Noname manuscript No.
(will be inserted by the editor)

Extended Regular Expressions: Succinctness and Decidability

Dominik D. Freydenberger

Received: date / Accepted: date

Abstract Most modern implementations of regular expression engines allow the use of variables (also called backreferences). 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, 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.

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

1 Introduction

Since being introduced by Kleene [23] 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

A preliminary version of this article appeared as [15].

D. D. Freydenberger
Institute for Computer Science
Goethe-University
Postfach 11 19 32
60054 Frankfurt am Main, Germany
Tel.:+49-69-79828250
Fax: +49-69-79828334
E-mail: freydenberger@em.uni-frankfurt.de

various aspects of their complexity, is still a very active area of investigation (see Holzer and Kutrib [20] 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 [16]), 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 *backreferences*, also called *variables*, which specify repetitions that increase the expressive power beyond the class of regular languages. For example, the (non-regular) language $L = \{ww \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$ is generated by the extended regular expression $\alpha := \big((\mathtt{a} \mid \mathtt{b})^*\big)\%x\, x$.

This expression can be understood as follows (for a more formal treatment, see Definition 4): For any expression β, $(\beta)\%x$ matches the same expression as β, and binds the match to the variable x. In the case of this example, the subexpression $\big((\mathtt{a} \mid \mathtt{b})^*\big)\%x$ can be matched to any word $w \in \{\mathtt{a},\mathtt{b}\}^*$, 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 $w \in \{\mathtt{a},\mathtt{b}\}^*$. Analogously, the expression $\big((\mathtt{a} \mid \mathtt{b})^*\big)\%x\, xx$ generates the language of all www with $w \in \{\mathtt{a},\mathtt{b}\}^*$.

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 [7]). Nonetheless, there is a common core to these variants, which was first formalized by Aho [2]. Later, Câmpeanu et al. [9] introduced a different formalization that is closer to the real world syntax, which addresses some questions of semantics that were implicitly left open in [2]. In addition to this, the *pattern expressions* by Câmpeanu and Yu [8] and the *H-expressions* by Bordihn et al. [5] use comparable repetition mechanisms and possess similar expressive power.

Still, theoretical investigation of extended regular expressions has been comparatively rare (in particular when compared to their more prominent subclass); see e. g., Larsen [25], Della Penna et al. [14], Câmpeanu and Santean [7], Carle and Narendran [10], and Reidenbach and Schmid [30].

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 [2]); 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 [2]). 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 $L_n{:=}\{www \mid w \in \{\mathtt{a}, \mathtt{b}\}^+, |w| = n\}$. These languages L_n are finite, and hence, regular. For the sake of this example, we define the *length* of an extended regular expression α as the total number of symbols that occur in α (in literature, this measure is often called *size*, cf. [21]).

With some effort, one can prove that every proper regular expression for L_n is at least of length exponential in n, e. g., by using the technique by Glaister and Shallit [17] to prove that every NFA for L_n requires at least $O(2^n)$ states. Due to the construction used in the proof of Theorem 2.3 in Hopcroft and Ullman [22], this also gives a lower bound on the length of the regular expressions for L_n.

In contrast to this, L_n is generated by the extended regular expression

$$\alpha_n{:=}(\underbrace{(\mathtt{a} \mid \mathtt{b}) \cdots (\mathtt{a} \mid \mathtt{b})}_{n \text{ times } (\mathtt{a} \mid \mathtt{b})})\%x\, xx,$$

which is of a length that is linear in n. ◇

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

Example 2 demonstrates that the use of variables can lead to languages that are (non-trivially) regular. The phenomenon that an expression like α can generate a cofinite language is strongly related to the notion of *avoidable patterns* (cf. Cassaigne [11]), and involves some very hard combinatorial questions (in particular, Example 2 illustrates this connection for the pattern xx over a binary alphabet).

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 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 [2]), 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. Section 2.3 for additional context). 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 preliminaries. Section 3 consists of Theorem 10 (the main undecidability result), its proof, and the required additional technical preliminaries, while Section 4 discusses the consequences and some extensions of Theorem 10.

2 Preliminaries

This paper is largely self-contained. Unexplained notions can be found in Hopcroft and Ullman [22], Cutland [13], and Minsky [27].

2.1 Basic Definitions

Let $\mathbb{N}$ be the set of natural numbers, including 0. The function div denotes integer division, and mod denotes its remainder (e. g., $5 \operatorname{div} 3 = 1$ and $5 \bmod 3 = 2$). The symbol ∞ denotes infinity.

The symbols $\subseteq$, $\subset$, $\supseteq$ and $\supset$ refer to the subset, proper subset, superset and proper superset relation, respectively. The symbol $\emptyset$ denotes the empty set, $\setminus$ denotes the set difference (defined by $A \setminus B := \{x \in A \mid x \notin B\}$). For every set A, $\mathcal{P}(A)$ denotes the power set of A.

We denote the *empty string* by λ. For the *concatenation* of two strings w_1 and w_2, we write $w_1 \cdot w_2$ or simply $w_1 w_2$. 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 $w = u_1 v u_2$. 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^{ω} and, for every $a \in A$, let a^{ω} 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 $w = u\, a^{\omega}$ with $u \in A^*$ 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 $w_0 \cdot \ldots \cdot w_{|u|-1} = u$ and $w_{i+|u|} = v_i$ for every $i \geq 0$, while vu is undefined. In particular, note that $a\, a^{\omega} = a^{\omega}$ 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, the following definition of syntax is due to Aho [2]:

Definition 3 Let Σ be an infinite set of *terminals*, let X be an infinite set of *variables*, and let the set of *metacharacters* consist of λ, (,), |, *, and %, where all three sets are pairwise disjoint. We define the set of *extended regular expressions* to be the smallest set that satisfies the following conditions:

1. Every $a \in \Sigma \cup X \cup \{\lambda\}$ is an extended regular expression.
2. If α_1 and α_2 are extended regular expression, then
   (a) $(\alpha_1)(\alpha_2)$ *(concatenation)*,
   (b) $(\alpha_1) \mid (\alpha_2)$ *(alternation)*,
   (c) $(\alpha_1)^*$ *(Kleene star)*
   are extended regular expressions.
3. For every extended regular expression α and every variable $x \in X$ such that $\%x$ is not a factor of α, $(\alpha)\%x$ is an extended regular expression *(variable binding)*.

We denote the set of all extended regular expressions by RegEx. A *proper regular expression* is an extended regular expression that contains neither %, nor any variable (hence, proper regular expressions are those expressions that are commonly called “regular expressions” in theoretical computer science).

If an extended regular expression β is a factor of an extended regular expression α, we call β a *subexpression* of α. We denote the set of all subexpressions of α by $\mathrm{SUB}(\alpha)$.

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 * and $^+$ ranking over concatenation ranking over the alternation operator |.

In Aho [2], an informal definition of the semantics of extended regular expressions is given. In Aho's approach, extended regular expressions are interpreted as language generators in the following way: An extended regular expression α is interpreted from left to right. A subexpression of the form $(\beta)\%x$ generates the same language as the expression β; in addition to this, the variable x is bound to the word w that was generated from β (if x already has a value, that value is overwritten). Every occurrence of x that is not in the context of a variable binding is then replaced with w.

When following this approach, there are some cases where the semantics are underspecified. For example, Aho [2] does not explicitly address the rebinding of variables (cf. Example 5, further down), and the semantics of expressions like $((\mathtt{a})\%x \mid \mathtt{b})\, x$ are unclear.

Although the proofs in the present paper are not affected by the ambiguities that arise from the informal approach, we include a formal definition of the semantics, which is an adaption of the semantics of Câmpeanu et al. [9] to the syntax from Definition 3:

Definition 4 A *match tree* of an extended regular expression α is a finite (directed, ordered) tree T_α, where the nodes of T_α are labeled with elements of $\Sigma^* \times \mathrm{SUB}(\alpha)$, and T_α is constructed according to the following rules:

1. The root of T_α is labeled with some (w, α), $w \in \Sigma^*$.
2. If a node v of T_α is labeled by some (w, a) with $a \in \Sigma \cup \{\lambda\}$, then v is a leaf, and $w = a$ holds.
3. If a node v of T_α is labeled by some (w, β) with $\beta = (\beta_1)(\beta_2)$, then v has exactly two children v_1 and v_2 (as left and right child, respectively), with respective labels (w_1, β_1) and (w_2, β_2), where $w_1, w_2 \in \Sigma^*$, and $w = w_1 w_2$.
4. If a node v of T_α is labeled by some (w, β) with $\beta = (\beta_1) \mid (\beta_2)$, then v has exactly one child v', that is labeled (w, β_1) or (w, β_2).
5. If a node v of T_α is labeled by some (w, β) with $\beta = (\beta_1)^*$, we distinguish two cases:
   (a) If $w = \lambda$, v has exactly one child v_1 that is labeled (λ, β), and v_1 is a leaf of T_α.
   (b) If $w \neq \lambda$, v has $k \geq 1$ children that are labeled by $(w_1, \beta_1), \ldots, (w_k, \beta_1)$ (from left to right), where $w_1, \ldots, w_k \in \Sigma^+$ and $w = w_1 \cdots w_k$.
6. If a node v of T_α is labeled by some (w, β) with $\beta = (\beta_1)\%x$, then v has exactly one child that is labeled (w, β_1).
7. If a node v of T_α is labeled by some (w, x), where x is a variable, we let $\prec$ denote the post-order on the nodes of T_α (that results from a left-to right, depth-first traversal), and distinguish the following two cases:
   (a) If there is no node v_1 of T_α with $v_1 \prec v$ such that v_1 is labeled with some $(w_1, (\beta_1)\%x)$, v is a leaf, and $w = \lambda$.
   (b) Otherwise, let v_1 denote that node with $v_1 \prec v$ that is $\prec$-maximal among the nodes that have some $(\beta')\%x$ as the second component of their label. Then v is a leaf, and $w = w_1$, where w_1 is the first component of the label of v_1.

