1, \ldots, q_\nu\}$ for some $\nu \geq 1$, where $q_1$ is the initial state,
2. has $\delta(0, q_1) = (0, L, q_2),$
3. has $\delta(a, q) = \text{HALT}$ for at least one pair $(a, q) \in \Gamma \times Q$.

Obviously, every extended Turing machine can be straightforwardly (and effectively) adapted to satisfy these criteria.

As every tape word contains only finitely many occurrences of 1, we can interpret tape sides as natural numbers in the following (canonical) way: For sequences $t = (t_i)_{i=0}^\infty$ over $\Gamma$, define $e(t) := \sum_{i=0}^\infty 2^i e(t_i)$, where $e(0) := 0$ and $e(1) := 1$. Most of the time, we will not distinguish between single letters and their values under $e$, and simply write $a$ instead of $e(a)$ for all $a \in \Gamma$. It is easily seen that $e$ is a bijection between $\mathbb{N}$ and $\Gamma^*0^\omega$, the set of all tape words over $\Gamma$. Intuitively, every tape word is read as a binary number, starting with the cell closest to the head as the least significant bit, extending toward infinity.
Expressing the three parts of the tape (left and right tape word and head symbol) as natural numbers allows us to compute the tape parts of successor configurations using elementary integer operations. The following straightforward observation shall be a very important tool in the proof of Theorem 19:

**Observation 17.** Assume that an extended Turing machine $\mathcal{X} = (Q, q_1, \delta)$ is in some configuration $C = (t_L, t_R, a, q_i)$, and $\delta(a, q_i) = (b, M, q_j)$ for some $b \in \Gamma$, some $M \in \{L, R\}$ and some $q_j \in Q$. For the (uniquely defined) successor configuration $C' = (t'_L, t'_R, a', q_j)$ with $C \vdash_C C'$, the following holds:

If $M = L$: 
$$e(t'_L) = e(t_L) \div 2, \quad e(t'_R) = 2e(t_R) + b, \quad a' = e(t_L) \mod 2,$$

if $M = R$: 
$$e(t'_L) = 2e(t_L) + b, \quad e(t'_R) = e(t_R) \div 2, \quad a' = e(t_R) \mod 2.$$

These equations are fairly obvious – when moving the head in direction $M$, $\mathcal{X}$ turns the tape cell that contained the least significant bit of $e(t_M)$ into the new head symbol, while the other tape side gains the tape cell containing the new letter $b$ that was written over the head symbol as new least significant bit.

Using the encoding $e$, we define an encoding $\text{enc}$ of configurations of $\mathcal{X}$ by

$$\text{enc}(t_L, t_R, a, q_i) := 00^i(\text{enc}(t_L))\#00^i(\text{enc}(t_R))\#00^i$$

for every configuration $(t_L, t_R, a, q_i)$ of $\mathcal{X}$. We extend $\text{enc}$ to an encoding of finite sequences $C = (C_i)_{i=1}^n$ (where every $C_i$ is a configuration of $\mathcal{X}$) by

$$\text{enc}(C) := \#\#\text{enc}(C_1)\#\text{enc}(C_2)\#\cdots\#\text{enc}(C_n)\#.$$

A valid computation of $\mathcal{X}$ is a sequence $C = (C_i)_{i=1}^n$ of configurations of $\mathcal{X}$ where $C_1$ is an initial configuration (i.e., some configuration $(0^\omega, w, 0, q_i)$ with $w \in \Gamma^\omega0^\omega$), $C_n$ is a halting configuration, and for every $i < n$, $C_i \vdash_C C_{i+1}$. Thus, let

$$\text{VALC}(\mathcal{X}) = \{\text{enc}(C) \mid C \text{ is a valid computation of } \mathcal{X}\},$$

$$\text{INVALC}(\mathcal{X}) = \{0, \#\}^* \setminus \text{VALC}(\mathcal{X}).$$

The main part of the proof of Theorem 9 is Theorem 19 (still further down), which states that, given an extended Turing machine $\mathcal{X}$, one can effectively construct an expression from $\text{RegEx}(1)$ that generates $\text{INVALC}(\mathcal{X})$. Note that in $\text{enc}(C)$, $\#\#$ serves as a boundary between the encodings of individual configurations, which will be of use in the proof of Theorem 19. Building on Convention 16, we observe the following fact on the regularity of $\text{VALC}(\mathcal{X})$ for a given extended Turing machine $\mathcal{X}$:

**Lemma 18.** For every extended Turing machine $\mathcal{X}$, $\text{VALC}(\mathcal{X})$ is regular if and only if $\text{dom}_{\mathcal{X}}(\mathcal{X})$ is finite.

The if direction is obvious, the only if direction follows from Convention 16 and the application of a generalized sequential machine. For details, see Section A.8 in the Appendix. We are now ready to state the central part of our proof of Theorem 9:

**Theorem 19.** For every extended Turing machine $\mathcal{X}$, one can effectively construct an extended regular expression $\alpha_{\mathcal{X}} \in \text{RegEx}(1)$ such that $L(\alpha_{\mathcal{X}}) = \text{INVALC}(\mathcal{X})$.

In the present section, we only sketch the proof of Theorem 19; its full version can be found in Section A.9 in the Appendix. Given an extended Turing machine $\mathcal{X}$, the expression $\alpha_{\mathcal{X}}$ can be assembled from various subexpressions that describe a complete list of sufficient criteria.
for membership in $\text{INVALC}(\mathcal{X})$. Intuitively, every word in $\text{INVALC}(\mathcal{X})$ contains (at least) one error. We first consider so-called structural errors, where a word is not an encoding of any sequence $(C_i)_{i=1}^{n}$ over configurations of $\mathcal{X}$ for some $n$, or the word is such an encoding, but $C_1$ is not an initial, or $C_n$ is not a halting configuration. These errors can be described by a proper regular expression, which can be straightforwardly derived from the definition of $\mathcal{X}$.

If a word in $\text{INVALC}(\mathcal{X})$ does not contain any structural errors, it is an encoding of some sequence of configurations $(C_i)_{i=0}^{n}$ of $\mathcal{X}$, but there is a configuration $C_i$ with $i < n$ such that $C_i \not\vdash_{\mathcal{X}} C_{i+1}$ does not hold. We call these types of errors behavioral errors, and distinguish state, head, and tape side errors, depending on which part of the configuration contains an error. We can describe state and head errors in words that do not contain structural errors using proper regular expressions; the variable is only used in the description of tape side errors. Here, Observation 17 allows us to specify all errors where the e-value of a left or right tape side is too small or too large. Furthermore, all these subexpressions are in $\text{RegEx}(1)$, and start with $\#0(0^*)\%x$. This allows us to combine them into a single expression from $\text{RegEx}(1)$. Then the expressions for all types of error can be combined to a single expression $\alpha_{\mathcal{X}} \in \text{RegEx}(1)$ with $L(\alpha_{\mathcal{X}}) = \text{INVALC}(\mathcal{X})$, which concludes the proof of Theorem 19.

Theorem 9 follows almost immediately from Theorem 19 and Lemmas 15 and 18. A detailed reasoning is provided in Section A.10 in the Appendix.

5 Conclusions

The present paper shows that extending regular expressions with only a single variable already leads to an immense increase in succinctness and expressive power. The good part of this news is that in certain applications, using the right extended regular expression instead of a proper regular expression can lead to far more efficient running times, even with the same matching engine. The bad part of this news is that this additional power can only be harnessed in full if one is able to solve undecidable problems, which greatly diminishes the usefulness of extended regular expressions as more efficient alternative to proper regular expressions.

Due to underlying undecidable problems, some questions of designing optimal extended regular expressions are of comparable difficulty to designing optimal programs. For applied computer scientists, it could be worthwhile to develop heuristics and good practices to identify cases where the non-conventional use of extended regular expressions might offer unexpected speed advantages. For theoretical computer scientists, the results in the present paper highlight the need for appropriate restrictions other than the number of variables; restrictions that lead to large and natural subclasses with decidable decision problems. One possible approach that does not extend the expressive power of proper regular expressions beyond regular languages would be a restriction of the length of the words on which variables can be bound. As the results in the present paper show, any extension of proper regular expressions that includes some kind of repetition operator needs to be approached with utmost care.

Acknowledgements

The author wishes to thank Nicole Schweikardt and the anonymous referees for their helpful remarks.
References

A

A.1 Proof of Corollary 10

Let \( c \) be a complexity measure for RegEx(1) and assume there is such a function \( m_c \). Let \( \Sigma \) be any finite alphabet with \( |\Sigma| \geq 2 \), and let

\[
U_c := \{ m_c(\alpha) \mid L(\alpha) = \Sigma^* \}.
\]

By definition of \( c \), \( U_c \) is a finite set, and therefore recursive. As \( L(\alpha) = \Sigma^* \) if and only if \( m_c(\alpha) \in U_c \), \( m_c \) and the characteristic function of \( U_c \) can be used to decide universality for RegEx(1), a problem that is not decidable (cf. Theorem 9). This is a contradiction.

A.2 Proof of Corollary 11

The result for RegEx(1) and RegEx(0) follows immediately from Theorem 9 and Theorem 4 in Hartmanis [15]: As non-RegEx(0)-ity is not semi-decidable for RegEx(1), the tradeoff between RegEx(1) and RegEx(0) is non-recursive (see also Kutrib [19] for more detailed explanations than in [15]).

Applied to RegEx(1) and RegEx(0), Hartmanis’ proof scheme gives the following proof: As non-regularity is not semi-decidable for RegEx(1), the set

\[
\Delta := \{ \alpha \in \text{RegEx}(1) \mid L(\alpha) \text{ is not regular} \}
\]

is not partially recursive. Now assume that, for a given complexity measure \( c \), the tradeoff from RegEx(1) to RegEx(0) is recursively bounded by some recursive function \( f_c \). We can then use \( f_c \) to construct a semi-decision procedure for \( \Delta \) as follows: Given some \( \alpha \in \text{RegEx}(1) \), we compute \( n := f_c(c(\alpha)) \), and let

\[
F_n := \{ \beta \in \text{RegEx}(0) \mid c(\beta) \leq n \}.
\]

As \( c \) is a complexity measure, \( F_n \) is finite, and we can effectively list all its elements (as we can effectively list all \( \beta \in \text{RegEx}(1) \) with \( c(\beta) \leq n \), and we can decide whether \( \beta \in \text{RegEx}(0) \) by searching \( \beta \) for occurrences of variables or the metacharacter \%). For each \( \beta \in F_n \), we then semi-decide \( L(\beta) \neq L(\alpha) \) by checking \( w \in L(\alpha) \) and \( w \in L(\beta) \) for all \( w \in \Sigma^* \) successively.

