Extended Regular Expressions: Succinctness and Decidability

Dominik D. Freydenberger

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

Abstract

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

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

1998 ACM Subject Classification F.4.3: Formal Languages

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

Digital Object Identifier 10.4230/LIPIcs.STACS.2011.507

1 Introduction

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

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

This expression can be understood as follows (for a more formal treatment, see Definition 3): 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 $\left((\mathtt{a} \mid \mathtt{b})^*\right) \%x$ 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

28th Symposium on Theoretical Aspects of Computer Science (STACS'11).
Editors: Thomas Schwentick, Christoph Dürr; pp. 507–518

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


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 [6]). Nonetheless, there is a common core to these variants, which was first formalized by Aho [1], and later by Câmpeanu, Salomaa and Yu [5]. Still, theoretical investigation of extended regular expressions has been comparatively rare (in particular when compared to their more prominent subclass); see e. g., Larsen [20], Della Penna et al. [12], Câmpeanu and Santean [6], Carle and Narendran [8], and Reidenbach and Schmid [22].

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

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

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

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. ◇

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

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

[1] One can show this by proving that every NFA for L_n requires at least $O(2^n)$ states, e. g., by using the technique by Glaister and Shallit [14]. Due to the construction used in the proof of Theorem 2.3 in [17], this also gives a lower bound on the length of the regular expressions for L_n.

regular expressions describe regular languages? These natural questions are also motivated by practical concerns: If a given application reuses an expression many times, it might pay off to invest resources in the search for an expression that is shorter, or uses fewer variables, and thus can be matched more efficiently.

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

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

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

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

2 Preliminaries

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

2.1 Basic Definitions

Let $\mathbb{N}$ be the set of natural numbers, including 0. The function div denotes integer division, and mod denotes its remainder (e. g., $5 \operatorname{div} 3 = 1$ and $5 \bmod 3 = 2$). 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, this formalization is due to Aho [1]:

▶ **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 *extended regular expressions* inductively as follows:

1. Each $a \in \Sigma \cup \{\lambda\}$ is an extended regular expression that matches the word a.
2. Each $x \in X$ is an extended regular expression that matches the word to which x is bound.
3. If α_1 and α_2 are extended regular expressions, then $(\alpha_1 \mid \alpha_2)$ is an extended regular expression that matches any word matched by α_1 or by α_2.
4. If α_1 and α_2 are extended regular expressions, then $(\alpha_1\alpha_2)$ is an extended regular expression that matches any word of the form vw, where v matches α_1 and w matches α_2.
5. If α is an extended regular expression, then $(\alpha)^*$ is an extended regular expression that matches any word of the form $w_1 \cdots w_n$ with $n \geq 0$, where α matches each w_i with $1 \leq i \leq n$.
6. If α is an extended regular expression that matches a word w, and $x \in X$, then $(\alpha)\%x$ is an extended regular expression that matches the word w, and x is bound to the value w.

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

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

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

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

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

$$\alpha_1 := \left((\mathtt{a} \mid \mathtt{b})^*\right) \%x\, xx\, \left((\mathtt{a} \mid \mathtt{b})^*\right) \%x\, x, \qquad \alpha_2 := \left(\left((\mathtt{a} \mid \mathtt{b})^*\right) \%x\, x\right)^+.$$

These expressions generate the following languages:

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

Note that both expressions rely on the fact that variables can be bound multiple times, and implicitly assume that we parse from left to right. ◇

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

In general, the membership problem for RegEx is NP-complete, as shown in Theorem 6.2 in Aho [1]. As explained in that proof, this problem is solvable in polynomial-time if the number of different variables is bounded. It is not clear how (or if) Aho's reasoning applies to expressions like α_2 in our Example 4; therefore, we formalize a slightly stronger restriction than Aho, and consider the following subclasses of RegEx:

▶ **Definition 5.** 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 4, 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 $\alpha_1' := \left((\mathtt{a} \mid \mathtt{b})^*\right) \%x\, x x \left((\mathtt{a} \mid \mathtt{b})^*\right) \%y\, y$ generates the same language as α_1, and $\alpha_1' \in (\text{RegEx}(2) \setminus \text{RegEx}(1))$. In contrast to this, $\alpha_2 \notin$ 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 $\alpha \in$ 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 FRegEx(k):= FRegEx $\cap$ 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 $\alpha \in$ 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 9, 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 6.** Let Σ denote a fixed terminal alphabet. For all $k, l \geq 0$, we define the following decision problems for RegEx(k):
Universality Given $\alpha \in$ RegEx(k), is $L(\alpha) = \Sigma^*$?
Cofiniteness Given $\alpha \in$ RegEx(k), is $\Sigma^* \setminus L(\alpha)$ finite?
RegEx(l)-ity Given $\alpha \in$ RegEx(k), is there a $\beta \in$ RegEx(l) with $L(\alpha) = L(\beta)$?

As we shall see, Theorem 9 – one of our main technical results – states that these problems are undecidable (to various degrees). We use the undecidability of the universality problem to show that there is no effective procedure that minimizes extended regular expressions with respect to their length, and the undecidability of RegEx(l)-ity to conclude the same for minimization with respect to the number of variables. Furthermore, cofiniteness and RegEx(l)-ity help us to obtain various results on the relative succinctness of proper and extended regular expressions.

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

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

▶ **Definition 7.** Let $\mathcal{R}$ be a class of extended regular expressions. A *complexity measure* for $\mathcal{R}$ is a total recursive function $c : \mathcal{R} \to \mathbb{N}$ 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 well as most of its natural extensions (for example, in our context, one could define a complexity measure that gives additional weight to the number or distance of occurrences of variables, or their nesting level). Kutrib [19] provides more details on (and an extensive motivation of) complexity measures. Using this definition, we are able to define the notion of tradeoffs between classes of extended regular expressions:

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

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

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

3 Main Results

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

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

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

Of course, all these undecidability results also hold for every RegEx(k) with $k \geq 2$, and for the whole class RegEx of extended regular expressions (as RegEx(1) is contained in all

these classes)[2]. Theorem 9 also demonstrates that inclusion and equivalence are undecidable for RegEx(1) (and, hence, all of RegEx). We also see, as an immediate consequence to Theorem 9, that there is no algorithm that minimizes the number of variables in an extended regular expression, as such an algorithm could be used to decide regularity.

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

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

▶ **Corollary 10.** *Let c be a complexity measure for* RegEx(1)*. Then there is no recursive function m_c that, given an expression $\alpha \in$ 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 $L(\beta) = L(\alpha)$.*

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

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

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

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

▶ **Lemma 12.** *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)*).*

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

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

▶ **Corollary 13.** *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 $L(\beta) = \Sigma^* \setminus L(\alpha)$, $c(\beta) \geq f_c(c(\alpha))$.*

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

In this proof, we concatenate the languages from the proof of Theorem 9 with languages that are RegEx($k+1$)-languages, but not RegEx(k)-languages. 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 11.

4 Proof of Theorem 9

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

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

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

We partition the tape of an extended Turing machine $\mathcal{X}$ into three disjoint areas: The *head symbol*, which is (naturally) the tape symbol at the position of the head, the *right tape side*, which contains the tape word that starts immediately to the right of the head symbol and extends rightward into infinity, and the *left tape side*, which starts immediately left to the head symbol and extends infinitely to the left. When speaking of a configuration, we denote the head symbol by a and refer to the contents of the left or right tape side as the *left*

tape word t_L or the *right tape word* t_R, respectively. 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 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 [21]). As we shall see, our use of extended regular expressions does not allow us to express the fact that every input is ended by exactly one \$ symbol. Without the $\mathrm{CHECK}_{\mathrm{R}}$-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 18 – and thereby most of the main results in Section 3 – crucially depends on the fact that there are extended Turing machines with a finite domain, we use $\mathrm{CHECK}_{\mathrm{R}}$ 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 [21]) and the correspond undecidability results, we establish the following negative results on decision problems for extended Turing machines:

▶ **Lemma 15.** *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.