We define the language $L(\alpha)$ that is generated by an extended regular expression α as

$$L(\alpha) := \{w \in \Sigma^* \mid (w, \alpha) \text{ labels the root of some match tree } T_\alpha \text{ of } \alpha\}.$$

Example 5 Consider the following extended regular expressions:

$$\begin{aligned}
\alpha_1 &:= \big((\mathtt{a} \mid \mathtt{b})^*\big)\%x\, xx\ \big((\mathtt{a} \mid \mathtt{b})^*\big)\%x\, x,\\
\alpha_2 &:= \big(\big((\mathtt{a} \mid \mathtt{b})^*\big)\%x\, x\big)^+,\\
\alpha_3 &:= ((\mathtt{a})\%x \mid \mathtt{b})\, x,\\
\alpha_4 &:= (\mathtt{a})\%x(\lambda(\lambda)\%x)^* x.
\end{aligned}$$

These expressions generate the following languages:

$$\begin{aligned}
L(\alpha_1) &= \left\{ vvvww \mid v, w \in \{\mathtt{a}, \mathtt{b}\}^* \right\},\\
L(\alpha_2) &= \left\{ w_1 w_1 \cdots w_n w_n \mid n \geq 1, w_i \in \{\mathtt{a}, \mathtt{b}\}^* \right\},\\
L(\alpha_3) &= \{\mathtt{a\,a}, \mathtt{b}\},\\
L(\alpha_4) &= \{\mathtt{a\,a}\}.
\end{aligned}$$

Note that in the case of α_4, which was pointed out by one of the anonymous referees, the semantics given in Definition 4 might be considered counterintuitive, as one could expect $L(\alpha_4) = \{\mathtt{a}, \mathtt{a\,a}\}$. The results in the present paper do not rely on such pathological cases, which shall be excluded by Definition 6 a little further down. $\diamond$

In general, the membership problem for RegEx is NP-complete, as shown in Theorem 6.2 in Aho [2]. 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 to expressions like α_2 in our Example 5; therefore, and in order to exclude problematic expressions like α_4 Example 5, in we formalize a slightly stronger restriction than Aho, and consider the following subclasses of RegEx:

Definition 6 For $k \geq 0$, let RegEx(k) denote the class of all extended regular expressions α that satisfy the following properties:

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

Intuitively, these restrictions on extended regular expressions in 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 RegEx(0) is equivalent to the class of proper regular expressions; furthermore, observe that RegEx(k) $\subset$ RegEx($k+1$) for every $k \geq 0$.

Referring to the extended regular expressions given in Example 5, we observe that, as $\%x$ occurs twice in α_1, α_1 is not element of any RegEx(k) with $k \geq 0$, but the extended regular expression $\alpha_1' := \big((\mathtt{a} \mid \mathtt{b})^*\big)\%x\, xx\ \big((\mathtt{a} \mid \mathtt{b})^*\big)\%y\, y$ generates the same language as α_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 $()^*$ subexpression (as we defined $^+$ through *).

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

We also consider the class FRegEx of all extended regular expressions that do not use the operator * (or $^+$), and its subclasses

$$\text{FRegEx}(k) := \text{FRegEx} \cap \text{RegEx}(k)$$

for $k \geq 0$. Thus, FRegEx contains exactly those expressions that generate finite (and, hence, regular) languages. Analogously, for every $k \geq 0$, we define a class CoFRegEx(k) as the class of all $\alpha \in \text{RegEx}(k)$ such that $L(\alpha)$ is cofinite. Unlike the classes FRegEx(k), these classes have no straightforward syntactic definition – as we shall prove in Theorem 10, cofiniteness is not semi-decidable for RegEx(k) (if $k \geq 1$).

2.3 Decision Problems and Descriptional Complexity

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

Definition 7 Let Σ denote a fixed terminal alphabet. For all $k, l \geq 0$, we define the following decision problems for 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 10 – 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. As a side note, note that cofiniteness for RegEx 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 [12]).

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. [3]).

In order to examine the relative succinctness of RegEx(1) in comparison to RegEx(0), we build on well-established notions from descriptional complexity (see Holzer and Kutrib [21] and the less recent Goldstine et al. [18] for survey articles on the area). In particular, we use the following notion of complexity measures, which is based on the more general notion of complexity measures for descriptional systems from [24]:

Definition 8 Let $\mathcal{R}$ be a class of extended regular expressions. A *complexity measure* for $\mathcal{R}$ is a total recursive function $c : \mathcal{R} \to \mathbb{N}$ such that, for every alphabet Σ, the set of all $\alpha \in \mathcal{R}$ with $L(\alpha) \subseteq \Sigma^*$

1. 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 used in Example 1), 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 [24] 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, which is again based on the more general definition from [24]:

Definition 9 Let $k > l \geq 0$ and let c be a complexity measure for $\mathrm{RegEx}(k)$ (and thereby also for $\mathrm{RegEx}(l)$). A recursive function $f_c : \mathbb{N} \to \mathbb{N}$ is said to be a *recursive upper bound for the tradeoff between* $\mathrm{RegEx}(k)$ *and* $\mathrm{RegEx}(l)$ if, for all those $\alpha \in \mathrm{RegEx}(k)$ for which $L(\alpha)$ is a $\mathrm{RegEx}(l)$-language, there is a $\beta \in \mathrm{RegEx}(l)$ with $L(\beta) = L(\alpha)$ and $c(\beta) \leq f_c(c(\alpha))$.

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

The first non-recursive tradeoffs where demonstrated by Meyer and Fischer [26]; since then, there has been a considerable amount of results on a wide range of non-recursive tradeoffs between various description mechanisms. For a survey on non-recursive tradeoffs, see Kutrib [24].

2.4 Extended Regular Expressions in the Real World

In this section we take a brief closer look at the relation between our definition of extended regular expressions, and the variable-like mechanisms in the real-world regex dialects that inspired that definition. This section is supposed to provide a wider context of the theoretical model discussed in the present paper and can be skipped without loss of continuity.

While the syntax in Câmpeanu et al. [9] is based on the *backreferences* that can be found in Perl Compatible Regular Expressions, POSIX, and various related implementations (cf. Friedl [16]), our definition is syntactically closer to the *named capture groups* that originated from Python (ibid.). As an example, consider the extended regular expression

$$\alpha := (\mathtt{a}) \mid \left((\mathtt{a}\,\mathtt{a}^+)\%x\; x^+\right),$$

which is due to Abigail [1]. It is easy to see that this expression generates the language

$$L(\alpha) = \{\mathtt{a}^n \mid n \geq 1,\ n \text{ is not a prime number}\}.$$

In Python, the same language can be expressed using the expression

```
(a)|((?P=<x>(aa+))(?P=x)+)
```

where the (prefix) operator `?P=<x>` acts like our (postfix) variable binding operator $\%x$, while `(?P=x)` repeats that variable and corresponds to our use of x (without %). A similar syntax is used in .NET, see p. 137 in Friedl [16].

Although this method of explicit naming and referencing of capture groups is also supported in newer versions of PERL (from version 5.10 onwards), traditionally, PERL allows only implicit naming of capture groups, using the aforementioned backrefences. Using these, α would be written

```
(a)|((aa+)(\3)+)
```

Here, the backreference `\3` repeats the match of the third pair of parentheses (as defined by the third opening parenthesis when reading from the left). To the author's knowledge, there is no valid way of expressing pathological examples like α_1 from Example 5 in these programming languages — in particular, the reuse of capture group names is forbidden. Nonetheless, all expressions from RegEx(1) can be easily converted to each of this dialects. Hence, all "negative" results in the present paper apply not only to our theoretical model of RegEx, but to each of these real world regex dialects.

As an unrelated side note, the author wishes to point out that $L(\alpha)$ is an example of an extended regular expression over a unary terminal alphabet that generates a non-regular language (another example of such a language can be found in Carle and Narendran [10]).

3 The Main Theorem and Its Proof

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 8, or with respect to the number of 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 7, as shown in the main theorem of this section:

Theorem 10 *For* RegEx(1)*, universality is not semi-decidable; and regularity and cofiniteness are neither semi-decidable, nor co-semi-decidable.*

The proof of Theorem 10 requires considerable technical preparation and takes up the remainder of the present section. Readers who are more interested in implications and applications of Theorem 10 are invited to skip over to Section 4.

On a superficial level, we prove Theorem 10 by using Theorem 14 (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 7). 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. Most importantly, when applied to "standard" Turing machines, the construction procedure and the encoding we shall use in the proof do not preserve the finiteness of the domain of the encoded machine. As the distinction between Turing machines with finite and infinite domain is a central element of the proofs further down, we introduce the notion of an extended Turing machine as an intermediate step in the construction of extended regular expressions from Turing machines.

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 δ is a function $\delta : \Gamma \times Q \to (\Gamma \times \{L, R\} \times Q) \cup \{\mathrm{HALT}\} \cup (\{\mathrm{CHECK_R}\} \times Q)$. 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 $\delta(a, q) = \mathrm{HALT}$, the machine halts and accepts.

The command $\mathrm{CHECK_R}$ works as follows: If $\delta(a, q) = (\mathrm{CHECK_R}, p)$ 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, $\mathcal{X}$ remains in q. As the tape is never changed during a $\mathrm{CHECK_R}$-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 14, 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 1.

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 $\vdash_{\mathcal{X}}$ denotes

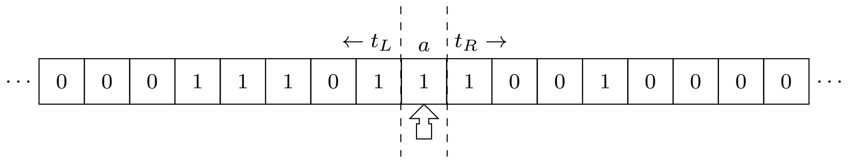

Fig. 1 An illustration of tape words of an extended Turing machine (as defined in Section 3). The arrow below the tape symbolizes the position of the head, while the dashed lines show the borders between the left tape side, the head position and the right tape side. Assuming that all tape cells that 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 $a = 1$.

the successor relation on configurations of $\mathcal{X}$, i. e., $C \vdash_{\mathcal{X}} C'$ if $\mathcal{X}$ enters C' immediately after C.

We define $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$, 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 $\mathrm{dom}_{\mathrm{X}}$ 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 [27]). 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 $\mathrm{CHECK_R}$-instruction in an extended Turing machine $\mathcal{X}$, we then would have to deal with the unfortunate side effect that a nonempty $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ could never be finite: Assume $w \in \Gamma^*$ such that $w\,0^\omega \in \mathrm{dom}_{\mathrm{X}}(\mathcal{X})$. 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 $wv1\,0^\omega \in \mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ and $wv0\,0^\omega \in \mathrm{dom}_{\mathrm{X}}(\mathcal{X})$, as $\mathcal{X}$ will not reach the part where $wv1$ and $wv0$ differ. This observation leads to $wvx\,0^\omega \in \mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ for every $x \in \Gamma^*$, and applies to various other extensions of the Turing machine model. As Lemma 13 – and thereby most of the main results in Section 4 – crucially depends on the fact that there are extended Turing machines with a finite domain, we use $\mathrm{CHECK_R}$ 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 [27]) and the corresponding undecidability results, we establish the following negative results on decision problems for extended Turing machines:

Lemma 11 *Consider the following decision problems for extended Turing machines:*

Emptiness: Given an extended Turing machine $\mathcal{X}$, is $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ empty?
Finiteness: Given an extended Turing machine $\mathcal{X}$, is $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ finite?

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

Proof 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 $\mathcal{M} = (Q, q_1, \hat{\Gamma}, 0, ¢, \$, \delta)$, where Q is a finite set of states, q_1 is the initial state, $0 \in \hat{\Gamma}$ is the blank tape symbol, $¢, \$ \in \hat{\Gamma}$ are distinct special symbols (with $¢, \$ \neq 0$) that are used to mark the beginning and end (respectively) of an input of $\mathcal{M}$, and

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

is the transition function. We interpret δ as for extended Turing machines, and us the same notion of tape words and configurations as for extended Turing machines.