If \( L(\alpha) \) is not regular, then for every \( \beta \in F_n \), we find some \( w_\beta \) in finite time that proves \( L(\beta) \neq L(\alpha) \) (as \( w_\beta \) is not contained in one of the two languages, but in the other), and we can proceed to the next expression in \( F_n \). If \( L(\alpha) \) is regular, and \( f_c \) is a bound on the tradeoff from RegEx(1) to RegEx(0), there is a \( \beta \in F_n \) with \( L(\beta) = L(\alpha) \), and the procedure will never terminate. If no such \( \beta \) can be found, we know that \( \alpha \in \Delta \), and the procedure can return 1. Thus, we can construct a semi-decision procedure for \( \Delta \), which contradicts the established fact that \( \Delta \) is not partially recursive.

Likewise, we can obtain the same result if we restrict the claim to expressions from CoFRegEx(1) and CoFRegEx(0). According to Theorem 9, non-cofiniteness for RegEx(1) is not semi-decidable. Thus, given a complexity measure \( c \) for RegEx(1), a bound \( f_c \) on the tradeoff from CoFRegEx(1) to CoFRegEx(0) could be used to give a semi-decision algorithm for the set

\[
\Delta_C := \{ \alpha \in \text{RegEx}(1) \mid L(\alpha) \text{ is not cofinite} \}.
\]

First, note that the use of a complexity measure for RegEx(1) instead of CoFRegEx(1) is intentional and serves to avoid complications with the fact that cofiniteness is not decidable for RegEx(1) (recall Theorem 9). We can construct a semi-decision procedure for \( \Delta_C \) from
Using almost the same reasoning as above: Given an $\alpha \in \text{RegEx}(1)$, we define $n := f(c(\alpha))$, and let

$$F_{C,n} := \{ \beta \in \text{RegEx}(0) \mid c(\beta) \leq n, L(\beta) \text{ is cofinite} \}.$$  

Enumerating all elements of $F_{C,n}$ is slightly more difficult than enumerating all elements of $F_n$ (see above), as we also need to decide whether $L(\beta)$ is cofinite for every $\beta \in F_n$. Luckily, this is not a problem, as cofiniteness is decidable for $\text{RegEx}(0)$ (e.g., given a $\beta \in \text{RegEx}(0)$, one could convert $\beta$ into an equivalent DFA, compute its complement, and check the resulting DFA for loops that contain a final state and are reachable from the initial state). From here on, the proof continues as above, 	extit{mutatis mutandis}.

A.3 Proof of Lemma 12

Let $k \geq 1$, and let $c$ be a complexity measure for $\text{RegEx}(k)$. By definition, no $\alpha \in \text{FRegEx}(k)$ contains a Kleene star (or Kleene plus). Thus, given an $\alpha \in \text{FRegEx}(k)$, we can effectively compute the finitely many words in $L(\alpha)$ by exhausting all possible combinations of choices for each alternation symbol in $\alpha$, and computing the corresponding word for each combination, handling all bindings accordingly.

For example, the expression $\alpha := (a \mid b)(a \mid b)^* x y$ contains three alternation symbols, totaling $2^3 = 8$ possible combinations of choices, each corresponding to one of the words in $L(\alpha)$. More generally, if $\alpha \in \text{FRegEx}(k)$ contains $n$ alternation symbols, $L(\alpha)$ contains at most $2^n$ words $w_1, \ldots, w_i$ ($i \leq 2^n$), and we can compute an $\hat{\alpha} \in \text{FRegEx}(0)$ with $L(\hat{\alpha}) = L(\alpha)$ simply by computing these words $w_1, \ldots, w_i$ and defining $\hat{\alpha} := w_1 \mid \cdots \mid w_i$.

Fixing an effective procedure that computes $\hat{\alpha}$, we can straightforwardly define the recursive function $f_c : \mathbb{N} \to \mathbb{N}$ as follows: For $n \geq 0$, we define

$$F_n := \{ \hat{\alpha} \in \text{FRegEx}(0) \mid \alpha \in \text{FRegEx}(k), c(\alpha) = n \}.$$  

By definition of complexity measures, every $F_n$ is finite, and given any $n$, we can effectively list all patterns from $F_n$ (again, as $c$ is a complexity measure, and membership in $\text{FRegEx}(k)$ is straightforwardly decidable for $\text{RegEx}(k)$). For any $n \geq 0$, we define

$$f_c(n) := \max \{ c(\hat{\alpha}) \mid \hat{\alpha} \in F_n \}.$$  

As every $F_n$ can be listed effectively, every $F_n$ is finite, and as $c(\hat{\alpha})$ can be computed effectively, $f_c$ is a recursive function. Furthermore, $f_c$ is a bound on the tradeoff from $\text{FRegEx}(1)$ to $\text{FRegEx}(0)$ by definition.

A.4 Proof of Corollary 13

Assume to the contrary that, for some complexity measure $c$ for $\text{RegEx}(1)$, there is a recursive function $f_1 : \mathbb{N} \to \mathbb{N}$ such that for every $\alpha \in \text{RegEx}(1)$, if $\Sigma^* \setminus L(\alpha)$ is a $\text{RegEx}(1)$-language, there is a $\beta \in \text{RegEx}(1)$ with $L(\alpha) = L(\beta)$ and $c(\beta) \leq f_1(c(\alpha))$ (in other words, $f_1$ is a recursive bound on the blowup of complementation for $\text{RegEx}(1)$).

We can now use $f_1$ and Lemma 12 to obtain a recursive bound on the tradeoff between $\text{CoFRegEx}(1)$ and $\text{CoFRegEx}(0)$, which contradicts Corollary 11.

First, note that according to Lemma 12, there is a recursive bound $f_2 : \mathbb{N} \to \mathbb{N}$ on the tradeoff between $\text{FRegEx}(1)$ and $\text{FRegEx}(0)$.

Furthermore, we can easily prove that there is a recursive bound $f_3 : \mathbb{N} \to \mathbb{N}$ on the blowup that occurs in the complementation for $\text{FRegEx}(1)$, as $f_3$ might be computed as
follows: For every input $n$, there are finitely many $\alpha \in \text{RegEx}(0)$ with $c(\alpha) = n$. Finiteness for $\text{RegEx}(0)$ is obviously decidable; thus, we can effectively construct the finite set

$$F_n := \{\alpha \in \text{RegEx}(0) \mid c(\alpha) = n\}.$$  

For every $\alpha \in F_n$, we (effectively) construct an $\overline{\alpha} \in \text{RegEx}(0)$ with $L(\overline{\alpha}) = \Sigma^* \setminus L(\alpha)$, for example, by converting $\alpha$ into a DFA, complementing it, and converting the resulting DFA into a proper regular expression, all using the standard techniques as described in Hopcroft and Ullman [17]. We then check all $\beta \in \text{RegEx}(0)$, ordered by growing size of $c(\beta)$, until we find the smallest $\beta$ (w.r.t. $c$) with $L(\beta) = \Sigma^* \setminus L(\alpha)$, and refer to this $\beta$ as $\overline{\alpha}$ (again, this is possible due to the decidability of equivalence for $\text{RegEx}(0)$, cf. [17]), and define

$$C_n := \{\overline{\alpha}_n \mid \alpha_n \in F_n\}.$$  

Finally, we define $f_3(n)$ to be the maximum of all $c(\overline{\beta})$ with $\beta \in C_n$. By definition, $f_3$ is recursive, and serves as an upper bound for the blowup that occurs when complementing expressions from $\text{FRegEx}(0)$.

Now, consider the function $f : \mathbb{N} \to \mathbb{N}$ that is defined by $f(n) := f_3(f_2(f_1(n)))$ for every $n \in \mathbb{N}$. We shall see that our assumption implies that $f$ is a recursive bound on the tradeoff between $\text{CoFRegEx}(1)$ and $\text{CoFRegEx}(0)$, which contradicts Corollary 11. An illustration of this argument can be found in Figure 1.

First, observe that $f$ is a recursive function, as $f_1$, $f_2$, and $f_3$ are recursive functions. Due to Corollary 11, there is an $\alpha_{fe} \in \text{CoFRegEx}(1)$ such that $L(\alpha_{fe})$ is regular, but for every $\beta \in \text{RegEx}(0)$ (and hence, $\beta \in \text{CoFRegEx}(0)$) with $L(\alpha_{fe}) = L(\beta)$, $c(\beta) > f(c(\alpha_{fe}))$.

By our assumption, there is an $\overline{\alpha}_{fe} \in \text{FRegEx}(1)$ with $L(\overline{\alpha}_{fe}) = \Sigma^* \setminus L(\alpha_{fe})$ and $c(\overline{\alpha}_{fe}) \leq f_1(c(\alpha_{fe}))$.

According to Lemma 12, there is a $\overline{\beta}_{fe} \in \text{FRegEx}(0)$ with $L(\overline{\beta}_{fe}) = L(\overline{\alpha}_{fe}) = \Sigma^* \setminus L(\alpha_{fe})$, and $c(\overline{\beta}_{fe}) \leq f_2(c(\overline{\alpha}_{fe}))$.

Finally, as explained above, there is a $\beta_{fe} \in \text{CoFRegEx}(0)$ with $L(\beta_{fe}) = \Sigma^* \setminus (\overline{\beta}_{fe}) = L(\alpha_{fe})$ and

$$c(\beta_{fe}) \leq f_3(c(\overline{\beta}_{fe}))$$
$$\leq f_3(f_2(c(\overline{\alpha}_{fe})))$$
$$\leq f_3(f_2(f_1(c(\alpha_{fe}))))$$
$$= f(c(\alpha_{fe})).$$

This contradicts our choice of $\alpha_{fe}$ and concludes the proof.

