Inside the Class of REGEX Languages

Markus L. Schmid

Department of Computer Science, Loughborough University,
Loughborough, Leicestershire, LE11 3TU, United Kingdom
M.Schmid@lboro.ac.uk

Abstract. We study different possibilities of combining the concept of homomorphic replacement with regular expressions in order to investigate the class of languages given by extended regular expressions with backreferences (REGEX). It is shown in which regard existing and natural ways to do this fail to reach the expressive power of REGEX. Furthermore, the complexity of the membership problem for REGEX with a bounded number of backreferences is considered.

Keywords: Extended Regular Expressions, REGEX, Pattern Languages, Pattern Expressions, Homomorphic Replacement

1 Introduction

Since their introduction by Kleene in 1956 [13], *regular expressions* have not only constantly challenged researchers in formal language theory, they also attracted pioneers of applied computer science as, e. g., Thompson [17], who developed one of the first implementations of regular expressions, marking the beginning of a long and successful tradition of their practical application (see Friedl [10] for an overview). In order to suit practical requirements, regular expressions have undergone various modifications and extensions which lead to so-called *extended regular expressions with backreferences* (*REGEX* for short), nowadays a standard element of most text editors and programming languages (cf. Friedl [10]). The introduction of these new features of extended regular expressions has frequently not been guided by theoretically sound analyses and only recent studies have led to a deeper understanding of their properties (see, e. g., Câmpeanu et al. [5]).

The main difference between REGEX and *classical* regular expressions is the concept of backreferences. Intuitively speaking, a backreference points back to an earlier subexpression, meaning that it has to be matched to the same word the earlier subexpression has been matched to. For example, $r := (_1\ (\mathtt{a} \mid \mathtt{b})^*\ )_1 \cdot \mathtt{c} \cdot \backslash 1$ is a REGEX, where $\backslash 1$ is a *backreference* to the *referenced subexpression* in between the parentheses $(_1$ and $)_1$. The language described by r, denoted by $\mathcal{L}(r)$, is the set of all words $w\mathtt{c}w$, $w \in \{\mathtt{a}, \mathtt{b}\}^*$; a non-regular language. Two aspects of REGEX deserve to be discussed in a bit more detail.

For the REGEX $((_1\ \mathtt{a}^+\ )_1 \mid \mathtt{b}) \cdot \mathtt{c} \cdot \backslash 1$, if we choose the option $\mathtt{b}$ in the alternation, then $\backslash 1$ points to a subexpression that has not been "initialised". Normally, such a backreference is then interpreted as the empty word, which

seems to be the only reasonable way to handle this situation, but, on the other hand, conflicts with the intended semantics of backreferences, particularly in the above example, since it actually means that $\backslash 1$ can be the empty word, whereas the referenced subexpression $(_1\ \mathtt{a}^+\ )_1$ does not match the empty word.

Another particularity appears whenever a backreference points to a subexpression under a star, e. g., $s := ((_1\ \mathtt{a}^*\ )_1 \cdot \mathtt{b} \cdot \backslash 1)^* \cdot \mathtt{c} \cdot \backslash 1$. One might expect s to define the set of all words of form $(\mathtt{a}^n\mathtt{b}\mathtt{a}^n)^m\mathtt{c}\mathtt{a}^n$, $n, m \geq 0$, but s really describes the set $\{\mathtt{a}^{n_1}\mathtt{b}\mathtt{a}^{n_1} \cdot \mathtt{a}^{n_2}\mathtt{b}\mathtt{a}^{n_2} \cdot \cdots \cdot \mathtt{a}^{n_m}\mathtt{b}\mathtt{a}^{n_m} \cdot \mathtt{c} \cdot \mathtt{a}^{n_m} \mid m \geq 1, n_i \geq 0, 1 \leq i \leq m\} \cup \{\mathtt{c}\}$. This is due to the fact that the star operation repeats a subexpression several times without imposing any dependencies between the single iterations. Consequently, in every iteration of the second star in s, the referenced subexpression $(_1\ \mathtt{a}^*\ )_1$ is treated as an individual instance and its scope is restricted to the current iteration. Only the factor that $(_1\ \mathtt{a}^*\ )_1$ matches in the very last iteration is then referenced by any backreference $\backslash 1$ outside the star. A way to see that this behaviour, which is called *late binding* of backreferences, is reasonable, is to observe that if we require $(_1\ \mathtt{a}^*\ )_1$ to take exactly the same value in every iteration of the star, then, for some REGEX r, this may lead to $\mathcal{L}(r^*) \neq (\mathcal{L}(r))^*$.