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

The definition of $\text{dom}_{\text{T}}(\mathcal{M})$ corresponds to the definition of the language of a Turing machine as given by Hopcroft and Ullman [22] and Minsky [27]. As for extended Turing machines, we consider the following decision problems for general Turing machines:

Emptiness: Given a general Turing machine $\mathcal{M}$, is $\text{dom}_{\text{T}}(\mathcal{M})$ empty?
Finiteness: Given a general Turing machine $\mathcal{M}$, is $\text{dom}_{\text{T}}(\mathcal{M})$ finite?

Emptiness of dom_{T} is undecidable due to Rice's Theorem; as it is obviously co-semi-decidable, it cannot be semi-decidable. Furthermore, due to the Rice-Shapiro Theorem, finiteness of dom_{T} is neither semi-decidable, nor co-semi-decidable (cf. Cutland [13], Hopcroft and Ullman [22] – in the latter reference, the Rice-Shapiro Theorem is called "Rice's Theorem for recursively enumerable index sets" (Chapter 8.4)).

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 $\mathcal{M}$, returns an extended Turing machine $\mathcal{X}$ such that $\text{dom}_{\text{T}}(\mathcal{M})$ is empty (or finite) if and only if $\text{dom}_{\text{X}}(\mathcal{X})$ is empty (or finite).

First, assume that $\mathcal{M}$ is defined over some tape alphabet $\hat{\Gamma} \supseteq \{\$, ¢, 0\}$. Using the common technique to simulate Turing machines with larger tape alphabets on Turing machines with a binary tape alphabet (cf. Chapter 6.3.1 in Minsky [27]), we choose a $k \geq 1$ with $2^k \geq |\hat{\Gamma}|$ and fix any injective function $b_k : \hat{\Gamma} \to \Gamma^k$ (every letter from $\hat{\Gamma}$ is encoded by a block of k letters from Γ), with $b_k(0) = 0^k$ (the blank symbol of $\mathcal{M}$ is mapped to k successive blank symbols of $\mathcal{X}$). We extend this function b_k canonically to an injective morphism $b_k : \hat{\Gamma}^* 0^\omega \to \Gamma^* 0^\omega$. Moreover, we partition the tape of $\mathcal{X}$ into non-overlapping blocks of k tape cells, 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 $b_k(¢\hat{w}\$\,0^\omega) = b_k(¢\hat{w}\$)\,0^\omega$ for some $\hat{w} \in (\hat{\Gamma} \setminus \{¢, \$\})^*$. 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 $w \in \Gamma^* 0^\omega$, $\mathcal{X}$ scans w and checks whether the first block of k letters is $b_k(¢)$ (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 $a \in (\hat{\Gamma} \setminus \{¢, \$\})$, $\mathcal{X}$ examines the next block. If a block contains $b_k(¢)$ or some sequence of k letters that is not an image of any letter from $\hat{\Gamma}$, $\mathcal{X}$ 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 $\mathrm{CHECK}_\mathrm{R}$-command. This leads the machine to enter an infinite loop if any occurrence of the non-blank symbol $1 \in \Gamma$ follows.

Thus, if there is no $\hat{w} \in (\hat{\Gamma} \setminus \{¢, \$\})^*$ such that $w = b_k(¢\hat{w}\$)\,0^\omega$, $\mathcal{X}$ 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 $w = b_k(¢\hat{w}\$)\,0^\omega$ for some $\hat{w} \in (\hat{\Gamma} \setminus \{¢, \$\})^*$, no tape cell containing 1 is found by $\mathrm{CHECK}_\mathrm{R}$. Then $\mathcal{X}$ enters its second phase: The machine returns to the left side of w (which it recognizes by the unique block containing $b_k(¢)$), and simulates $\mathcal{M}$ on the corresponding input $¢\hat{w}\$$ with $b_k(¢\hat{w}\$)\,0^\omega = w$, always using the finite control to read blocks $b_k(a)$ of length k which represent a tape letter $a \in \hat{\Gamma}$ 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 $\mathrm{CHECK}_\mathrm{R}$, we do not even need to keep track which part of the tape $\mathcal{X}$ has already seen.

Thus, if $w \in \mathrm{dom}_\mathrm{X}(\mathcal{X})$, there is exactly one $\hat{w} \in (\Gamma \setminus \{¢, \$\})^*$ with $\hat{w} \in \mathrm{dom}_\mathrm{T}(\mathcal{M})$ and $b_k(¢\hat{w}\$) = w$. Likewise, for every $\hat{w} \in \mathrm{dom}_\mathrm{T}(\mathcal{M})$ (which, by definition, implies that $\hat{w}$ does not contain any ¢ or \$), $b_k(¢\hat{w}\$)\,0^\omega \in \mathrm{dom}_\mathrm{X}(\mathcal{X})$. Thus, $\mathrm{dom}_\mathrm{T}(\mathcal{M}) = \emptyset$ if and only if $\mathrm{dom}_\mathrm{X}(\mathcal{X}) = \emptyset$, and likewise, $\mathrm{dom}_\mathrm{T}(\mathcal{M})$ is finite if and only if $\mathrm{dom}_\mathrm{X}(\mathcal{X})$ 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 semi-decidable, and as finiteness is neither semi-decidable nor co-semi-decidable, the claim follows. □

Those who are interested in these problems' exact position in the arithmetical hierarchy (cf. Odifreddi [28]) can use Propositions X.9.5 and X.9.6 from Odifreddi [29] 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 Π^0_1-complete, while finiteness of the domain is Σ^0_2-complete (hence, its complement is Π^0_2-complete).

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

Convention 12 *Every extended Turing machine*

1. *has the state set* $Q = \{q_1, \dots, q_\nu\}$ *for some* $\nu \geq 1$*, where* q_1 *is the initial state,*
2. *has* $\delta(0, q_1) = (0, L, q_2)$*,*
3. *has* $\delta(a, q) = \mathrm{HALT}$ *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 Γ, define $\mathrm{e}(t){:=}\sum_{i=0}^{\infty} 2^i\, \mathrm{e}(t_i)$, where $\mathrm{e}(0){:=}0$ and $\mathrm{e}(1){:=}1$. Most of the time, we will not distinguish between single letters and their values under e, and simply write a instead of $\mathrm{e}(a)$ 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 Γ. 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 14:

Observation 1 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 $C \vdash_{\mathcal{X}} C'$, the following holds:

$$\begin{aligned}
&\text{If } M = L\text{:} \quad \mathrm{e}(t'_L) = \mathrm{e}(t_L) \operatorname{div} 2, \quad \mathrm{e}(t'_R) = 2\,\mathrm{e}(t_R) + b, \quad a' = \mathrm{e}(t_L) \bmod 2,\\
&\text{if } M = R\text{:} \quad \mathrm{e}(t'_L) = 2\,\mathrm{e}(t_L) + b, \quad \mathrm{e}(t'_R) = \mathrm{e}(t_R) \operatorname{div} 2, \quad a' = \mathrm{e}(t_R) \bmod 2.
\end{aligned}$$

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 $\mathrm{e}(t_M)$ 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 enc of configurations of $\mathcal{X}$ by

$$\operatorname{enc}(t_L, t_R, a, q_i) {:=} 00^{\mathrm{e}(t_L)} \texttt{\#} 00^{\mathrm{e}(t_R)} \texttt{\#} 00^{\mathrm{e}(a)} \texttt{\#} 0^i$$

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

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

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 $(0^\omega, w, 0, q_1)$ with $w \in \Gamma^* 0^\omega$), C_n is a halting configuration, and for every $i < n$, $C_i \vdash_{\mathcal{X}} C_{i+1}$. Thus, let

$$\begin{aligned}
\mathrm{VALC}(\mathcal{X}) &= \{\operatorname{enc}(C) \mid C \text{ is a valid computation of } \mathcal{X}\},\\
\mathrm{INVALC}(\mathcal{X}) &= \{0, \texttt{\#}\}^* \setminus \mathrm{VALC}(\mathcal{X}).
\end{aligned}$$

The main part of the proof of Theorem 10 is Theorem 14 (still further down), which states that, given an extended Turing machine $\mathcal{X}$, one can effectively construct an expression from RegEx(1) that generates INVALC($\mathcal{X}$). Note that in enc(C), `##` serves as a boundary between the encodings of individual configurations, which will be of use in the proof of Theorem 14. Building on Convention 12, we observe the following fact on the regularity of VALC($\mathcal{X}$) for a given extended Turing machine $\mathcal{X}$:

Lemma 13 *For every extended Turing machine $\mathcal{X}$,* VALC($\mathcal{X}$) *is regular if and only if* $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ *is finite.*

Proof The *if* direction follows immediately: As $\mathcal{X}$ is deterministic and accepts by halting, every word in VALC($\mathcal{X}$) corresponds to exactly one word from $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ (and the computation of $\mathcal{X}$ on that word). Thus, if $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ 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 $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ 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

$$L_{\mathcal{X}} := \{0^{\mathrm{e}(t_R)}\texttt{\#}0^{\mathrm{e}(t_R)} \mid t_R \in \mathrm{dom}_{\mathrm{X}}(\mathcal{X})\}.$$

Due to $L_{\mathcal{X}}$ being an infinite subset of $\{0^n\texttt{\#}0^n \mid n \geq 0\}$, we can then obtain a contradiction using the Pumping Lemma. In order to achieve this result, we use our convention that $\mathcal{M}$ does not halt on the very first configuration (cf. Convention 12).

As $\mathcal{X}$ is deterministic, every word $w \in$ VALC($\mathcal{X}$) corresponds to exactly one tape word $t_R \in \mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ 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_j). Recall that, by Convention 12, $\delta(0, q_1) = (0, L, q_2)$. Using the first equation in Observation 1, we conclude $\mathrm{e}(t'_L) = 0$, $\mathrm{e}(t'_R) = 2\,\mathrm{e}(t_R)$, and $a' = 0$. This means that there are $w_p, w' \in \{0, \texttt{\#}\}^*$ such that $w = w_p w'$, and

$$w_p = \texttt{\#\#}0\underbrace{\quad}_{\mathrm{e}(t_L)}\texttt{\#}0\underbrace{0^{\mathrm{e}(t_R)}}_{\mathrm{e}(t_R)}\texttt{\#}0\underbrace{\quad}_{a}\texttt{\#}\underbrace{0}_{q_1}\texttt{\#\#}0\underbrace{\quad}_{\mathrm{e}(t'_L)}\texttt{\#}0\underbrace{0^{\mathrm{e}(t'_R)}}_{\mathrm{e}(t'_R)}\texttt{\#}0\underbrace{\quad}_{a'}\texttt{\#}\underbrace{0^2}_{q_2}\texttt{\#\#}. \quad (1)$$

We now define a GSM (generalized sequential machine, cf. Hopcroft and Ullman [22]) M to transform VALC($\mathcal{X}$) into the language $L_{\mathcal{X}}$. 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 $L_{\mathcal{X}}$ then follows from the assumed regularity of VALC($\mathcal{X}$).