### A.5 Proof of Lemma 14

We adapt the construction from the proof of Theorem 19 to the larger alphabet $\Sigma_k := \{0, #, $, a_1, b_1, \ldots, a_k, b_k\}$, where 0, #, $, all $a_i$, and all $b_i$ are pairwise distinct letters. For every $i$ with $1 \leq i \leq k$, 

![Figure 1](image-url)
let $\Sigma := \{a_i, b_i\}$. Given an extended Turing machine $\mathcal{X}$, we construct the extended regular expression $\alpha_{\mathcal{X}} \in \text{RegEx}(1)$ as in the proof of Theorem 19. For every $i$ with $1 \leq i \leq k$, we define an expression $\alpha_i \in \text{RegEx}(1)$ by

$$\alpha_i := ((a_i | b_i)^*)%x_i x_i,$$

where every $x_i$ is distinct from the variable $x$ in $\alpha_{\mathcal{X}}$, and from all $x_j$ with $j \neq i$. Finally, we define $\alpha_{\mathcal{X}, k} \in \text{RegEx}(k + 1)$ by

$$\alpha_{\mathcal{X}, k} := \alpha_{\mathcal{X}}$\alpha_1$\cdots$\alpha_k
$$
$$= \alpha_{\mathcal{X}}$((a_1 | b_1)^*)%x_1 x_1$\cdots$((a_k | b_k)^*)%x_k x_k.$$

Thus, $\alpha_{\mathcal{X}, k} \in \text{RegEx}(k + 1)$. It suffices to show that $L(\alpha_{\mathcal{X}, k})$ is a RegEx($k$)-language if and only if $L(\alpha_{\mathcal{X}})$ is regular, as neither regularity nor non-regularity is semi-decidable for RegEx($1$), cf. Theorem 9.

The if direction is obvious: If $L(\alpha_{\mathcal{X}})$ is regular, we can (non-effectively) replace that part of $\alpha_{\mathcal{X}, k}$ with an appropriate proper regular expression, and obtain an expression from RegEx($k$) for $L(\alpha_{\mathcal{X}, k})$.

For the only if direction, first note that, for every $i$, $L(\alpha_i) = \{ww \mid w \in \{a_i, b_i\}^+\}$, a language that is well known to be not regular (as can be easily verified with the Pumping Lemma, cf. Hopcroft and Ullman [17]). Likewise, note that $L_k := L(\alpha_1$\cdots$\alpha_k)$ is not a RegEx($k - 1$) language, as can be seen by the following line of reasoning: Assume there is an $\alpha \in \text{RegEx}(k - 1)$ with $L(\alpha) = L_k$. By definition, $\alpha$ contains at most $k - 1$ different variables. Note that, whenever $\alpha$ is matched to a $w \in L_k$, every variable $x$ in $\alpha$ that is bound and also referenced when matching $w$ contains only terminals from a single set $\Sigma_i \cup \{\$\}$, as $L_k \subset \{a_1, b_1\}^+$\cdots$\{a_k, b_k\}$, and repeating any string that contains some $a_j$ or $b_j$ with $j \neq i$ would break this structure. As every $L(\alpha_i)$ needs to use at least one variable, and no variable that is matched to a letter other than $\$\$ can cross the boundaries between the different $L(\alpha_i)$, there can be no $\alpha \in \text{RegEx}(k - 1)$ with $L(\alpha) = L_k$.

If $L(\alpha_{\mathcal{X}})$ is not regular, this reasoning extends to $L(\alpha_{\mathcal{X}, k})$ and RegEx($k$), as we need at least one variable to generate $L(\alpha_{\mathcal{X}})$, and $k$ variables for $L_k$, for a total of $k + 1$ variables.

Thus, $L(\alpha_{\mathcal{X}, k})$ is a RegEx($k$)-language if and only if $\alpha_{\mathcal{X}}$ is regular. As seen in the proof of Theorem 9, neither RegEx($k$)-ity nor its complement is semi-decidable for RegEx($k$).
A.6 Figure 2

Figure 2 An illustration of tape words of an extended Turing machine (as defined in Section 4). The arrow below the tape symbolizes the position of the head, while the dashed lines show the borders between the left tape side, the head position and the right tape side. Assuming that all tape cells that are not shown contain 0, we observe the left tape word $t_L = 10111\ 0^\omega$, the right tape word $t_R = 1001\ 0^\omega$, and the head letter $a = 1$.

A.7 Proof of Lemma 15

We show these results on extended Turing machines by reducing each of these problems for “non-extended” Turing machines (or, as we call them, general Turing machines, to its counterpart for extended Turing machines. A general Turing machine is a 7-tuple $M = (Q, q_1, \hat{\Gamma}, 0, \hat{\cdot}, \hat{\$}, \delta)$, where $Q$ is a finite set of states, $q_1$ is the initial state, $0 \in \hat{\Gamma}$ is the blank tape symbol, $\hat{\cdot}, \hat{\$} \in \hat{\Gamma}$ are distinct special symbols (with $\hat{\cdot}, \hat{\$} \neq 0$) that are used to mark the beginning and end (respectively) of an input of $M$, and

$$\delta : \hat{\Gamma} \times Q \rightarrow (\hat{\Gamma} \times \{L, R\} \times Q) \cup \{\text{HALT}\}$$

is the transition function. We interpret $\delta$ as for extended Turing machines, and use the same notion of tape words and configurations as for extended Turing machines.

The domain $\text{dom}_T(M)$ of a general Turing machine $M = (Q, q_1, \hat{\Gamma}, 0, \hat{\cdot}, \hat{\$}, \delta)$, is defined to be the set of all $w \in (\hat{\Gamma} \setminus \{\hat{\cdot}, \hat{\$}\})^*$ such that $M$, if started in the configuration $(0^\omega, t_R, 0, q_1)$ with $t_R = \hat{\cdot}w\hat{\$}0^\omega$, halts after finitely many steps.

The definition of $\text{dom}_T(M)$ corresponds to the definition of the language of a Turing machine as given by Hopcroft and Ullman [17] and Minsky [21]. As for extended Turing machines, we consider the following decision problems for general Turing machines:

- **Emptiness** Given a general Turing machine $M$, is $\text{dom}_T(M)$ empty?
- **Finiteness** Given a general Turing machine $M$, is $\text{dom}_T(M)$ finite?

Emptiness of $\text{dom}_T$ is undecidable due to Rice’s Theorem; and due to the Rice-Shapiro Theorem, both finiteness of $\text{dom}_T$ and its complement are not semi-decidable (cf. Cutland [11], Hopcroft and Ullman [17]).

In order to prove the present lemma’s claims on extended Turing machines, we now define an effective procedure that, given a general Turing machine $M$, returns an extended Turing machine $X$ such that $\text{dom}_T(M)$ is empty (or finite) if and only if $\text{dom}_X(X)$ is empty (or finite).

First, assume that $M$ is defined over some tape alphabet $\hat{\Gamma} \supseteq \{\hat{\$}, \hat{\cdot}, 0\}$. Using the common technique to simulate Turing machines with larger tape alphabets on Turing machines with a

\[\text{In [17], the Rice-Shapiro Theorem is called “Rice’s Theorem for recursively enumerable index sets” (Chapter 8.4).}\]
binary tape alphabet (cf. Chapter 6.3.1 in Minsky [21]), we choose a $k \geq 1$ with $2^k \geq |\bar{T}|$ and fix any injective function $b_k : \bar{T} \rightarrow \Gamma^k$ (every letter from $\bar{T}$ is encoded by a block of $k$ letters from $\Gamma$), with $b_k(0) = 0^k$ (the blank symbol of $\mathcal{M}$ is mapped to $k$ successive blank symbols of $\mathcal{X}$). We extended this function $b_k$ canonically to an injective morphism $b_k : \Gamma^* \bar{T}^* \rightarrow \Gamma^* \bar{T}^*$. Moreover, we partition the tape of $\mathcal{X}$ into non-overlapping blocks of $k$ tape letters, each representing a single tape cell of $\mathcal{M}$ as encoded by $b_k$.

The main idea of the construction is that $\mathcal{X}$ works in two phases. First, it checks that its right tape word is $b_k(\hat{\psi}_w 0^\omega) = b_k(\hat{\psi}_w 0^\omega)$ for some $\hat{w} \in (\bar{T} \setminus \{\hat{\psi}, \hat{\$}\})^\omega$. If this is the case, $\mathcal{X}$ simulates $\mathcal{M}$, always reading blocks of $k$ letters at a time and interpreting every block $b_k(a)$ as input $a$ for $\mathcal{M}$.

More explicitly, the first phase works as follows: If started on an input $w \in \Gamma^* \bar{T}^*$, $\mathcal{X}$ scans $w$ and checks whether the first block of $k$ letters is $b_k(\hat{\psi})$ (using its finite control to store the $k - 1$ letters of the block, and evaluating the whole block after reading its $k$-th letter). If this is not the case, $\mathcal{X}$ enters an infinite loop (and thus, rejects implicitly). Otherwise, $\mathcal{X}$ continues scanning to the right, evaluating every block of $k$ letters until a block with $b_k(\$)$ is encountered. On its way to the right, $\mathcal{X}$ performs the following checks: If a block contains some $b_k(a)$ with $a \in (\bar{T} \setminus \{\hat{\psi}, \hat{\$}\})$, $\mathcal{X}$ examines the next block. If a block contains $b_k(\hat{\psi})$ or some sequence of $k$ letters that is not an image of any letter from $\bar{T}$, $\mathcal{X}$ enters an infinite loop. If a $k$-letter block containing $b_k(\$)$ is found, $\mathcal{X}$ moves the head to the last letter of this block and executes the CHECKR-command. This leads the machine to enter an infinite loop if any occurrence of the non-blank symbol $\hat{1}$ in $\bar{T}$ follows.

Thus, if there is no $\hat{w} \in (\bar{T} \setminus \{\hat{\psi}, \hat{\$}\})^\omega$ such that $w = b_k(\hat{\psi}_w 0^\omega)$, $\mathcal{X}$ will never find a block $b_k(\$)$, and will never halt. Intuitively, $\mathcal{X}$ (implicitly) rejects any input that does not satisfy its sanity criteria by refusing to halt.