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

▶ Convention 16. Every extended Turing machine

1. has the state set $Q = \{q_1, \ldots, q_\nu\}$ for some $\nu \geq 1$, where q_1 is the initial state,
2. has $\delta(0, q_1) = (0, L, q_2)$,
3. has $\delta(a, q) = \mathrm{HALT}$ 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 19:

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

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

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

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

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

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

A *valid computation of* $\mathcal{X}$ is a sequence $C = (C_i)_{i=1}^n$ of configurations of $\mathcal{X}$ where C_1 is an initial configuration (i. e. some configuration $(0^\omega, w, 0, q_1)$ 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

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

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

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

The *if* direction is obvious, the *only if* direction follows from Convention 16 and the application of a generalized sequential machine. We are now ready to state the central part of our proof of Theorem 9:

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

In the present paper, we only sketch the proof of Theorem 19. Given an extended Turing machine $\mathcal{X}$, the expression $\alpha_{\mathcal{X}}$ can be assembled from various subexpressions that describe a complete list of sufficient criteria for membership in $\mathrm{INVALC}(\mathcal{X})$. Intuitively, every word

in INVALC($\mathcal{X}$) contains (at least) one error. We first consider so-called *structural errors*, where a word is not an encoding of any sequence $(C_i)_{i=1}^n$ over configurations of $\mathcal{X}$ for some n, or the word is such an encoding, but C_1 is not an initial, or C_n is not a halting configuration. These errors can be described by a proper regular expression, which can be straightforwardly derived from the definition of $\mathcal{X}$.

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

Theorem 9 follows almost immediately from Theorem 19 and Lemmas 15 and 18. Note that the encoding enc and various parts of the proof of Theorem 19 where inspired by the author's proof of a similar but more narrow result on pattern languages (Bremer and Freydenberger [4]).

5 Conclusions

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

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

Acknowledgements

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

References

1 A.V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 5, pages 255–300. Elsevier, 1990.

2 A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, chapter 10.6. Addison-Wesley, Reading, MA, 1974.

3 H. Bordihn, J. Dassow, and M. Holzer. Extending regular expressions with homomorphic replacements. *RAIRO Theoretical Informatics and Applications*, 44(2):229–255, 2010.

4 J. Bremer and D.D. Freydenberger. Inclusion problems for patterns with a bounded number of variables. In *Proc. DLT 2010*, volume 6224 of *LNCS*, pages 100–111, 2010.

5 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.

6 C. Câmpeanu and N. Santean. On the intersection of regex languages with regular languages. *Theoretical Computer Science*, 410(24–25):2336–2344, 2009.

7 C. Câmpeanu and S. Yu. Pattern expressions and pattern automata. *Information Processing Letters*, 92(6):267–274, 2004.

8 B. Carle and P. Narendran. On extended regular expressions. In *Proc. LATA 2009*, volume 5457 of *LNCS*, pages 279–289, 2009.

9 J. Cassaigne. Unavoidable patterns. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 3, pages 111–134. Cambridge University Press, Cambridge, New York, 2002.

10 James Currie. Open problems in pattern avoidance. *American Math. Monthly*, 100(8):790–793, 1993.

11 N. Cutland. *Computability*. Cambridge University Press, 1980.

12 G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Synchronized regular expressions. *Acta Informatica*, 39(1):31–70, 2003.

13 J.E.F. Friedl. *Mastering Regular Expressions*. O'Reilly Media, Sebastopol, CA, 2nd edition, 2002.

14 I. Glaister and J. Shallit. A lower bound technique for the size of nondeterministic finite automata. *Information Processing Letters*, 59(2):75–77, 1996.

15 J. Hartmanis. On Gödel speed-up and succinctness of language representations. *Theoretical Computer Science*, 26(3):335–342, 1983.

16 M. Holzer and M. Kutrib. The complexity of regular(-like) expressions. In *Proc. DLT 2010*, volume 6224 of *LNCS*, pages 16–30, 2010.

17 J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

18 S.C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon; J. McCarthy; W.R. Ashby, editor, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

19 M. Kutrib. The phenomenon of non-recursive trade-offs. *International Journal of Foundations of Computer Science*, 16(5):957–973, 2005.

20 K.S. Larsen. Regular expressions with nested levels of back referencing form a hierarchy. *Information Processing Letters*, 65(4):169–172, 1998.

21 M.L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Upper Saddle River, NJ, 1967.

22 D. Reidenbach and M. Schmid. A polynomial time match test for large classes of extended regular expressions. In *Proc. CIAA 2010*, 2010.