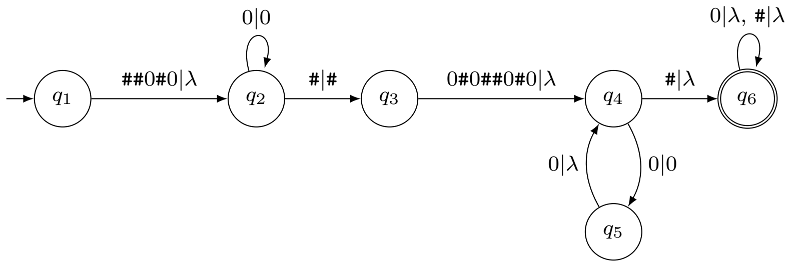

Fig. 2 The GSM M that is used in the proof of Lemma 13. Every transition shows the string that is read to the left of the | symbol, and the emitted string to the right. First, M erases ##0#0 and keeps the following continuous block of 0s and the # after it. It then erases 0#0##0#0 and halves the number of 0s in the next continuous block of 0s (using the loop between q_4 and q_5). After that block (as recognizable by #), all following letters are erased. Note that M relies on the fact that it is only used on words $w = w_p w'$, where w_p of the form that is described in (1). For all such words, w' is completely erased in the loop in q_6.

The GSM M is defined by the transition diagram in Figure 2. Compared to Hopcroft and Ullman [22], 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.

It is easily seen that $M(\mathrm{VALC}(\mathcal{X})) = L_{\mathcal{X}}$. By our initial assumption, $\mathrm{VALC}(\mathcal{X})$ is regular, and as the class of regular languages is closed under GSM mappings, $L_{\mathcal{X}}$ must be regular as well. Also by our initial assumption, $\mathrm{dom}_{\mathrm{X}}(\mathcal{X})$ is infinite, which means that $L_{\mathcal{X}}$ is an infinite subset of $\{0^n\#0^n \mid n \geq 0\}$. Using the Pumping Lemma (cf. Hopcroft and Ullman [22]), we can obtain the intended contradiction, as pumping any sufficiently large word from $L_{\mathcal{X}}$ would lead to a word that is not a subset of $0^*\#0^*$, or to a word $0^m\#0^n$ with $m \neq n$. □

We are now ready to state the central part of our proof of Theorem 10:

Theorem 14 *For every extended Turing machine $\mathcal{X}$, one can effectively construct an extended regular expression $\alpha_{\mathcal{X}} \in \mathrm{RegEx}(1)$ such that $L(\alpha_{\mathcal{X}}) = \mathrm{INVALC}(\mathcal{X})$.*

Proof Let $\mathcal{X} = (Q, q_1, \delta)$ be an extended Turing machine. Let $\nu \geq 2$ denote the number of states of $\mathcal{X}$; by Convention 12, $Q = \{q_1, \ldots, q_\nu\}$ for some $\nu \geq 2$. Intuitively, each element w of $\mathrm{INVALC}(\mathcal{X})$ contains at least one error that prevents w from being an encoding of a valid computation of $\mathcal{X}$. 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 $\mathcal{X}$ 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, 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

$$\alpha_{\mathcal{X}} := \alpha_{struc} \mid \alpha_{beha},$$

where the subexpressions α_{struc} and α_{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 α_{beha}; most of the encoding of INVALC($\mathcal{X}$) can be achieved with proper regular expressions. In order to define α_{struc}, we take a short detour and consider the language

$$\begin{aligned} S_{\mathcal{X}} := &\left(\texttt{##}0^{+}\texttt{#}0^{+}\texttt{#}0\{\lambda, 0\}\texttt{#}\left\{0^{i} \mid 1 \leq i \leq \nu\right\}\right)^{+}\texttt{##} \\ &\cap \texttt{##}0\texttt{#}0^{+}\texttt{#}0\texttt{#}0\texttt{##}\{0, \texttt{#}\}^{*} \\ &\cap \{0, \texttt{#}\}^{*}\texttt{#}\left\{00^{a}\texttt{#}0^{i}\texttt{##} \mid a \in \Gamma, \delta(a, q_i) = \mathrm{HALT}\right\}. \end{aligned}$$

Note that $S_{\mathcal{X}}$ is exactly the set of all enc(C), where $C = (C_i)_{i=1}^{n}$ (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 12 that $\mathcal{X}$ cannot halt in the first step). In other words, all that distinguishes $S_{\mathcal{X}}$ from 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, VALC($\mathcal{X}$) $\subseteq S_{\mathcal{X}}$.

Furthermore, $S_{\mathcal{X}}$ is a regular language, as it is obtained by an intersection of three regular languages. Thus, $\{0, \texttt{#}\}^{*} \setminus S_{\mathcal{X}}$ is also a regular language, and we define α_{struc} to be any proper regular expression with $L(\alpha_{struc}) = \{0, \texttt{#}\}^{*} \setminus S_{\mathcal{X}}$. It is easy to see that such an α_{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 ν and the halting instructions occurring in δ and can be constructed effectively, as can all the conversions that lead to α_{struc} (again, cf. Hopcroft and Ullman [22]). The exact shape of α_{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, VALC($\mathcal{X}$) $\subseteq S$, and thus, INVALC($\mathcal{X}$) $\supseteq L(\alpha_{struc})$. Furthermore, all elements of INVALC($\mathcal{X}$) $\setminus L(\alpha_{struc})$ are elements of $S_{\mathcal{X}}$ and encode a sequence $(C_i)_{i=1}^{n}$ ($n \geq 2$) of configurations of $\mathcal{X}$ such that $C_i \vdash_{\mathcal{X}} C_{i+1}$ does not hold for at least one i, $1 \leq i < n$.

Thus, INVALC($\mathcal{X}$) $\setminus L(\alpha_{struc})$ contains exactly those words from $S_{\mathcal{X}}$ that encode a sequence of configurations with at least one behavioral error. Therefore, when defining α_{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 $L(\alpha_{struc})$. This allows us to reason about the yet to be defined elements of INVALC($\mathcal{X}$) purely in terms of the execution of $\mathcal{X}$, as the encoding is already provided by the structure of $S_{\mathcal{X}}$, 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 $\mathrm{e}(t'_L)$ or $\mathrm{e}(t'_R)$ being different from the value that is expected according to Observation 1).

Each of these types of errors shall be handled by an expression α_{state}, α_{head} or α_{tape} (respectively, of course), and we define

$$\alpha_{beha} := (\alpha_{state} \mid \alpha_{head} \mid \alpha_{tape}).$$

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 α_{state} (thus, we do not need to consider HALT-instructions in α_{head} and α_{tape}). We can already note that α_{head} and α_{state} are proper regular expressions, as variables and the % metacharacter occur only in α_{tape} (recall that, as $\alpha_{\mathcal{X}} \in \mathrm{RegEx}(1)$, we are only allowed to use % once in the whole expression).

State errors: We begin with the definition of α_{state}. For every $a \in \Gamma$ and every i with $q_i \in Q$, we define a proper regular expression $\alpha^{state}_{a,i}$, and let

$$\alpha_{state} := \left(\alpha^{state}_{0,1} \mid \alpha^{state}_{1,1} \mid \alpha^{state}_{0,2} \mid \alpha^{state}_{1,2} \mid \cdots \mid \alpha^{state}_{0,\nu} \mid \alpha^{state}_{1,\nu}\right),$$

where each $\alpha^{state}_{a,i}$ lists all 'forbidden' follower states for q_i on a. More formally, if $\delta(a, q_i) = \mathrm{HALT}$, let

$$\alpha^{state}_{a,i} := (0 \mid \texttt{\#})^* \texttt{\#}00^a\texttt{\#}0^i\texttt{\#\#}0(0 \mid \texttt{\#})^*.$$

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 α_{struc}. Due to the definition of enc, every ## in words from $S_{\mathcal{X}}$ marks the boundary between two encoded configurations, and every string $\texttt{\#}0^i$ immediately to the left of such a ## encodes a state q_i. Likewise, when continuing to the left, $\texttt{\#}00^a$ encodes the head letter a. Thus, whenever a word from $S_{\mathcal{X}}$ contains a string $\texttt{\#}00^a\texttt{\#}0^i\texttt{\#\#}$, there is a configuration where $\mathcal{X}$ is in state q_i and reads a. As $\delta(a, q_i) = \mathrm{HALT}$, there may not be a succeeding configuration, and this definition of $\alpha^{state}_{a,i}$ 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 $L(\alpha_{struc})$.

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

$$\alpha^{state}_{a,i} := (0 \mid \texttt{\#})^* \texttt{\#}00^a\texttt{\#}0^i\texttt{\#\#}0^+\texttt{\#}0^+\texttt{\#}0^+\texttt{\#}\alpha^{not}_j\texttt{\#\#}(0 \mid \texttt{\#})^*,$$

where α_j^{not} is any proper regular expression with

$$L(\alpha_j^{not}) = \{0^k \mid 1 \leq k \leq \nu \text{ and } k \neq j\}.$$

Again, we use $\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}$ to identify an encoding of a configuration with head letter a in state q_i. To the right of $\mathtt{\#\#}$, the subexpression $0^+\mathtt{\#}0^+\mathtt{\#}0^+\mathtt{\#}$ 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 α_j^{not}. Thus, if a word from $S_{\mathcal{X}}$ contains any state error when reading a in q_i, the whole word belongs to $\alpha_{a,i}^{state}$, and $\alpha_{a,i}^{state}$ only matches such words.

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

$$\begin{aligned}\alpha_{a,i}^{state} := &\big((0 \mid \mathtt{\#})^*\mathtt{\#}0\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}0^+\mathtt{\#}0^+\mathtt{\#}0^+\mathtt{\#}\alpha_j^{not}\mathtt{\#\#}(0 \mid \mathtt{\#})^*\big)\\ &\mid \big((0 \mid \mathtt{\#})^*\mathtt{\#}00^+\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}0^+\mathtt{\#}0^+\mathtt{\#}0^+\mathtt{\#}\alpha_i^{not}\mathtt{\#\#}(0 \mid \mathtt{\#})^*\big),\end{aligned}$$

where α_j^{not} is defined as in the preceding paragraph. This expression is slightly more complicated, as it needs to distinguish two cases. Recall that the CHECK_R-instruction is to be interpreted as follows: If $t_R = 0^\omega$, $\mathcal{X}$ is supposed to change into state q_j; and if $t_R \neq 0^\omega$, $\mathcal{X}$ is supposed to remain 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 $\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}$ to identify configurations where $\mathcal{X}$ is in state q_i reading a.

In the first case, the string $\mathtt{\#}0\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}$ contains the additional information that $\mathrm{e}(t_R) = 0$, 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 $0^+\mathtt{\#}0^+\mathtt{\#}0^+\mathtt{\#}\alpha_j^{not}\mathtt{\#}$) and then matches all states but q_j.