But if $w = b_k(\hat{\psi}_w 0^\omega)$ for some $\hat{w} \in (\bar{T} \setminus \{\hat{\psi}, \hat{\$}\})^\omega$, no tape cell containing $1$ is found. Then $\mathcal{X}$ enters its second phase: The machine returns to the left side of $w$ (which it recognizes by the unique block containing $b_k(\hat{\psi})$), and simulates $\mathcal{M}$ on the corresponding input $\hat{\psi}_w 0^\omega$ with $b_k(\hat{\psi}_w 0^\omega) = w$, always using the finite control to read blocks $b_k(a)$ of length $k$ which represent a tape letter $a \in \bar{T}$ as input for $\mathcal{M}$, and halting if and only if $\mathcal{M}$ halts. By definition, the left tape side is initially empty; hence, due to $b_k(0) = 0^k$, and due to the sanity check using CHECKR, we do not even need to keep track which part of the tape $\mathcal{X}$ has already seen.

Thus, if $w \in \text{dom}_\mathcal{X}(\mathcal{X})$, there is exactly one $\hat{w} \in (\bar{T} \setminus \{\hat{\psi}, \hat{\$}\})^\omega$ with $\hat{w} \in \text{dom}_T(\mathcal{M})$ and $b_k(\hat{\psi}_w) = w$. Likewise, for every $\hat{w} \in \text{dom}_T(\mathcal{M})$ (which, by definition, implies that $\hat{w}$ does not contain any $\hat{\psi}$ or $\hat{\$}$), $b_k(\hat{\psi}_w 0^\omega) \in \text{dom}_\mathcal{X}(\mathcal{X})$. Thus, $\text{dom}_T(\mathcal{M}) = \emptyset$ if and only if $\text{dom}_\mathcal{X}(\mathcal{X}) = \emptyset$, and likewise, $\text{dom}_T(\mathcal{M})$ is finite if and only if $\text{dom}_\mathcal{X}(\mathcal{X})$ is finite.

As the whole construction process can be realized effectively, any algorithm that (semi-)decides any of these two problems for extended Turing machines could be converted into an algorithm that (semi-)decides the corresponding problem for general Turing machines.

Due to the fact that emptiness of the domain for general Turing machines is not decidable, and as neither finiteness nor its complement is semi-decidable, the claim follows.

Those who are interested in these problems’ exact position in the arithmetical hierarchy (cf. Odifreddi [22]) can use Propositions X.9.5 and X.9.6 from Odifreddi [23] and the canonical reasoning on the order of quantifiers for the respective levels to observe that – for general and for extended Turing machines – emptiness of the domain is $\Pi_1^0$-complete, while finiteness of the domain is $\Sigma_2^0$-complete (hence, its complement is $\Pi_2^0$-complete).
A.8 Proof of Lemma 18

The if direction follows immediately: As \( \mathcal{X} \) is deterministic, every word in VALC(\( \mathcal{X} \)) corresponds to exactly one word from \( \text{dom}_X(\mathcal{X}) \) (and the computation of \( \mathcal{X} \) on that word). Thus, if \( \text{dom}_X(\mathcal{X}) \) is finite, VALC(\( \mathcal{X} \)) is also finite, and thus, regular.

For the only if direction, let \( \mathcal{X} = (Q, q_1, \delta) \), and assume that \( \text{dom}_X(\mathcal{X}) \) is infinite, while VALC(\( \mathcal{X} \)) is regular. The main idea of the proof is to show that this assumption implies the regularity of the language

\[
L_X := \{0^{e(t_n)}#0^{e(t_R)} \mid t_R \in \text{dom}_X(\mathcal{X})\}.
\]

Due to \( L_X \) being a finite subset of \( \{0^n\#0^n \mid n \geq 0\} \), we can then obtain a contradiction using the Pumping Lemma. In order to achieve this result, we use our convention that \( M \) does not halt on the very first configuration (cf. Convention 16).

As \( \mathcal{X} \) is deterministic, every word \( w \in \text{VALC}(\mathcal{X}) \) corresponds to exactly one tape word \( t_R \in \text{dom}_X(\mathcal{X}) \) and its accepting computation. This means that \( w \) has a prefix that encodes the initial configuration \( (t_L, t_R, a, q_1) \) with \( t_L = 0^\omega \) and \( a = 0 \), and its successor configuration \( (t_L', t_R', a', q_2) \). Recall that, by Convention 16, \( \delta(0, q_1) = (0, L, q_2) \). Using the first equation in Observation 17, we conclude \( e(t_L') = 0, e(t_R') = 2 e(t_R) \), and \( a' = 0 \). This means that there are \( w_p, w' \in \{0, \#\}^* \) such that \( w = w_p w' \), and

\[
w_p = \#\#0 \quad 0^{e(t_n)} \quad 0^{e(t_R)} \quad \#0 \quad 0^{e(t_n)} \quad \#0 \quad 0^{e(t_R)} \quad \#0 \quad 0^2 \quad \#0 \quad \#0 \quad \#0 \quad q_2.
\]

We now define a GSM (generalized sequential machine, cf. Hopcroft and Ullman [17]) \( M \) to transform VALC(\( \mathcal{X} \)) into the language \( L_X \). Basically, generalized sequential machines can be understood as an extension to nondeterministic finite automata. In addition to the usual behavior of an NFA, a GSM supplements every transition with an output; i.e., whenever a GSM reads a symbol, it also emits a string (as specified by its transition relation).

Thus, every path through a GSM also yields the concatenation of the emitted strings as an output. Applying a GSM \( M \) to a word \( w \) yields the language \( M(w) \) that consists of every string emitted by \( M \) along an accepting path for \( w \). Likewise, for every language \( L \), \( M(L) := \bigcup_{w \in L} M(w) \). As \( M \) maps \( L \) to \( M(L) \), this process is called a GSM mapping. As regular languages are closed under GSM mappings, the regularity of \( L_X \) then follows from the assumed regularity of VALC(\( \mathcal{X} \)).

The GSM \( M \) is defined by the transition diagram in Figure 3. Compared to [17], this definition of \( M \) uses a slightly streamlined notation by allowing \( M \) to read multiple letters in the transition between \( q_1 \) and \( q_2 \) and between \( q_3 \) and \( q_4 \). By introducing additional states, one can easily convert \( M \) into a GSM that reads one letter after the other. E.g., instead of directly transitioning from \( q_1 \) to \( q_2 \) in a single step reading \( \#\#0\#0 \) and emitting \( \lambda \), \( M \) could be redefined by adding four additional states \( q_{1,1} \) to \( q_{1,4} \), such that \( M \) goes from \( q_1 \) to \( q_{1,1} \) reading \( \# \), then \( q_{1,2} \) reading \( \# \), then \( q_{1,3} \) reading \( 0 \), then to \( q_{1,4} \) reading \( \# \), and finally to \( q_2 \) reading \( 0 \), while emitting \( \lambda \) in every step.

Thus, for every \( w \in \text{VALC}(\mathcal{X}) \), there are words \( w_p, w' \in \{0, \#\}^* \) such that \( w = w_p w' \), where \( w_p \) satisfies Equation 1 and encodes some values \( e(t_R) \) and \( e(t_R') \). Now assume that \( M \) is applied to \( w \). First note that \( M \) is deterministic, thus, \( M(w) \) can contain at most one word (in principle, \( M(w) \) could even be empty if \( M \) does not reach the accepting state \( q_6 \), but considering the structure of all words in VALC(\( \mathcal{X} \)), it is fairly easy to see that this is impossible).

When transitioning from \( q_1 \) to \( q_2 \), \( M \) removes the prefix \( \#\#0\#0 \) from \( w \). This erases the encoding of \( e(t_L) \), the surrounding \( \# \), and the 0 that precedes the encoding of \( e(t_R) \). While \( M \)
loops in $q_2$, every 0 from the encoding of $e(t_R)$ is passed through unchanged, as is the $\#$ that marks the end of that encoding when transitioning to $q_2$. Therefore, when $M$ has reached $q_3$, it has emitted $0^{e(t_R)}\#$ and read the part $\#\#0^00^{e(t_R)}\#$ of the input. In the transition from $q_3$ to $q_4$, $M$ reads $0\#0\#0\#0\#0\#$ and emits only $\lambda$—thus, the GSM passes over those parts of $w_p$ that encode $a$, $q_1$ (of $X$) and $e(t'_L)$, and over the 0 that precedes the encoding of $e(t_R)$, and emits only $\lambda$. In the loop between $q_4$ and $q_5$, $M$ emits half of the 0s that encode $e(t_R)$. When reaching the $\#$ that marks the end of that block, $M$ transitions to $q_6$ and erases all remaining letters of $w_p$ and all of $w'$. Thus, $M(w) = 0^{e(t_R)}\#0^{e(t_R)} = 0^{e(t_R)}\#0^{e(t_R)}$. Hence, if we consider not a single $w \in \text{VALC}(X)$, but the whole language, we have

$$M(\text{VALC}(X)) = \bigcup_{w \in \text{VALC}(X)} M(w) = \{0^{e(t_R)}\#0^{e(t_R)} \mid t_R \in \text{dom}_X(X)\} = L_X.$$ 

By our initial assumption, $\text{VALC}(X)$ is regular, and as the class of regular languages is closed under GSM mappings, $L_X$ must be regular as well. Also by our initial assumption, $\text{dom}_X(X)$ is infinite, which means that $L_X$ is an infinite subset of $\{0^n\#0^n \mid n \geq 0\}$. Using the Pumping Lemma (cf. Hopcroft and Ullman [17]), we can obtain the intended contradiction, as pumping any sufficiently large word from $L_X$ would lead to a word that is not a subset of $0^n\#0^n$, or to a word $0^m\#0^n$ with $m \neq n$.