Likewise, in the second case, $\mathtt{\#}00^+\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}$ matches all cases where (when reading a in q_i) $\mathrm{e}(t_R) > 0$, 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 α_i^{not} to identify all states that are not the correct successor state q_i.

Head errors: As α_{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

$$\alpha_{head} := \big(\alpha_{0,1}^{head} \mid \alpha_{1,1}^{head} \mid \alpha_{0,2}^{head} \mid \alpha_{1,2}^{head} \mid \cdots \mid \alpha_{0,\nu}^{head} \mid \alpha_{1,\nu}^{head}\big),$$

omitting those $\alpha_{a,i}^{head}$ 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 $\alpha_{a,i}^{head}$ as follows.

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

$$\begin{aligned}\alpha_{a,i}^{head} := &\big((0 \mid \mathtt{\#})^*\mathtt{\#}0(00)^*\mathtt{\#}0^+\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}0^+\mathtt{\#}0^+\mathtt{\#}00\mathtt{\#}(0 \mid \mathtt{\#})^*\big)\\ &\mid \big((0 \mid \mathtt{\#})^*\mathtt{\#}00(00)^*\mathtt{\#}0^+\mathtt{\#}00^a\mathtt{\#}0^i\mathtt{\#\#}0^+\mathtt{\#}0^+\mathtt{\#}0\mathtt{\#}(0 \mid \mathtt{\#})^*\big).\end{aligned}$$

According to the first equation in Observation 1, after a left movement of the head, $a' = \mathrm{e}(t_L) \bmod 2$ must hold. The two lines in the $\alpha_{a,i}^{head}$ distinguish the two possible cases for $\mathrm{e}(t_L) \bmod 2$. In both cases, we once again identify a and q_i in the encoding using $\texttt{\#}00^a\texttt{\#}0^i\texttt{\#}$. In the first line, the expression ignores $\mathrm{e}(t_R)$ (using the 0^+ to the left of $\texttt{\#}00^a\texttt{\#}$), and describes all cases where $\mathrm{e}(t_L)$ is even (by the $(00)^*$ part of $\texttt{\#}0(00)^*$). To the right of $\texttt{\#\#}$, the expression skips t'_L and t'_R and finds $a' = 1$, thus exactly those cases where $\mathrm{e}(t_L)$ is even, but $a' = 1$. Likewise, the second line handles the cases where $\mathrm{e}(t_L)$ is odd, but $a' = 0$. As $a' \in \{0, 1\}$ is ensured by $S_{\mathcal{X}}$, 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

$$\begin{aligned}\alpha^{head}_{a,i} := &\big((0 \mid \texttt{\#})^*\texttt{\#}0(00)^*\texttt{\#}00^a\texttt{\#}0^i\texttt{\#\#}0^+\texttt{\#}0^+\texttt{\#}00\texttt{\#}(0 \mid \texttt{\#})^*\big)\\ &\mid \big((0 \mid \texttt{\#})^*\texttt{\#}00(00)^*\texttt{\#}00^a\texttt{\#}0^i\texttt{\#\#}0^+\texttt{\#}0^+\texttt{\#}0\texttt{\#}(0 \mid \texttt{\#})^*\big)\,.\end{aligned}$$

This expression uses the second equation in Observation 1, $a' = \mathrm{e}(t_R) \bmod 2$, 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 $\delta(a, q_i) = \mathrm{CHECK}_{\mathrm{R}}(q_j)$ for some $q_j \in Q$, we define

$$\alpha^{head}_{a,i} := (0 \mid \texttt{\#})^*\texttt{\#}00^a\texttt{\#}0^i\texttt{\#\#}0^+\texttt{\#}0^+\texttt{\#}00^{1-a}\texttt{\#}(0 \mid \texttt{\#})^*.$$

As $\mathrm{CHECK}_{\mathrm{R}}$-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_i (again using $\texttt{\#\#}$ 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 $\alpha^{head}_{a,i}$, we can use the shorthand notation 0^{1-a} without any formal problems (as it is just another notation for 0 or λ, depending on a).

Tape side errors: As mentioned above, α_{tape} shall be 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 α_{tape}.

In order to increase the readability, we shall define α_{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 to be worried about the fact that most of the following definitions contain $\%x$; as we shall see, the resulting expression α_{tape} 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 $\mathrm{CHECK}_{\mathrm{R}}$-instructions.

If $\delta(a, q_i)$ is a $\mathrm{CHECK_R}$-instruction, we define

$$\begin{aligned}
\alpha^{a,i}_{L,>} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%x \texttt{\#} 0^+ \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0x0^+ \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{L,<} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%x0^+ \texttt{\#} 0^+ \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0x \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{R,>} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%x \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0^+ \texttt{\#} 0x0^+ \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{R,<} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%x0^+ \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0^+ \texttt{\#} 0x \texttt{\#} (0 \mid \texttt{\#})^*.
\end{aligned}$$

Intuitively, for $M \in \{L, R\}$, $\alpha^{a,i}_{M,>}$ is used to describe all successor configurations (after reading a in q_i), where $\mathrm{e}(t'_M) > \mathrm{e}(t_M)$. Likewise, $\alpha^{a,i}_{M,<}$ handles all those cases where $\mathrm{e}(t'_M) < \mathrm{e}(t_M)$. We discuss the correctness of these expressions using $\alpha^{a,i}_{L,>}$ as an example, the three other expressions behave analogously. If $\alpha^{a,i}_{L,>}$ matches a word from $S_{\mathcal{X}}$, x is bound to some word 0^n with $n \geq 0$ (due to $(0^*)\%x$). As this 0^n 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 $\texttt{\#\#}0x0^+\texttt{\#}$ matches the block that encodes $\mathrm{e}(t'_L)$. As $\texttt{\#\#}00^n0^+\texttt{\#}$ is expanded to some $\texttt{\#\#}00^n0^m$ (with $m \geq 1$), we know that $\mathrm{e}(t'_L) = 0^m0^n > \mathrm{e}(t_L)$. Likewise, for every $\mathrm{e}(t'_L) > \mathrm{e}(t_L)$, we can find an appropriate $m \geq 1$ and expand 0^+ to 0^m. Thus, $\alpha^{a,i}_{L,>}$ matches exactly those cases in which $\mathcal{X}$ reads a in q_i, and the resulting $\mathrm{e}(t'_L)$ is larger than it should be (i. e., larger than $\mathrm{e}(t_L)$, as $\mathrm{CHECK_R}$-instructions do not change the tape).

The three other expressions behave analogously; but note that for $\alpha^{a,i}_{R,>}$ and $\alpha^{a,i}_{R,<}$, the subexpression $0(0^*)\%x$ matches the coding of $\mathrm{e}(t_R)$ instead of $\mathrm{e}(t_L)$, 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 1 allow us to compute $\mathrm{e}(t'_L)$ and $\mathrm{e}(t'_R)$ from $\mathrm{e}(t_L)$, $\mathrm{e}(t_R)$ and $\delta(a, q_i)$.

If $\delta(a, q_i) \in (b, L, q_j)$ for some $q_j \in Q$, we define

$$\begin{aligned}
\alpha^{a,i}_{L,>} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%xx(0 \mid \lambda) \texttt{\#} 0^+ \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0x0^+ \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{L,<} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%xx000^* \texttt{\#} 0^+ \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0x \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{R,>} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%x \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0^+ \texttt{\#} 0xx0^b0^+ \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{R,<} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0(0^*)\%x0^+ \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0^+ \texttt{\#} 0xx0^b \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{mod} &:= (0 \mid \texttt{\#})^* \texttt{\#} 00^a \texttt{\#} 0^i \texttt{\#\#} 0^+ \texttt{\#} 0(00)^* 0^{1-b} \texttt{\#} (0 \mid \texttt{\#})^*.
\end{aligned}$$

These expressions fulfill the same purpose as their equally named counterparts we defined to handle tape side errors for configurations in which a $\mathrm{CHECK_R}$-instruction is executed, i. e., they describe the cases where $\mathrm{e}(t'_L)$ or $\mathrm{e}(t'_R)$ contains too much or too little. For technical reasons that shall be explained a little later, we also use a proper regular expression $\alpha^{a,i}_{mod}$ to describe the errors where $\mathrm{e}(t_R)$ has the wrong parity.

We begin with the expressions that handle errors on the left tape side. According to the first equation in Observation 1, the correct t'_L is characterized by $\mathrm{e}(t'_L) = \mathrm{e}(t_L) \operatorname{div} 2$.

First, we consider $\alpha^{a,i}_{L,>}$. As before, a and q_i are matched in $\texttt{\#}00^a\texttt{\#}0^i\texttt{\#\#}$. To the left of that block, the expression skips the encoding of $\mathrm{e}(t_R)$, and matches $00^{\mathrm{e}(t_L)}$ with the expression $0(0^*)\%xx(0 \mid \lambda)$. Note that x is bound to $0^{\mathrm{e}(t_L) \operatorname{div} 2}$, and $0^{\mathrm{e}(t_L) \bmod 2}$ matches 0 or λ in $(0 \mid \lambda)$. To the right of $\texttt{\#\#}$, $0x0^+$ matches the encoding of $00^{\mathrm{e}(t'_L)}$. It is easily seen that this describes exactly those cases where $\mathrm{e}(t'_L) > (\mathrm{e}(t_L) \operatorname{div} 2)$, as the 0^+ is used to match all possible values of $\mathrm{e}(t'_L) - (\mathrm{e}(t_L) \operatorname{div} 2)$.

Next, consider $\alpha^{a,i}_{L,<}$. Note that if $\mathrm{e}(t'_L) = (\mathrm{e}(t_L) \operatorname{div} 2)$, either $\mathrm{e}(t_L) = 2\,\mathrm{e}(t'_L)$ or $\mathrm{e}(t_L) = 2\,\mathrm{e}(t'_L) + 1$ holds. Thus, $\mathrm{e}(t'_L)$ is too small if and only if $\mathrm{e}(t_L) > 2\,\mathrm{e}(t'_L) + 1$, which holds if and only if $\mathrm{e}(t_L) = 2\,\mathrm{e}(t'_L) + 2 + m$ for some $m \geq 0$. We can easily see that (for words from $S_{\mathcal{X}}$), $\alpha^{a,i}_{L,<}$ matches exactly those encodings of successive configurations (t_L, t_R, a, q_i) and (t'_L, t_R, a', q_j), where $\mathrm{e}(t_L) = 2n + 2 + m$ and $\mathrm{e}(t'_L) = n$ for some $m, n \geq 0$, as x binds to 0^n, and $0(0^*)\%xx000^*$ corresponds to $00^n0^n000^m = 00^{\mathrm{e}(t_L)}$. Thus, this expression handles exactly those cases where $\mathrm{e}(t'_L)$ is too small.

Hence, $\alpha^{a,i}_{L,>}$ and $\alpha^{a,i}_{L,<}$ 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 1), the correct right tape word t'_R is characterized by $\mathrm{e}(t'_R) = 2\,\mathrm{e}(t_R) + b$.