### A.9 Proof of Theorem 19

Let $X = (Q, q_1, \delta)$ be an extended Turing machine. Let $\nu \geq 2$ denote the number of states of $X$; by Convention 16, $Q = \{q_1, \ldots, q_\nu\}$ for some $\nu \geq 2$. Intuitively, each element $w$ of $\text{INVALC}(X)$ contains at least one error that prevents $w$ from being an encoding of a valid computation of $X$. We distinguish two kinds of errors:

1. **Structural errors**, where a word is not an encoding of any sequence $(C_i)_{i=1}^n$ over configurations of $X$ for some $n$, or the word is such an encoding, but $C_1$ is not an initial, or $C_n$ is not a halting configuration, and
2. behavioral errors, where a word is an encoding of some sequence of configurations \((C_i)_{i=0}^{n}\) of \(\mathcal{X}\), but there is an \(i < n\) such that \(C_i \vdash_{\mathcal{X}} C_{i+1}\) does not hold. The extended regular expression \(\alpha_{\mathcal{X}}\) is defined by
\[
\alpha_{\mathcal{X}} := \alpha_{\text{struc}} \mid \alpha_{\text{beha}},
\]
where the subexpressions \(\alpha_{\text{struc}}\) and \(\alpha_{\text{beha}}\) describe all structural and all behavioral errors, respectively. Both expressions shall be defined later. Note that the variable reference mechanism shall be used only for some extended regular expressions in \(\alpha_{\text{beha}}\): most of the encoding of \(\text{INVALC}(\mathcal{X})\) can be achieved with proper regular expressions. In order to define \(\alpha_{\text{struc}}\), we take a short detour and consider the language
\[
S_\mathcal{X} := \{##0^+##0^+##0^+\{\lambda, 0\}##\{0^i \mid 1 \leq i \leq \nu\}\}^+ ## \cap \{##0^+##0^+##0^+\{0, \#\}^*
\cap \{0, \#\}^* \{00^*\#0^*## | a \in \Gamma, \delta(a, q_i) = \text{HALT}\}.
\]
Note that \(S_\mathcal{X}\) is exactly the set of all \(\text{enc}(C)\), where \(C = (C_i)_{i=1}^{\nu}\) (with \(n \geq 2\)) is a sequence of configurations of \(\mathcal{X}\); with \(C_1\) being an initial configuration (where neither the left tape side nor the head cell contain any 1), and \(C_n\) being a halting configuration (\(n \geq 2\) follows from our Convention 16 that \(\mathcal{X}\) cannot halt in the first step). In other words, all that distinguishes \(S_\mathcal{X}\) from \(\text{VALC}(\mathcal{X})\) is that for \(S_\mathcal{X}\), we do not require that \(C_i \vdash_{\mathcal{X}} C_{i+1}\) holds for all \(i < \nu\). Thus, \(\text{VALC}(\mathcal{X}) \subseteq S\).

Furthermore, \(S_\mathcal{X}\) is a regular language, as it is obtained by an intersection of three regular languages. Thus, \(\{0, \#\}^+ \setminus S\) is also a regular language, and we define \(\alpha_{\text{struc}}\) to be any proper regular expression with \(L(\alpha_{\text{struc}}) = \{0, \#\}^+ \setminus S\). It is easy to see that such an \(\alpha_{\text{struc}}\) can be constructed effectively solely from \(\mathcal{X}\), for example by constructing a deterministic finite automaton \(A\) for \(S_\mathcal{X}\), complementing \(A\) (by turning accepting into non-accepting states, and vice versa), and converting the resulting nondeterministic automaton into a proper regular expression. The DFA \(A\) depends only on \(\nu\) and the halting instructions occurring in \(\delta\) and can be constructed effectively, as can all the conversions that lead to \(\alpha_{\text{struc}}\) (again, cf. Hopcroft and Ullman [17]). The exact shape of \(\alpha_{\text{struc}}\) is of no significance to this proof, as we require only that the expression is a proper regular expression, and can be obtained effectively.

As mentioned above, \(\text{VALC}(\mathcal{X}) \subseteq S\), and thus, \(\text{INVALC}(\mathcal{X}) \supseteq L(\alpha_{\text{struc}})\). Furthermore, all elements of \(\text{INVALC}(\mathcal{X}) \setminus L(\alpha_{\text{struc}})\) are elements of \(S_\mathcal{X}\) and encode a sequence \((C_i)_{i=1}^{\nu}\) of configurations of \(\mathcal{X}\) such that \(C_i \vdash_{\mathcal{X}} C_{i+1}\) does not hold for at least one \(i\), \(1 \leq i < \nu\).

Thus, \(\text{INVALC}(\mathcal{X}) \setminus L(\alpha_{\text{struc}})\) contains exactly those words from \(S_\mathcal{X}\) that encode a sequence of configurations with at least one behavioral error. Therefore, when defining \(\alpha_{\text{beha}}\) to describe all these remaining errors, we can safely assume that the word in question is an element of \(S_\mathcal{X}\), as otherwise, it is already contained in \(L(\alpha_{\text{struc}})\). This allows us to reason about the yet to be defined elements of \(\text{INVALC}(\mathcal{X})\) purely in terms of the execution of \(\mathcal{X}\), as the encoding is already provided by the structure of \(\mathcal{X}\), and to understand all errors that are yet to be defined as incorrect transitions between configurations.

We distinguish three kinds of behavioral errors in the transition between a configuration \(C = (t_L, t_R, a, q_i)\) and a configuration \(C' = (t'_L, t'_R, a', q_j)\), where \(C \vdash_{\mathcal{X}} C'\) does not hold:

1. state errors, where \(q_j\) has a wrong value,
2. head errors, where \(a'\) is wrong, and
3. tape side errors, where \(t'_L\) or \(t'_R\) contains an error (characterized by \(e(t'_L)\) or \(e(t'_R)\) being different from the value that is expected according to Observation 17).
Each of these types of errors shall be handled by an expression \( \alpha_{\text{state}} \), \( \alpha_{\text{head}} \) or \( \alpha_{\text{tape}} \) (respectively, of course), and we define
\[
\alpha_{\text{behav}} := (\alpha_{\text{state}} \mid \alpha_{\text{head}} \mid \alpha_{\text{tape}}).
\]

Basically, each of these expressions lists all combinations of \( a \in \Gamma \) and \( q_i \in Q \), and describes the corresponding errors of the respective kind. The error that \( \mathcal{X} \) continues its computation after encountering a HALT-instruction is considered a state error and handled in \( \alpha_{\text{state}} \) (thus, we do not need to consider HALT-instructions in \( \alpha_{\text{head}} \) and \( \alpha_{\text{tape}} \)). We can already note that \( \alpha_{\text{head}} \) and \( \alpha_{\text{state}} \) are proper regular expressions, as variables and the \% metacharacter occur only in \( \alpha_{\text{tape}} \) (recall that, as \( \alpha_{\mathcal{X}} \in \mathsf{RegEx}(1) \), we are only allowed to use \% once in the whole expression).

**A.9.0.1 State errors:**

We begin with the definition of \( \alpha_{\text{state}} \). For every \( a \in \Gamma \) and every \( i \) with \( q_i \in Q \), we define a proper regular expression \( \alpha_{a,i}^{\text{state}} \), and let
\[
\alpha_{\text{state}} := \left( \alpha_{0,0}^{\text{state}} \mid \alpha_{1,1}^{\text{state}} \mid \alpha_{0,2}^{\text{state}} \mid \alpha_{1,2}^{\text{state}} \mid \cdots \mid \alpha_{0,\nu}^{\text{state}} \mid \alpha_{1,\nu}^{\text{state}} \right),
\]
where each \( \alpha_{a,i}^{\text{state}} \) lists all ‘forbidden’ follower states for \( q_i \) on \( a \). More formally, if \( \delta(a,q_i) = \text{HALT} \), let
\[
\alpha_{a,i}^{\text{state}} := (0 \mid \#)^* \#00^*\#00^i\#\#0(0 \mid \#)^*.
\]

For all words in \( S_{\mathcal{X}} \), this expression describes all cases where \( \mathcal{X} \) reads \( a \) in state \( q_i \), and continues instead of halting. First, as mentioned above, we only need to consider words from \( S_{\mathcal{X}} \), as all other words are already matched by \( \alpha_{\text{struc}} \). Due to the definition of enc, every \#\# in words from \( S_{\mathcal{X}} \) marks the boundary between two encoded configurations, and every string \#0\# immediately to the left of such a \#\# encodes a state \( q_i \). Likewise, when continuing to the left, \#00\# encodes the head letter \( a \). Thus, whenever a word from \( S_{\mathcal{X}} \) contains a string \#00\#\#0\#\#, there is a configuration where \( \mathcal{X} \) is in state \( q_i \) and reads \( a \). As \( \delta(a,q_i) = \text{HALT} \), there may not be a succeeding configuration, and this definition of \( \alpha_{a,i}^{\text{state}} \) describes all cases where \( \mathcal{X} \) continues after reading \( a \) in \( q_i \). Note that we do not need to deal with cases where \#\# is followed by yet another \#, as such words are not contained in \( S_{\mathcal{X}} \) and, thus, contained in \( L(\alpha_{\text{struc}}) \).

For those cases where \( \delta(a,q_i) = (b,M,q_j) \) for some \( M \in \{L,R\} \), some \( b \in \Gamma \), and some \( q_j \in Q \), we define
\[
\alpha_{a,i}^{\text{state}} := (0 \mid \#)^* \#00^*\#00^i\#00^+\#0^+\#0^+\#0^+\#\alpha_j^{\text{not}}\#\#(0 \mid \#)^*,
\]
where \( \alpha_j^{\text{not}} \) is any proper regular expression with
\[
L(\alpha_j^{\text{not}}) = \{0^k \mid 1 \leq k \leq \nu \text{ and } k \neq j\}.
\]

Again, we use \#00^*\#0\#\# to identify an encoding of a configuration with head letter \( a \) in state \( q_i \). To the right of \#\#, the subexpression \#0^+\#0^+\#\# is used to skip over the encodings of \( t_L', t_R' \) and \( a' \), as we only deal with state errors (for now). By definition, the invalid successor states are exactly all states from \( Q \setminus \{q_j\} \), and these are described by \( \alpha_j^{\text{not}} \). Thus, if a word from \( S_{\mathcal{X}} \) contains any state error when reading \( a \) in \( q_i \), the whole word belongs to \( \alpha_{a,i}^{\text{state}} \), and \( \alpha_{a,i}^{\text{state}} \) only matches such words.
Finally, if \( \delta(a, q_i) = (\text{CHECK}\_R, q_j) \) for some \( q_j \in Q \), we define

\[
\alpha_{a, i}^{\text{state}} := (0 \mid \#)^* \#00^0 \#0^i \#0^\#0^+ \#0^+ \#c_{i,j} \#0^\#0^+ \#(0 \mid \#)^* \\
| (0 \mid \#)^* \#00^0 \#0^i \#0^\#0^+ \#0^+ \#c_{i,j} \#0^\#0^+ \#(0 \mid \#)^* ,
\]

where \( \alpha_{j}^{\text{not}} \) is defined as in the preceding paragraph. This expression is slightly more complicated, as it needs to distinguish two cases. Recall that the \text{CHECK}\_R-instruction is to be interpreted as follows: If \( t_R = 0^\omega \), \( X \) is to change into state \( q_j \); and if \( t_R \neq 0^\omega \), \( X \) is to stay in \( q_i \), which will lead to an infinite loop. The first line of the definition handles all cases where \( t_R = 0^\omega \), while the second handles those where \( t_R \neq 0^\omega \). Again, both cases use \#00^0 \#0^i \#0^\#0^+ \#c_{i,j} \#0^\#0^+ \#(0 \mid \#)^* to identify configurations where \( X \) is in state \( q_i \), reading \( a \).

In the first case, the string \#00^0 \#0^i \#0^\#0^+ \#c_{i,j} \#0^\#0^+ \#(0 \mid \#)^* contains the additional information that \( e(t_R) = 0 \), and thus, \( t_R = 0^\omega \). The correct successor state would be \( q_j \), and the expression skips over the encodings of \( t'_L \), \( t'_R \) and \( a' \) (using \#00^0 \#0^i \#0^\#0^+ \#c_{i,j} \#0^\#0^+ \#(0 \mid \#)^* and then matches all states but \( q_j \). Likewise, in the second case, \#00^0 \#0^i \#0^\#0^+ \#c_{i,j} \#0^\#0^+ \#(0 \mid \#)^* matches all cases where (when reading \( a \) in \( q_i \)) \( e(t_R) > 0 \), which is equivalent to \( t_R \neq 0^\omega \). Again, the expression skips over the encodings of \( t'_L \), \( t'_R \) and \( a' \) and uses \( \alpha_{i}^{\text{not}} \) to identify all states that are not the correct successor state \( q_i \).

### A.9.0.2 Head errors:

As \( \alpha_{\text{state}} \) handles all cases where a halting configuration is followed by any other configuration, we can restrict our definition of the various head errors to cases where a non-halting instruction should be executed. We define

\[
\alpha_{\text{head}} := \big\{ \alpha_{0, 1}^{\text{head}} \mid \alpha_{1, 1}^{\text{head}} \mid \alpha_{0, 2}^{\text{head}} \mid \alpha_{1, 2}^{\text{head}} \mid \cdots \mid \alpha_{0, \nu}^{\text{head}} \mid \alpha_{1, \nu}^{\text{head}} \big\} ,
\]

omitting those \( \alpha_{a, j}^{\text{head}} \) with \( \delta(a, q_i) = \text{HALT} \). For all \( a \in \Gamma \), \( q_i \in Q \) with \( \delta(a, q_i) \neq \text{HALT} \), we define \( \alpha_{a, i}^{\text{head}} \) as follows.

If \( \delta(a, q_i) = (b, L, q_j) \) (for some \( q_j \in Q \)), let

\[
\alpha_{a, i}^{\text{head}} := (0 \mid \#)^* \#0^0 \#0^i \#0^\#0^+ \#0^+ \#0^0 \#(0 \mid \#)^* \\
| (0 \mid \#)^* \#0^0 \#0^i \#0^\#0^+ \#0^+ \#0^0 \#(0 \mid \#)^* .
\]

According to the first equation in Observation 17, after a left movement of the head, \( a' = e(t_L) \) mod 2 must hold. The two lines in the \( \alpha_{a, i}^{\text{head}} \) distinguish the two possible cases for \( e(t_L) \) mod 2. In both cases, we once again identify \( a \) and \( q_i \) in the encoding using \#00^0 \#0^i \#0^\#0^+ \#0^0 \#(0 \mid \#)^*.

In the first line, the expression ignores \( e(t_L) \) (using the \#0^+ to the left of \#00^0 \#0^i \#0^\#0^+ \#0^0 \#(0 \mid \#)^*), and describes all cases where \( e(t_L) \) is even (by the \( \#0^+ \) part of \#00^0 \#0^i \#0^\#0^+ \#0^0 \#(0 \mid \#)^*). To the right of \#0^\#0^+ \#0^0 \#(0 \mid \#)^*, the expression skips \( t'_L \) and \( t'_R \) and finds \( a' = 1 \), thus exactly those cases where \( e(t_L) \) is even, but \( a' = 1 \). Likewise, the second line handles the cases where \( e(t_L) \) is odd, but \( a' = 0 \). As \( a' \in \{0, 1\} \) is ensured by \( S_X \), these expressions describe exactly the head errors after \( L \)-movements.

Likewise, if \( \delta(a, q_i) = (b, R, q_j) \) (for some \( q_j \in Q \)), let

\[
\alpha_{a, i}^{\text{head}} := (0 \mid \#)^* \#0^0 \#0^i \#0^\#0^+ \#0^+ \#00 \#(0 \mid \#)^* \\
| (0 \mid \#)^* \#0^0 \#0^i \#0^\#0^+ \#0^+ \#00 \#(0 \mid \#)^* .
\]

This expression uses the second equation in Observation 17, \( a' = e(t_R) \) mod 2, and works like the expression for \( L \)-moves, the only difference being that it does not skip over the encoding of \( t_R \).

Finally, if \( \delta(a, q_i) = \text{CHECK}\_R(q_j) \) for some \( q_j \in Q \), we define

\[
\alpha_{a, i}^{\text{head}} := (0 \mid \#)^* \#0^0 \#0^i \#0^\#0^+ \#0^+ \#a \#(0 \mid \#)^* .
\]
As CHECKR-instructions do not change the tape or the head symbol, we just need to describe the case where \( a' \neq a \). The expression identifies an encoding of a configuration with head symbol \( a \) in state \( q_1 \) (again using \( \# \) as a navigation tool), skips over \( t'_L \) and \( t'_R \), and finds a head symbol \( a' = 1 \) if \( a = 0 \), or \( a' = 0 \) if \( a = 1 \). As \( a \) is fixed within every \( \alpha^{\text{head}}_{a,i} \), we can use the shorthand notation \( 0^{1-a} \) without any formal problems (as it is just another notation for 0 or \( \lambda \), depending on \( a \)).

### A.9.0.3 Tape side errors:

As mentioned above, \( \alpha_{\text{tape}} \) is the only expression in this proof that uses variables and variable bindings. In fact, as we operate in RegEx(1), we are only allowed to use a single variable (which shall be called \( x \)), and bind it only once in all of \( \alpha_{\text{tape}} \).

In order to increase the readability, we shall define \( \alpha_{\text{tape}} \) using numerous subexpressions. As most of these expressions contain the binding operator \( \% \), simply connecting them with \(|\) (as we did with the proper regular expressions in the previous cases) would force us out of RegEx(1). The main idea of this part of the proof is that, in all these expressions, the binding occurs only in a prefix that they all have in common. This allows us ‘factor out’ the variable binding, and to capture all tape side errors without leaving RegEx(1). Therefore, we do not need not be worried by the fact that most of the following definitions contain \( \% x \); as we shall see, the resulting expression \( \alpha_{\text{struct}} \) contains only a single \( \% x \).

In this section, we do not follow our usual order, as we discuss tape side errors for \( L \) and \( R \)-instructions after the tape side errors for CHECKR-instructions. If \( \delta(a,q_i) \) is a CHECKR-instruction, we define

\[
\begin{align*}
\alpha_{L,>}^{a,i} &:= (0 | \#)^+ \# 0(0^*) \% x 0^+ \# 00^+ \# 00^+ \# 0 \# x 0^+ \# (0 | \#)^*, \\
\alpha_{L,<}^{a,i} &:= (0 | \#)^+ \# 0(0^*) \% x 0^+ \# 00^+ \# 00^+ \# 0 \# x 0^+ \# (0 | \#)^*, \\
\alpha_{R,>}^{a,i} &:= (0 | \#)^+ \# 0(0^*) \% x 0^+ \# 00^+ \# 00^+ \# 0 \# x 0^+ \# (0 | \#)^*, \\
\alpha_{R,<}^{a,i} &:= (0 | \#)^+ \# 0(0^*) \% x 0^+ \# 00^+ \# 00^+ \# 0 \# x 0^+ \# (0 | \#)^*.
\end{align*}
\]

Intuitively, for \( M \in \{L, R\} \), \( \alpha_{M,>}^{a,i} \) is used to describe all successor configurations (after reading \( a \) in \( q_i \)), where \( e(t'_M) > e(t_M) \). Likewise, \( \alpha_{M,<}^{a,i} \) handles all those cases where \( e(t'_M) < e(t_M) \). We discuss the correctness of these expressions using \( \alpha_{L,>}^{a,i} \) as an example, the three other expressions behave analogously. If \( \alpha_{L,>}^{a,i} \) matches a word from \( S_X \), \( x \) is bound to some word \( 0^m \) with \( m \geq 0 \) (due to \( (0^*)\% x \)). As this \( 0^m \) belongs to the fourth block of 0s when counting from \#\# to the left, \( 0(0)^n \) corresponds to \( 00^e(t_L) \) for a configuration \((t_L, t_R, a, q_i)\). Analogously, the subexpression \( 00^e(t_L)0^+ \# \) matches the block that encodes \( e(t'_L) \). As \( 00^e(t_L)0^+ \# \) is expanded to some \( #00^00^m \) (with \( m \geq 1 \)), we know that \( e(t'_L) = 00^e(t_L) > e(t_L) \). Likewise, for every \( e(t'_L) > e(t_L) \), we can find an appropriate \( m \geq 1 \) and expand \( 0^+ \) to \( 0^m \). Thus, \( \alpha_{R,>}^{a,i} \) matches exactly those cases in which \( X \) reads \( a \) in \( q_i \), and the resulting \( e(t'_L) \) is larger than it should be (i.e., larger than \( e(t_L) \), as CHECKR-instructions do not change the tape).