It is easy to see that $\alpha^{a,i}_{R,>}$ handles exactly those words (from $S_{\mathcal{X}}$, with $\mathcal{X}$ reading a in q_i) where $\mathrm{e}(t'_R)$ is too large. First, x is bound to $0^{\mathrm{e}(t_R)}$. For t'_R, we have $0\,xx0^b0^+$, and thus, $\mathrm{e}(t'_R) = 2\,\mathrm{e}(t_R) + b + n$, with $n \geq 1$, where 0^+ expresses the difference between $\mathrm{e}(t'_R)$ 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 $\mathrm{e}(t'_R) \bmod 2 = b$ must hold. The expression $\alpha^{a,i}_{mod}$ describes those cases where this condition is not satisfied; i. e., $\mathcal{X}$ reads a in q_i, but $\mathrm{e}(t'_R) \bmod 2 \neq b$. Therefore, we can restrict our definition of $\alpha^{a,i}_{R,<}$ to those cases where $\mathrm{e}(t'_R)$ and b have the same parity. In these cases, $\mathrm{e}(t'_R) = 2n + b$ for some $n \geq 0$, but $\mathrm{e}(t_R) > n$, which holds if and only if there is an $m \geq 0$ with $\mathrm{e}(t_R) = n + m + 1$. The words from $S_{\mathcal{X}}$ that match $\alpha^{a,i}_{R,<}$ (but not $\alpha^{a,i}_{mod}$) are exactly those that satisfy this condition: The variable x is bound to 0^n, while 0^+ corresponds to $m + 1$. Thus, $\alpha^{a,i}_{mod} \mid \alpha^{a,i}_{R,<}$ handles the cases where $\mathrm{e}(t'_R)$ is too small (but not exactly those cases, even when restricted to $S_{\mathcal{X}}$, as $\alpha^{a,i}_{mod}$ also matches encodings of computations where $\mathrm{e}(t'_R)$ 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

$$\begin{aligned}
\alpha^{a,i}_{R,>} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0 (0^*) \% x x (0 \mid \lambda) \texttt{\#} 0 0^a \texttt{\#} 0^i \texttt{\#\#} 0^+ \texttt{\#} 0 x 0^+ \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{R,<} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0 (0^*) \% x x 0 0 0^* \texttt{\#} 0 0^a \texttt{\#} 0^i \texttt{\#\#} 0^+ \texttt{\#} 0 x \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{L,>} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0 (0^*) \% x \texttt{\#} 0^+ \texttt{\#} 0 0^a \texttt{\#} 0^i \texttt{\#\#} 0 x x 0^b 0^+ \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{L,<} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0 (0^*) \% x 0^+ \texttt{\#} 0^+ \texttt{\#} 0 0^a \texttt{\#} 0^i \texttt{\#\#} 0 x x 0^b \texttt{\#} (0 \mid \texttt{\#})^*,\\
\alpha^{a,i}_{mod} &:= (0 \mid \texttt{\#})^* \texttt{\#} 0 0^a \texttt{\#} 0^i \texttt{\#\#} 0 (00)^* 0^{1-b} \texttt{\#} (0 \mid \texttt{\#})^*.
\end{aligned}$$

As already suggested by the similarities (of the equations in Observation 1, 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 $\mathcal{X}$ reads a in state q_i (again assuming that we only consider words from $S_{\mathcal{X}}$).

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

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

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

Next, note that all other expressions for tape side errors use exactly one variable binding, and start with the common prefix $(0 \mid \texttt{\#})^* \texttt{\#} 0 (0^*) \% x$. 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

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

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

$$\begin{aligned}
\alpha_{var} = (0 \mid \texttt{\#})^* \texttt{\#} 0 (0^*) \% x \Big(& \hat{\alpha}^{0,1}_{L,>} \mid \hat{\alpha}^{0,1}_{L,<} \mid \hat{\alpha}^{0,1}_{R,>} \mid \hat{\alpha}^{0,1}_{R,<} \mid \hat{\alpha}^{1,1}_{L,>} \mid \hat{\alpha}^{1,1}_{L,<} \mid \hat{\alpha}^{1,1}_{R,>} \mid \hat{\alpha}^{1,1}_{R,<} \mid \\
& \cdots \\
& \mid \hat{\alpha}^{0,\nu}_{L,>} \mid \hat{\alpha}^{0,\nu}_{L,<} \mid \hat{\alpha}^{0,\nu}_{R,>} \mid \hat{\alpha}^{0,\nu}_{R,<} \mid \hat{\alpha}^{1,\nu}_{L,>} \mid \hat{\alpha}^{1,\nu}_{L,<} \mid \hat{\alpha}^{1,\nu}_{R,>} \mid \hat{\alpha}^{1,\nu}_{R,<} \Big),
\end{aligned}$$

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

$S_{\mathcal{X}}$ that encode a tape side error at some point of the encoded computation. Therefore, for every word $w \in \{0, \texttt{\#}\}^*$, $w \in L(\alpha_{\mathcal{X}})$ 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 $w \in L(\alpha_{\mathcal{X}})$ if and only $w \in \text{INVALC}(\mathcal{X})$, and thereby, $L(\alpha_{\mathcal{X}}) = \text{INVALC}(\mathcal{X})$.

Finally, as this proof defines $\alpha_{\mathcal{X}}$ constructively, using only ν and δ, it also describes an effective procedure to compute $\alpha_{\mathcal{X}}$ from $\mathcal{X}$. This concludes the proof of Theorem 14. $\square$

Theorem 10 follows almost immediately from Theorem 14, and Lemmas 11 and 13:

Proof (of Theorem 10) We prove each of the claims by reduction from one of three problems for extended Turing machines that are listed in Lemma 11 (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 14).

Then $\text{dom}_{\text{X}}(\mathcal{X}) = \emptyset$ if and only if $\text{VALC}(\mathcal{X}) = \emptyset$, which holds if and only if $\text{INVALC}(\mathcal{X}) = \{0, \texttt{\#}\}^*$, which holds if and only if $L(\alpha_{\mathcal{X}}) = \{0, \texttt{\#}\}^*$, which holds if and only if $L(\alpha) \supseteq \{0, \texttt{\#}\}^*$. Thus, any algorithm that decides universality for RegEx(1) could be used to decide the emptiness of the domain for extended Turing machines, which is undecidable according to Lemma 11.

Furthermore, $\text{dom}_{\text{X}}(\mathcal{X})$ 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 13), 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 RegEx(1) would lead to semi-decidability of finiteness of dom_{X}, a problem that is not semi-decidable (according to Lemma 11)

Likewise, as $\text{dom}_{\text{X}}(\mathcal{X})$ is finite if and only if $L(\alpha_{\mathcal{X}})$ is regular, $\text{dom}_{\text{X}}(\mathcal{X})$ is infinite if and only if $L(\alpha_{\mathcal{X}})$ is not regular. Therefore, semi-decidability of non-regularity for RegEx(1) contradicts the fact that the complement of finiteness of dom_{X} is not semi-decidable (see Lemma 11).

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 RegEx(1). $\square$

Those who are interested in the exact position of these problems in the arithmetical hierarchy (cf. Odifreddi [28]) can conclude that universality is Π^0_1-complete, while regularity and co-finiteness are Σ^0_2-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 10 (see the remark after the proof of Lemma 11). 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 Π^0_1. Likewise, non-regularity can be expressed as $\forall \beta \in \mathrm{RegEx}(0) : \exists w \in \Sigma^* : [w \in L(\alpha) \Leftrightarrow w \notin L(\beta)]$, which shows that non-regularity is in Π^0_2. Actually, under a strict interpretation of the definition given by Odifreddi [28], we would need to quantify over natural numbers; but as there are computable bijections between $\mathbb{N}$ and Σ^*, as well as between $\mathbb{N}$ and RegEx(0), we can omit this technical detail.

4 Consequences of Theorem 10

In Section 3, we introduced and proved Theorem 10, which states that for RegEx(1), universality is not semi-decidable; and regularity and cofiniteness are neither semi-decidable, nor co-semi-decidable.

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 10 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 10, 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 10, 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 10 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 by Della Penna et al. [14]. We discuss an adaption to another model in Section 4.2.

In addition to this, the construction from Theorem 14 (which we used to prove Theorem 10, 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 4.1.

From the undecidability of universality, we can immediately conclude that RegEx(1) cannot be minimized effectively:

Corollary 15 *Let c be a complexity measure for* RegEx(1)*. Then there is no recursive function m_c that, given an expression $\alpha \in$* RegEx(1)*, returns an expression $m_c(\alpha) \in$* RegEx(1) *with 1. $L(m_c(\alpha)) = L(\alpha)$, and 2. $c(\beta) \geq c(m_c(\alpha))$ for every $\beta \in$* RegEx(1) *with $L(\beta) = L(\alpha)$.*

Proof Let c be a complexity measure for RegEx(1) and assume there is such a function m_c. Let Σ 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 10). This is a contradiction. □

Following the classic proof method of Hartmanis [19] (cf. Kutrib [24]), 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:

Corollary 16 *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)*.*

Proof The result for RegEx(1) and RegEx(0) follows immediately from Theorem 10 and Theorem 4 in Hartmanis [19]: As non-regularity is not semi-decidable for RegEx(1), the tradeoff between RegEx(1) and RegEx(0) is non-recursive (see also Kutrib [24] for more detailed explanations than in Hartmanis [19]).

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

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

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 Δ as follows: Given some $\alpha \in \mathrm{RegEx}(1)$, we compute $n := f_c(c(\alpha))$, and let

$$F_n := \{\beta \in \mathrm{RegEx}(0) \mid c(\beta) \leq n\}.$$

As c is a complexity measure, F_n is finite, and we can effectively list all its elements (as we can effectively list all $\beta \in \mathrm{RegEx}(1)$ with $c(\beta) \leq n$, and we can decide whether $\beta \in \mathrm{RegEx}(0)$ by searching β 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_β in finite time that proves $L(\beta) \neq L(\alpha)$ (as w_β 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 β can be found, we know that $\alpha \in \Delta$, and the procedure can return 1. Thus, we can construct a semi-decision procedure for Δ, which contradicts the established fact that Δ 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 10, 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

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

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 10). We can construct a semi-decision procedure for Δ_C from f_c 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 RegEx(0) (e. g., given a $\beta \in \text{RegEx}(0)$, one could convert β 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, *mutatis mutandis.* □

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:

Lemma 17 *For every $k \geq 1$, the tradeoff between* FRegEx(k) *and* FRegEx(0) *is recursive (even when considering complexity measures for* RegEx(k) *instead of* FRegEx(k)*).*

Proof Let $k \geq 1$, and let c be a complexity measure for 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 α, and computing the corresponding word for each combination, handling all bindings accordingly.

For example, the expression $\alpha{:=}(\mathtt{a} \mid \mathtt{b})(\mathtt{a} \mid \mathtt{b})\%x(\mathtt{a} \mid \mathtt{b})\%y\, x\, y$ 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 bound $f_c : \mathbb{N} \to \mathbb{N}$ 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 expressions from F_n (again, as c is a complexity measure, and membership in FRegEx(k) is straightforwardly decidable for 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_n is a recursive function. Furthermore, f_c is a bound on the tradeoff from FRegEx($k+1$) to FRegEx(k) by definition. □

Although the class of RegEx-languages is not closed under complementation (Lemma 2 in Câmpeanu et al. [9]), there are languages L such that both L and its complement $\Sigma^* \setminus L$ are RegEx-languages (e. g., all regular languages). Combining Lemma 17 and Corollary 16, 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:

Corollary 18 *Let Σ be a finite alphabet. Let c be a complexity measure for* RegEx(1). *For any recursive function $f_c : \mathbb{N} \to \mathbb{N}$, there exists an $\alpha \in$* RegEx(1) *such that $\Sigma^* \setminus L(\alpha)$ is a* RegEx(1)*-language, and for every $\beta \in$* RegEx(1) *with $L(\beta) = \Sigma^* \setminus L(\alpha)$, $c(\beta) \geq f_c(c(\alpha))$.*

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

We can now use f_1 and Lemma 17 to obtain a recursive bound on the tradeoff between CoFRegEx(1) and CoFRegEx(0), which contradicts Corollary 16.

First, note that according to Lemma 17, there is a recursive bound $f_2 : \mathbb{N} \to \mathbb{N}$ on the tradeoff between FRegEx(1) and 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 FRegEx(1), as f_3 might be computed as follows: For every input n, there are finitely many $\alpha \in$ RegEx(0) with $c(\alpha) = n$. Finiteness for RegEx(0) is obviously decidable; thus, we can effectively construct the finite set

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

For every $\alpha \in F_n$, we (effectively) construct an $\overline{\alpha} \in$ RegEx(0) with $L(\overline{\alpha}) = \Sigma^* \setminus L(\alpha)$, for example, by converting α 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 [22]. We then check all $\beta \in$ RegEx(0), ordered by growing size of $c(\beta)$, until we find the smallest β (w.r.t. c) with $L(\beta) = \Sigma^* \setminus L(\alpha)$, and refer to this β as $\tilde{\alpha}$ (again, this is possible due to the decidability of equivalence for RegEx(0), cf. Hopcroft and Ullman [22]), and define

$$C_n := \{\widetilde{\alpha_n} \mid \alpha_n \in F_n\}.$$

$$\begin{tikzcd}
\mathrm{CoFRegEx}(1) \arrow[r, "f"] \arrow[d, "f_1"'] & \mathrm{CoFRegEx}(0) \\
\mathrm{FRegEx}(1) \arrow[r, "f_2"'] & \mathrm{FRegEx}(0) \arrow[u, "f_3"']
\end{tikzcd}$$

Fig. 3 An illustration of the functions that are used in the proof of Corollary 18.

Finally, we define $f_3(n)$ to be the maximum of all $c(\tilde{\beta})$ 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 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 CoFRegEx(1) and CoFRegEx(0), which contradicts Corollary 16. An illustration of this argument can be found in Figure 3.

First, observe that f is a recursive function, as f_1, f_2, and f_3 are recursive functions. Due to Corollary 16, there is an $\alpha_{fc} \in$ CoFRegEx(1) such that $L(\alpha_{fc})$ is regular, but for every $\beta \in$ RegEx(0) (and hence, $\beta \in$ CoFRegEx(0)) with $L(\alpha_{fc}) = L(\beta)$, $c(\beta) > f(c(\alpha_{fc}))$.

By our assumption, there is an $\overline{\alpha}_{fc} \in$ RegEx(1) with $L(\overline{\alpha}_{fc}) = \Sigma^* \setminus L(\alpha_{fc})$ and $c(\overline{\alpha}_{fc}) \leq f_1(c(\alpha_{fc}))$. As $\alpha_{fc} \in$ CoFRegEx(1), $L(\alpha_{fc})$ is cofinite, hence, $L(\overline{\alpha}_{fc})$ is finite, and $\overline{\alpha}_{fc} \in$ FRegEx(1).

According to Lemma 17, there is a $\overline{\beta}_{fc} \in$ FRegEx(0) with $L(\overline{\beta}_{fc}) = L(\overline{\alpha}_{fc}) = \Sigma^* \setminus L(\alpha_{fc})$, and $c(\overline{\beta}_{fc}) \leq f_2(c(\overline{\alpha}_{fc}))$.

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

$$\begin{aligned}
c(\beta_{fc}) &\leq f_3(c(\overline{\beta}_{fc})) \\
&\leq f_3(f_2(c(\overline{\alpha}_{fc}))) \\
&\leq f_3(f_2(f_1(c(\alpha_{fc})))) \\
&= f(c(\alpha_{fc})).
\end{aligned}$$

This contradicts our choice of α_{fc} and concludes the proof. □

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

Lemma 19 *Let $k \geq 1$. For* RegEx($k+1$)*,* RegEx(k)*-ity is neither semi-decidable, nor co-semi-decidable.*

Proof We adapt the construction from the proof of Theorem 14 to the larger alphabet $\Sigma_k{:=}\{0, \texttt{\#}, \$, \texttt{a}_1, \texttt{b}_1, \ldots, \texttt{a}_k, \texttt{b}_k\}$, where 0, #, \$, all $\texttt{a}_i$, and all $\texttt{b}_i$ are pairwise distinct letters. For every i with $1 \leq i \leq k$, let $\Sigma_i{:=}\{\texttt{a}_i, \texttt{b}_i\}$. Given an extended Turing machine $\mathcal{X}$, we construct the extended regular expression $\alpha_{\mathcal{X}} \in \mathrm{RegEx}(1)$ as in the proof of Theorem 14. For every i with $1 \leq i \leq k$, we define an expression $\alpha_i \in \mathrm{RegEx}(1)$ by

$$\alpha_i{:=}((\mathtt{a}_i \mid \mathtt{b}_i)^*)\%x_i x_i,$$

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 $\alpha_{\mathcal{X},k} \in \mathrm{RegEx}(k+1)$ by

$$\begin{aligned}\alpha_{\mathcal{X},k}&{:=}\alpha_{\mathcal{X}}\$\alpha_1\$\cdots\$\alpha_k\\ &= \alpha_{\mathcal{X}}\$((\mathtt{a}_1 \mid \mathtt{b}_1)^+)\%x_1x_1\$\cdots\$((\mathtt{a}_k \mid \mathtt{b}_k)^+)\%x_kx_k.\end{aligned}$$

Thus, $\alpha_{k,\mathcal{X}} \in \mathrm{RegEx}(k+1)$. It suffices to show that $L(\alpha_{k,\mathcal{X}})$ 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 10.

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 [22]). 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 $\alpha \in \mathrm{RegEx}(k-1)$ with $L(\alpha) = L_k$. By definition, α contains at most $k-1$ different variables. Note that, whenever α is matched to a $w \in L_k$, every variable x in α that is bound and also referenced when matching w contains only terminals from a single set $\Sigma_i \cup \{\$\}$, as $L_k \subset \{\mathtt{a}_1, \mathtt{b}_1\}^+\$\cdots\$\{\mathtt{a}_k, \mathtt{b}_k\}^+$, and repeating any string that contains some $\mathtt{a}_j$ or $\mathtt{b}_j$ 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 $\alpha \in \mathrm{RegEx}(k-1)$ with $L(\alpha) = L_k$.

If $L(\alpha_{\mathcal{X}})$ is not regular, this reasoning extends to $L(\alpha_{k,\mathcal{X}})$ 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 10, this proves that for RegEx(k), RegEx(k)-ity is neither semi-decidable, nor co-semi-decidable. □

Non-recursive tradeoffs between RegEx($k+1$) and RegEx(k) for every $k \geq 1$ follow immediately, using Hartmanis' proof technique as in the proof of Corollary 16.

Although the proof of Lemma 19 uses an unbounded terminal alphabet $\Sigma_k{:=}\{0, \mathtt{\#}, \$, \mathtt{a}_1, \mathtt{b}_1, \ldots, \mathtt{a}_k, \mathtt{b}_k\}$, the construction can be adapted to an alphabet Σ of constant size in the following way. First, let $\Sigma{:=}\{0, \mathtt{\#}, \$, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and define a morphism $h_k : \Sigma_k^* \to \Sigma^*$ by $h(\mathtt{a}_i){:=}\,\mathtt{c}\,\mathtt{a}^i\,\mathtt{c}$ and $h(\mathtt{b}_i){:=}\,\mathtt{c}\,\mathtt{b}^i\,\mathtt{c}$ for all $1 \leq i \leq k$, and $h(a) = a$ for all $a \in \{0, \mathtt{\#}, \$\}$. Instead of the languages $L(\alpha_{\mathcal{X},k})$, we then consider the languages $h_k(L(\alpha_{\mathcal{X},k}))$. The reasoning then proceeds as in the original proof, with the sole exception that, instead of arguing that the variables must contain characteristic *letters* $\mathtt{a}_i$ and $\mathtt{b}_i$, they now contain characteristic *segments* $\mathtt{c}\,\mathtt{a}^i\,\mathtt{c}$ and $\mathtt{c}\,\mathtt{b}^i\,\mathtt{c}$.

Using the same approach, the proof can be adapted to binary terminal alphabets.

4.1 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 14 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 δ 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 [6], these limitations can be overcome by using a single universal Turing machine: First, let $\psi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a universal partially recursive function, i. e., for every partially recursive function $\phi : \mathbb{N} \to \mathbb{N}$, there is an $m \geq 0$ such that $\psi(m,n) = \phi(n)$ for every $n \geq 0$. Note that it is an elementary fact of recursion theory that such a function (which is often called a *numbering*) exists and can be defined constructively (cf. Cutland [13], Odifreddi [28]). For every $m \geq 0$, we define the function $\psi_m : \mathbb{N} \to \mathbb{N}$ by $\psi_m(n) := \psi(m,n)$ for all $n \geq 0$. Furthermore, we define the function

$$\mathrm{dom}(\psi_m) := \{n \in \mathbb{N} \mid \psi_m(n) \text{ is defined}\},$$

from which we derive the index sets

$$E_\psi := \{m \mid \mathrm{dom}(\psi_m) \text{ is empty}\}, \text{ and}$$
$$F_\psi := \{m \mid \mathrm{dom}(\psi_m) \text{ is finite}\}.$$

As in Lemma 11, one can use Rice's Theorem and its extension (again, cf. [13, 28]) to show that E_ψ is not semi-decidable, and F_ψ is neither semi-decidable, nor co-semi-decidable.