The three other expressions behave analogously; but note that for \( \alpha_{R,>}^{a,i} \) and \( \alpha_{R,<}^{a,i} \), the subexpression \( 0(0^*)\% x \) matches a the coding of \( e(t_R) \) instead of \( e(t_L) \), as can be seen by the location of \#\#.

Handling tape side errors for configurations that lead to a left or right movement of the head follows the same basic principle, but is a little more complicated, as we have to deal with changes to the tape. First, recall that the equations given in Observation 17 allow us to compute \( e(t'_L) \) and \( e(t'_R) \) from \( e(t_L) \), \( e(t_R) \) and \( \delta(a,q_i) \).
If $\delta(a, q_i) \in (b, L, q_j)$ for some $q_j \in Q$, we define

$$
\alpha_{L,>}^{a,i} := (0 | \#)^*0(0^*)\%xx(0 | \lambda)\#0^+\#00^a\#0^0\#0x0^+\#(0 | \#)^* ,
\alpha_{L,<}^{a,i} := (0 | \#)^*0(0^*)\%xx000^*\%0^+\#00^a\#0^0\#0x0^+\#(0 | \#)^* ,
\alpha_{R,=}^{a,i} := (0 | \#)^*0(0^*)\%xx000^*\#0^+\#0x0^+\#0(0 | \#)^* ,
\alpha_{R,<}^{a,i} := (0 | \#)^*0(0^*)\%xx000^*\#0^+\#0x0^+\#0(0 | \#)^* ,
\alpha_{mod}^{a,i} := (0 | \#)^*0(0^a\#0^0\#0^0\#0^0\#00^a\#000^a)^{1-b}\#(0 | \#)^* .
$$

These expressions fulfill the same purpose as their equally named counterparts we defined to handle tape side errors for configurations in which a CHECKR-instruction is executed, i.e., they describe the cases where $e(t_L')$ or $e(t_R')$ contains too much or too little. For technical reasons that shall be explained a little later, we also use a proper regular expression $\alpha_{mod}^{a,i}$ to describe the errors where $e(t_R)$ has the wrong parity.

We begin with the expressions that handle errors on the left tape side. According to the first equation in Observation 17, the correct $t_L'$ is characterized by $e(t_L') = e(t_L) \bmod 2$.

First, consider $\alpha_{L,>}^{a,i}$. As before, $a$ and $q_i$ are matched in $0(0^a\#0^0\#0^0\#0^0\#00^a)$. To the left of that block, the expression skips the encoding of $e(t_R)$, and matches $0(\epsilon(t_L))$ with the expression $0(0^a\%xx(0 | \lambda)$, Note that $x$ is bound to $0(\epsilon(t_L)) \bmod 2$, and $0(\epsilon(t_L) \bmod 2)$ matches $0$ or $\lambda$ in $(0 | \lambda)$. To the right of $\#\#\#$, $0x0^+$ matches the encoding of $0(\epsilon(t_L))$. It is easily seen that this describes exactly those cases where $e(t_L') > (e(t_L) \bmod 2)$, as the $0^+$ is used to match all possible values of $e(t_L') - (e(t_L) \bmod 2)$.

Next, consider $\alpha_{L,<}^{a,i}$. Note that if $e(t_L') = (e(t_L) \bmod 2)$, either $e(t_L') = 2e(t_L')$ or $e(t_L') = 2e(t_L') + 1$. Thus, $e(t_L')$ is too small and only if $e(t_L') > e(t_L') + 1$, which holds if and only if $e(t_L) = e(t_L') + 2 + m$ for some $m \geq 0$. We can easily see that (for words from $S_X$), $\alpha_{L,<}^{a,i}$ matches exactly those encodings of successive configurations $(t_L, t_R, a, q_i)$ and $(t_L', t_R', a', q_j)$, where $e(t_L) = 2n + 2 + m$ and $e(t_L') = n$ for some $m, n \geq 0$, as $x$ binds to $0^a$, and $0(0^a\%xx000^a)$ corresponds to $00^a000^a00^a = 00^a(t_L)$. Thus, this expression handles exactly those cases where $e(t_L')$ is too small.

Hence, $\alpha_{L,>}^{a,i}$ and $\alpha_{L,<}^{a,i}$ handle all errors on the left tape side (for given $a, q_i$ with $\delta(a, q_i) = (b, L, q_j)$). As we still need to handle errors on the right tape side, recall that (according to the first equation in Observation 17), the correct right tape word $t_R'$ is characterized by $e(t_R') = 2e(t_L') + b$.

It is easy to see that $\alpha_{L,>}^{a,i}$ handles exactly those words (from $S_X$, with $X$ reading $a$ in $q_i$, where $e(t_L')$ is too large. First, $x$ is bound to $0\epsilon(t_n)$. For $t_R'$, we have $0x0^+0^0$, and thus,

$$
\epsilon(t_R') = 2e(t_R') + b + n, \text{ with } n \geq 1, \text{ where } 0^+ \text{ expresses the difference between } e(t_R') \text{ and its intended value.}
$$

The final case, where $t_R'$ is too small, is more complicated. We handle this case with two expressions. First, note that $e(t_R') \bmod 2 = b$ must hold. The expression $\alpha_{mod}^{a,i}$ describes those cases where this condition is not satisfied; i.e., $X$ reads $a$ in $q_i$, but $e(t_R') \bmod 2 \neq b$.

Therefore, we can restrict our definition of $\alpha_{R,=}^{a,i}$ to those cases where $e(t_R')$ or $b$ have the same parity. In these cases, $e(t_R') = 2n + b$ for some $n \geq 0$, but $e(t_R') > n$, which holds if and only if there is an $m \geq 0$ with $e(t_R') = n + m + 1$. The words from $S_X$ that match $\alpha_{R,=}^{a,i}$ (but not $\alpha_{mod}^{a,i}$) are exactly those that satisfy this condition: The variable $x$ is bound to $0^a$, while $0^+$ corresponds to $m + 1$. Thus, $\alpha_{mod}^{a,i} \alpha_{R,=}^{a,i}$ handles the cases where $e(t_R')$ is too small (but not exactly those cases, even when restricted to $S_X$, as $\alpha_{mod}^{a,i}$ also matches encodings of computations where $e(t_R')$ is too large and of the wrong parity).

As we have seen, these five expressions describe all tape side errors occurring during left
movements of the head. Analogously, if $\delta(a, q_i) \in (b, R, q_j)$ for some $q_j \in Q$, we define

$$
\alpha^{a,i}_{R,:} := (0 \mid \#)^*\#0(0^*)\%x(0 \mid \lambda)^*00a\#00\#0x0^+\#(0 \mid \#)^*,$n
$$
\alpha^{a,i}_{R,>} := (0 \mid \#)^*\#0(0^*)\%x000^*\#00\#0^+\#(0 \mid \#)^*,$n
$$
\alpha^{a,i}_{L,:} := (0 \mid \#)^*\#0(0^*)\%x\#00\#0^+\#0x0^+\#(0 \mid \#)^*,$n
$$
\alpha^{a,i}_{L,>} := (0 \mid \#)^*\#0(0^*)\%x0^+\#00\#0^+\#00\#x0^+\#(0 \mid \#)^*,$n
$$
\alpha^{a,i}_{mod} := (0 \mid \#)^*\#00\#0^+\#(01^+)\#(0 \mid \#)^*.
$$

As already suggested by the similarities (of the equations in Observation 17, and of the definitions of the five expressions), tape side errors for $R$-movements can be handled analogously to tape side errors for $L$-movements. Thus, it can be easily verified that these expressions describe exactly the tape side errors for all transitions where $X$ reads a in state $q_i$ (again assuming that we only consider words from $S_X$).

We can now combine all these expressions to define $\alpha_{struc}$. First, note that whenever it is defined, $\alpha^{a,i}_{mod}$ is a proper regular expression. We define

$$
\alpha_{mod} := \alpha^{0,1}_{mod} \mid \alpha^{1,1}_{mod} \mid \alpha^{0,2}_{mod} \mid \alpha^{1,2}_{mod} \mid \ldots \mid \alpha^{0,\nu}_{mod} \mid \alpha^{1,\nu}_{mod},
$$

omitting those $\alpha^{a,i}_{mod}$ that are undefined (i.e., $\delta(a, q_i)$ is a HALT- or a CHECKR-instruction).

Next, note that all other expressions for tape side errors use exactly one variable binding, and start with the common prefix $(0 \mid \#)^*\#0(0^*)\%x$. For every $\alpha^{a,i}_{M,c}$ (with $M \in \{L, R\}$ and $c \in \{>, <\}$), let $\hat{\alpha}^{a,i}_{M,c}$ be the (uniquely defined) extended regular expression that satisfies

$$
\hat{\alpha}^{a,i}_{M,c} = (0 \mid \#)^*\#0(0^*)\%x \alpha^{a,i}_{M,c}.
$$

In other words, $\hat{\alpha}^{a,i}_{M,c}$ is obtained from $\alpha^{a,i}_{M,c}$ by factoring out the prefix that contains the variable binding. We then combine all this expressions into a single expression $\alpha_{var} \in \text{RegEx}(1)$ by

$$
\alpha_{var} = (0 \mid \#)^*\#0(0^*)\%x \left( \hat{\alpha}^{0,1}_{L,>} \mid \hat{\alpha}^{0,1}_{L,<} \mid \hat{\alpha}^{0,1}_{L,R} \mid \hat{\alpha}^{0,1}_{R,L} \mid \hat{\alpha}^{1,1}_{L,>} \mid \hat{\alpha}^{1,1}_{L,<} \mid \hat{\alpha}^{1,1}_{L,R} \mid \hat{\alpha}^{1,1}_{R,L} \mid \ldots \right)
$$

$$
\left( \mid \hat{\alpha}^{0,\nu}_{L,>} \mid \hat{\alpha}^{0,\nu}_{L,<} \mid \hat{\alpha}^{0,\nu}_{L,R} \mid \hat{\alpha}^{0,\nu}_{R,L} \mid \hat{\alpha}^{1,\nu}_{L,>} \mid \hat{\alpha}^{1,\nu}_{L,<} \mid \hat{\alpha}^{1,\nu}_{L,R} \mid \hat{\alpha}^{1,\nu}_{R,L} \right),
$$

omitting all subexpressions that refer to $(a, q_i)$ where $\delta(a, q_i) = \text{HALT}$. Finally, we set

$$
\alpha_{tape} := \alpha_{mod} \mid \alpha_{var}.
$$

As discussed before, it is easy to see that $L(\alpha_{tape})$ is the union of all the languages that are generated by the various regular expressions we defined to handle tape side errors, and thus, $L(\alpha_{tape})$ contains exactly those words from $S_X$ that encode a tape side error at some point of the encoded computation. Therefore, for every word $w \in \{0, \#\}^*$, $w \in L(\alpha_X)$ if and only if

$$
w \in L(\alpha_{struc}) \cup L(\alpha_{state}) \cup L(\alpha_{head}) \cup L(\alpha_{tape}),
$$

and this holds if and only if $w$ contains a structural or behavioral error. Hence, we observe that $w \in L(\alpha_X)$ if and only $w \in \text{INVALC}(X)$, and thereby, $L(\alpha_X) = \text{INVALC}(X)$.

Finally, as this proof defines $\alpha_X$ constructively, using only $\nu$ and $\delta$, it also describes an effective procedure to compute $\alpha_X$ from $X$. This concludes the proof of Theorem 19.
A.10 Proof of Theorem 9

We prove each of the claims by reduction from one of three problems for extended Turing machines that are listed in Lemma 15 (these being emptiness, finiteness, and the complement of finiteness). Each reduction uses the same construction: Given an extended Turing machine \(\mathcal{X}\), we construct an extended regular expression \(\alpha_{\mathcal{X}} \in \text{RegEx}(1)\) with \(\text{INVALC}(\mathcal{X}) = L(\alpha_{\mathcal{X}})\) (this is possible according to Theorem 19).

Then \(\text{dom}_X(\mathcal{X}) = \emptyset\) if and only if \(\text{VALC}(\mathcal{X}) = \emptyset\), which holds if and only if \(\text{INVALC}(\mathcal{X}) = \{0, \#\}^*\), which holds if and only if \(L(\alpha_{\mathcal{X}}) = \{0, \#\}^*\), which holds if and only if \(L(\alpha) \supseteq \{0, \#\}^*\). Thus, any algorithm that decides universality for \(\text{RegEx}(1)\) could be used to decide the emptiness of the domain for extended Turing machines, which is undecidable according to Lemma 15.

Furthermore, \(\text{dom}_X(\mathcal{X})\) is finite if and only if \(\text{VALC}(\mathcal{X})\) is finite, which holds if and only if \(\text{VALC}(\mathcal{X})\) is regular (according to Lemma 18), which holds if and only if \(\text{INVALC}(\mathcal{X})\) is regular (as the class of regular languages is closed under complementation), which holds if and only if \(L(\alpha_{\mathcal{X}})\) is regular. Hence, semi-decidability of regularity for \(\text{RegEx}(1)\) would lead to semi-decidability of finiteness of \(\text{dom}_X\), a problem that is not semi-decidable (according to Lemma 15)

Likewise, as \(\text{dom}_X(\mathcal{X})\) is finite if and only if \(L(\alpha_{\mathcal{X}})\) is regular, \(\text{dom}_X(\mathcal{X})\) is infinite if and only if \(L(\alpha_{\mathcal{X}})\) is not regular. Therefore, semi-decidability of non-regularity for \(\text{RegEx}(1)\) contradicts the fact that the complement of finiteness of \(\text{dom}_X\) is not semi-decidable (see Lemma 15).

As \(\text{INVALC}(\mathcal{X})\) is cofinite if and only if \(\text{INVALC}(\mathcal{X})\) is regular, the results for regularity and non-regularity also show that neither cofiniteness nor non-cofiniteness is semi-decidable for \(\text{RegEx}(1)\).

Those who are interested in the exact position of these problems in the arithmetical hierarchy (cf. Odifreddi [22]) can conclude that universality is \(\Pi_1^0\)-complete, while regularity and co-finiteness are \(\Sigma_2^0\)-complete. For each of these problems, hardness for the respective class follows from the respective completeness of each of the problems on extended Turing machines used in the proof of Theorem 9 (see the remark at the end of the proof of Lemma 15). Membership in the respective level of the hierarchy is easily proved using the appropriate representation for that class; e.g., universality of some \(L(\alpha)\) can be expressed as \(\forall w \in \Sigma^* : w \in L(\alpha)\). As the membership problem for extended regular expressions is decidable, this proves that universality is in \(\Pi_1^0\). Likewise, non-regularity can be expressed as \(\forall \beta : \exists w \in \Sigma^* : \left[ w \in L(\alpha) \land w \notin L(\beta) \right]\), which shows that non-regularity is in \(\Pi_2^0\). Actually, under a strict interpretation of the definition given by Odifreddi [22], we would need to quantify over natural numbers; but as there are computable bijections between \(\mathbb{N}\) and \(\Sigma^*\), as well as between \(\mathbb{N}\) and \(\text{RegEx}(0)\), we can omit this technical detail.

A.11 A Technical Note on Bounded Occurrences of Variables

Although the extended regular expressions \(\alpha_{\mathcal{X}}\) that follow from the construction in the proof of Theorem 19 use only one variable \(x\), there is no bound on the number of occurrences of \(x\) in \(\alpha_{\mathcal{X}}\). In fact, the number of occurrences grows with the number of fields in the transition table \(\delta\) of \(\mathcal{X}\), and limiting that number would not allow to simulate infinitely many extended Turing machines, which is required to obtain the results in the present paper.

Nonetheless, similar to the proofs for the undecidability of inclusion for pattern languages in Bremer and Freydenberger [4], these limitations can be overcome by using a single universal Turing machine: First, let \(\psi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}\) be a universal partially recursive function, i.e,
for every partially recursive function $\phi : \mathbb{N} \rightarrow \mathbb{N}$, there is a $m \geq 0$ such that $\psi(m, n) = \phi(n)$ for every $n \geq 0$. It is an elementary fact of recursion theory that such a function exists (cf. Cutland [11], Odifreddi [22]), and moreover, there is a Turing machine $\mathcal{U}$ over some tape alphabet $\hat{\Gamma} \subseteq \{0, 1, \hat{\epsilon}, \hat{s}, a, b\}$ such that

$$\text{dom}_T(\mathcal{U}) = \{a^m 0 b^n \mid \psi(m, n) \text{ is defined}\}.$$

Using the same construction as in the proof of Lemma 15, we can build an extended Turing machine $\hat{\mathcal{U}} = (Q, q_0, \delta)$ that simulates $\mathcal{U}$, using an appropriate injective function $b_k : \hat{\Gamma} \rightarrow \Gamma^k$ with $b_k = 0^k$, for some appropriate $k \geq 3$. Instead of constructing a single extended regular expression $\alpha_{\hat{\mathcal{U}}}$, we construct an expression $\alpha_{\hat{\mathcal{U}}, m}$ for every function number $m$, using a slight modification of the proof of Theorem 19. Instead of allowing arbitrary contents for the right tape word in the initial configuration (as can be seen in the second line of the definition of the language $S_X$), we define

$$S_{\hat{\mathcal{U}}, m} := (\#^0\#^+\#^0\{\lambda, 0\}^+ \{0^i \mid 1 \leq i \leq \nu\})^+ \#^+$$

$$\cap \#^+(\#^0(b_k(\xi a^m 0)) \{0^{(2+m)k}\}^+ \#^0\#^+\#^0\{0, \#\}^*$$

$$\cap \{0, \#\}^* \{00^n\#^0\#^0 \mid a \in \hat{\Gamma}, \delta(a, q_t) = \text{HALT}\}.$$

The only difference to the definition of $S_X$ is the second line. Evidently, the new definition allows exactly those initial right tape sides $t_R$ for which $e(t_R) = e(b_k(\xi a^m 0)) + i(2^{2+m})k$ for some $i \geq 0$. As $|b_k(\xi a^m 0)| = (2 + m)k$, these are exactly that $t_R$ have $b_k(\xi a^m 0)$ as a prefix.

The construction of $\alpha_{\hat{\mathcal{U}}, m}$ then uses $S_{\hat{\mathcal{U}}, m}$ to construct $\alpha_{\text{struc}}$, and proceeds as in the original proof. Thus, $\{0, \#\}^* \setminus L(\alpha_{\hat{\mathcal{U}}, m})$ is exactly that set that corresponds to the valid computations of $\mathcal{U}$ on some input that starts with $a^m 0$. Furthermore, the number of occurrences of $x$ in every $\alpha_{\hat{\mathcal{U}}, m}$ depends only on $\delta$, not on $m$, which means that we can bound that number.

For any partially recursive function $\phi : \mathbb{N} \rightarrow \mathbb{N}$, let $\text{dom}(\phi) := \{n \geq 0 \mid \phi(n) \text{ is defined}\}$, and, for every $m \geq 0$, let $\psi_m$ denote the function that is defined by $\psi_m(n):=\psi(m, n)$ for every $n \geq 0$. Then $L(\alpha_{\hat{\mathcal{U}}, m})$ is

- regular if and only if $\text{dom}(\psi_m)$ is finite,
- cofinite if and only if $\text{dom}(\psi_m)$ is finite,
- universal if and only if $\text{dom}(\psi_m)$ is empty,

following the same reasoning as for $L(\alpha_X)$. As finiteness of dom is not semi-decidable, and emptiness not decidable (cf. Odifreddi [23] or the analogous results for Turing machines in Lemma 15), we arrive at the same conclusions as in Theorem 9 and Corollaries 11 and 13 for $\text{RegEx}(1)$ with a limited number of occurrences of the single variable.