Moreover, there is a Turing machine $\mathcal{U}$ over some tape alphabet $\hat{\Gamma} \subseteq \{0, ¢, \$, \mathtt{a}, \mathtt{b}\}$ such that

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

Again, this machine can be defined constructively (e. g. from the constructive definition of ψ). Using the same construction as in the proof of Lemma 11, we can build an extended Turing machine $\hat{\mathcal{U}} = (Q, q_1, \delta)$ that simulates $\mathcal{U}$, using an appropriate injective function $b_k : \hat{\Gamma} \to \Gamma^k$ with $b_k(0) = 0^k$, 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 natural number m, using a slight modification of the proof of Theorem 14. 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 $S_{\mathcal{X}}$), we define

$$\begin{aligned} S_{\hat{\mathcal{U}},m} := & \left(\texttt{\#\#}0^{+}\texttt{\#}0^{+}\texttt{\#}0\{\lambda, 0\}\texttt{\#}\left\{0^{i} \mid 1 \leq i \leq \nu\right\}\right)^{+}\texttt{\#\#} \\ & \cap \texttt{\#\#}0\texttt{\#}0^{\mathrm{e}(b_k(\text{¢}\,\mathtt{a}^{m}\,0))}\left(0^{2^{(2+m)k}}\right)^{+}\texttt{\#}0\texttt{\#}0\texttt{\#\#}\{0, \texttt{\#}\}^{*} \\ & \cap \{0, \texttt{\#}\}^{*}\texttt{\#}\left\{00^{a}\texttt{\#}0^{i}\texttt{\#\#} \mid a \in \hat{\Gamma}, \delta(a, q_i) = \mathrm{HALT}\right\}. \end{aligned}$$

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

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

Hence, given an $m \in \mathbb{N}$, we can effectively construct an expression $\alpha_{\hat{\mathcal{U}},m}$ such that the language $L(\alpha_{\hat{\mathcal{U}},m})$ is

- universal iff $\mathrm{dom}(\psi_m)$ is empty iff $m \in E_\psi$,
- regular iff $\mathrm{dom}(\psi_m)$ is finite iff $m \in F_\psi$,
- cofinite iff $\mathrm{dom}(\psi_m)$ is finite iff $m \in F_\psi$,

which holds following the same reasoning as for $L(\alpha_{\mathcal{X}})$. Hence, a procedure that decides one of these problems for RegEx(1) with a bounded number of variable occurrences to some degree can be effectively transformed into procedure that decides one of the index sets E_ψ and F_ψ to the same degree.

As mentioned above, E_ψ and F_ψ are not semi-decidable, and F_ψ is also not co-semi-decidable. Hence, we arrive at the same conclusions as in Theorem 10 and Corollaries 16 and 18 for RegEx(1) with a bounded number of occurrences of the single variable.

4.2 A Note on H-Systems

As mentioned multiple times above, Theorem 14 — and hence, Theorem 10 and the resulting consequences — can be easily adapted to various other models that use similar repetition mechanisms. One of these models are the so-called H-expressions by Bordihn et al. [5]. These H-expressions are based on *H-systems*, which were introduced by Albert and Wegner [4].

In fact, instead of H-expressions, the construction can be implemented using these (less expressive) H-systems. An H-system is a 4-tuple $H = (X, \Sigma, L_1, \phi)$, where X and Σ are finite alphabets (the *meta alphabet* and the *terminal alphabet*, respectively), $L_1 \subseteq X^*$ is called the *meta language*, and $\phi : X \to \mathcal{P}(\Sigma^*)$ is a function that assigns to each $x \in X$ a language $\phi(x) = L_x \subseteq \Sigma^*$. The language of H is defined as

$$L(H) := \{h(w) \mid w \in L_1, h \text{ is a homomorphism with } h(x) \in \phi(x) \text{ for all } x \in X\}.$$

Less formally, every letter x from the meta alphabet is replaced uniformly with a word from $\phi(x)$.

Furthermore, if $\mathcal{L}_1$ and $\mathcal{L}_2$ are classes of languages, $\mathcal{H}(\mathcal{L}_1, \mathcal{L}_2)$ denotes the class of H-system languages of $\mathcal{L}_1$ and $\mathcal{L}_2$, i. e., the class of all languages that are generated by H-systems that use a language $L_1 \in \mathcal{L}_1$ as metalanguage, and have $\phi(x) \in \mathcal{L}_2$ for every x in their meta alphabet X.

It is easy to see that, for every extended Turing machine $\mathcal{X}$, the expressions $\alpha_{\mathcal{X}}$ that are constructed in the proof of Theorem 14 can be easily converted into an H-system $H = (X_H, \Sigma_H, L_1, \phi)$ with $L(H) = \text{INVALC}(\mathcal{X})$ and $L(H) \in \mathcal{H}(\text{REG}, \text{REG})$ by proceeding as follows.

First, we select $X_H := \{0, \texttt{\#}, x\}$ and $\Sigma_H := \{0, \texttt{\#}\}$. We then replace the single occurrence of $(0^*)\%x$ in $\alpha_{\mathcal{X}}$ with x, and obtain a proper regular expression $\hat{\alpha}_{\mathcal{X}}$ over the alphabet Σ_H. This allows us to choose $L_1 := L(\hat{\alpha}_{\mathcal{X}})$, while $L_1 \in$ REG holds. Next, we define $\phi(0) = \{0\}$, $\phi(\texttt{\#}) = \{\texttt{\#}\}$, and $\phi(x) := \{0\}^*$. Hence, $L(H) = L(\alpha_{\mathcal{X}}) = \text{INVALC}(\mathcal{X})$ follows immediately.

Thus, we can adapt these proofs to the class $\mathcal{H}(\text{REG}, \text{REG})$ of H-system languages, and conclude the same levels of undecidability of the respective decision problems for $\mathcal{H}(\text{REG}, \text{REG})$. As $\mathcal{H}(\text{REG}, \text{REG})$ is a subclass of the class of H-expression languages, this also proves that equivalence for H-expression languages is undecidable, a problem that was explicitly mentioned as open by Bordihn et al. [5].

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. As regular expressions are often precompiled, this heuristics might be employed as compiler-level optimization features, comparable to the compilation of programming languages.

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.

On the other hand, the author wishes to point out that the enormous relative succinctness of extended regular expressions should be considered an advantage, especially as the existence of an *effective* minimization procedure for proper regular expressions is probably of little practical use, considering the fact that there is no *efficient* minimization procedure (unless P=PSPACE). In particular, the author is convinced that the results on descriptional complexity emphasize the potential usefulness of large and natural subclasses of RegEx for which the membership problem can be decided efficiently, as opposed to the NP-completeness of the general problem. One possible example of such restrictions can be found in Reidenbach and Schmid [30].

Acknowledgements

The author wishes to thank Nicole Schweikardt and the anonymous referees for the conference version [15] and the present version for their helpful remarks.

References

1. Abigail. Re: Random number in perl. Posting in the newsgroup comp.lang.perl.misc, October 1997. Message-ID slrn64sudh.qp.abigail@betelgeuse.wayne.fnx.com.
2. A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. Elsevier, Amsterdam, 1990.
3. A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*, chapter 10.6, pages 395–400. Addison-Wesley, Reading, MA, 1974.
4. J. Albert and L. Wegner. Languages with homomorphic replacements. *Theoretical Computer Science*, 16:291–305, 1981.
5. H. Bordihn, J. Dassow, and M. Holzer. Extending regular expressions with homomorphic replacements. *RAIRO Theoretical Informatics and Applications*, 44(2):229–255, 2010.
6. J. Bremer and D. D. Freydenberger. Inclusion problems for patterns with a bounded number of variables. In *Proc. 14th International Conference on Developments in Language Theory, DLT 2010*, volume 6224 of *LNCS*, pages 100–111, Heidelberg, 2010. Springer.

7. C. Câmpeanu and N. Santean. On the intersection of regex languages with regular languages. *Theoretical Computer Science*, 410(24–25):2336–2344, 2009.
8. C. Câmpeanu and S. Yu. Pattern expressions and pattern automata. *Information Processing Letters*, 92(6):267–274, 2004.
9. C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14:1007–1018, 2003.
10. B. Carle and P. Narendran. On extended regular expressions. In *Proc. Language and Automata Theory and Applications, Third International Conference, LATA 2009*, volume 5457 of *LNCS*, pages 279–289, Heidelberg, 2009. Springer.
11. J. Cassaigne. Unavoidable patterns. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 3, pages 111–134. Cambridge University Press, Cambridge, New York, 2002.
12. J. Currie. Open problems in pattern avoidance. *American Mathematical Monthly*, 100(8):790–793, 1993.
13. N. Cutland. *Computability.* Cambridge University Press, Cambridge, 1980.
14. G. Della Penna, B. Intrigila, E. Tronci, and M. V. Zilli. Synchronized regular expressions. *Acta Informatica*, 39(1):31–70, 2003.
15. D. D. Freydenberger. Extended Regular Expressions: Succinctness and Decidability. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, volume 9 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 507–518, Dagstuhl, 2011. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
16. J. Friedl. *Mastering Regular Expressions.* O'Reilly Media, Sebastopol, CA, 3rd edition, 2006.
17. I. Glaister and J. Shallit. A lower bound technique for the size of nondeterministic finite automata. *Information Processing Letters*, 59(2):75–77, 1996.
18. J. Goldstine, M. Kappes, C. M. R. Kintala, H. Leung, A. Malcher, and D. Wotschke. Descriptional complexity of machines with limited resources. *Journal of Universal Computer Science*, 8(2):193–234, 2002.
19. J. Hartmanis. On Gödel speed-up and succinctness of language representations. *Theoretical Computer Science*, 26(3):335–342, 1983.
20. M. Holzer and M. Kutrib. The complexity of regular(-like) expressions. In *Proc. 14th Conference on Developments in Language Theory, DLT 2010*, volume 6224 of *LNCS*, pages 16–30, Heidelberg, 2010. Springer.
21. M. Holzer and M. Kutrib. Descriptional complexity—an introductory survey. In C. Martín-Vide, editor, *Scientific Applications of Language Methods*, pages 1–58. Imperial College Press, 2010.
22. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.
23. S. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon, J. McCarthy, and W. R. Ashby, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, NJ, 1956.

24. M. Kutrib. The phenomenon of non-recursive trade-offs. *International Journal of Foundations of Computer Science*, 16(5):957–973, 2005.
25. K. Larsen. Regular expressions with nested levels of back referencing form a hierarchy. *Information Processing Letters*, 65(4):169–172, 1998.
26. A. R. Meyer and M. J. Fischer. Economy of description by automata, grammars, and formal systems. In *12th Annual Symposium on Switching and Automata Theory, SWAT (FOCS)*, pages 188–191. IEEE Computer Society, 1971.
27. M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Upper Saddle River, NJ, 1967.
28. P. Odifreddi. *Classical Recursion Theory*, volume I. Elsevier, Amsterdam, 1989.
29. P. Odifreddi. *Classical Recursion Theory*, volume II. Elsevier, Amsterdam, 1999.
30. D. Reidenbach and M. L. Schmid. A polynomial time match test for large classes of extended regular expressions. In *Proc. 15th International Conference on Implementation and Application of Automata, CIAA 2010*, volume 6482 of *LNCS*, pages 241–250, Heidelberg, 2010. Springer.