A suitable language theoretical approach to these backreferences is the concept of *homomorphic replacement*. For example, the REGEX r can also be given as a string $x\mathtt{b}x$, where the symbol x can be homomorphically replaced by words from $\{\mathtt{a}, \mathtt{b}\}^*$, i. e., both occurrences of x must be replaced by the same word. Numerous language generating devices can be found that use various kinds of homomorphic replacement. The most prominent example are probably the well-known L systems (see Kari et al. [12] for a survey), but also many types of grammars as, e. g., Wijngaarden grammars, macro grammars, Indian parallel grammars or deterministic iteration grammars, use homomorphic replacement as a central concept (cf. Albert and Wegner [2] and Bordihn et al. [4] and the references therein). Albert and Wegner [2] and Angluin [3] introduced H-systems and pattern languages, respectively, which both use homomorphic replacement in a more puristic way, without any grammar like mechanisms. More recent models like pattern expressions (Câmpeanu and Yu [7]), synchronized regular expressions (Della Penna et al. [15]) and EH-expressions (Bordihn et al. [4]) are mainly inspired directly by REGEX. While all these models have been introduced and analysed in the context of formal language theory, REGEX have mainly been formed by applications and especially cater for practical requirements. Hence, there is the need in formal language theory to catch up on these practical developments concerning REGEX and we can note that recent work is concerned with exactly that task (see, e. g., [5–9, 14]).

The contribution of this paper is to investigate alternative possibilities to combine the two most elementary components of REGEX, i. e., regular expressions and homomorphic replacement, with the objective of reaching the expressive power of REGEX as close as possible, without exceeding it. Particularly challenging about REGEX is that due to the possible nesting of referenced subexpression the concepts of regular expressions and homomorphic replacement seem to be inherently entangled and there is no easy way to treat them separately. We

illustrate this with the example $t := (_1\ \mathtt{a}^*\ )_1 \cdot (_2\ (\mathtt{b} \cdot \backslash 1)^*\ )_2 \cdot \backslash 2 \cdot \backslash 1$. The language $\mathcal{L}(t) := \{\mathtt{a}^n(\mathtt{ba}^n)^m(\mathtt{ba}^n)^m\mathtt{a}^n \mid n, m \geq 0\}$ cannot that easily be described in terms of a single string with a homomorphic replacement rule, e. g., by the string $xyyx$, where x can be replaced by words from $\{\mathtt{a}^n \mid n \geq 0\}$, and y by words of form $\{(\mathtt{ba}^n)^m \mid n, m \geq 0\}$, since then we can obtain words $\mathtt{a}^n(\mathtt{ba}^{n'})^m(\mathtt{ba}^{n'})^m\mathtt{a}^n$ with $n \neq n'$. In fact, two steps of homomorphic replacement seem necessary, i. e., we first replace y by words from $\{(\mathtt{b}z)^n \mid n \geq 0\}$ and after that we replace x and z by words from $\{\mathtt{a}^n \mid n \geq 0\}$, with the additional requirement that x and z are substituted by the same word. More intuitively speaking, the nesting of referenced subexpressions require *iterated* homomorphic replacement, but we also need to carry on information from one step of replacement to the next one.

The concept of homomorphic replacement is covered best by so-called *pattern languages* as introduced by Angluin [3]. A pattern is a string containing variables and terminal symbols and the corresponding pattern language is the set of all words that can be obtained from the pattern by homomorphically replacing the variables by terminal words. We combine Angluin's patterns with regular expressions by first adding the alternation and star operator to patterns and, furthermore, by letting their variables be typed by regular languages, i. e., the words variables are replaced with are from given regular sets. Then we iterate this step by using this new class of languages again as types for variables and so on. We also take a closer look at *pattern expressions*, which were introduced by Câmpeanu and Yu [7] as a convenient tool to define REGEX languages. In [7], many examples are provided that show how to translate a REGEX into an equivalent pattern expression and vice versa. It is also stated that this is possible in general, but a formal proof for this statement is not provided. In the present work we show that pattern expressions are in fact much weaker than REGEX and they describe a proper subset of the class of REGEX languages (in fact, they are even weaker than REGEX that do not contain referenced subexpressions under a star). These limits in expressive power are caused by the above described difficulties due to the nesting of referenced subexpressions.

On the other hand, pattern expressions still describe an important and natural subclass of REGEX languages, that has been independently defined in terms of other models and, as shown in this work, also coincides with the class of languages resulting from the modification of patterns described above. We then refine the way of how pattern expressions define languages in order to accommodate the nesting of referenced subexpressions and we show that the thus obtained class of languages coincides with the class of languages given by REGEX that do not contain a referenced subexpression under a star.

Finally, we briefly discuss the membership problem for REGEX with a restricted number of backreferences, which, in the unrestricted case, is NP-complete. Although it seems trivial that this problem can be solved in polynomial time, the situation is complicated by subexpressions that occur and are referenced under a star, which represent arbitrarily many distinct subexpressions with individual backreferences.

Note that, due to space constraints, all proofs are omitted.

2 General Definitions

Let $\mathbb{N} := \{1, 2, 3, \ldots\}$ and let $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. For an arbitrary alphabet A, a *word* (*over* A) is a finite sequence of symbols from A, and ε stands for the *empty word*. The notation A^+ denotes the set of all nonempty words over A, and $A^* := A^+ \cup \{\varepsilon\}$. For the *concatenation* of two words w_1, w_2 we write $w_1 \cdot w_2$ or simply $w_1 w_2$. We say that a word $v \in A^*$ is a *factor* of a word $w \in A^*$ if there are $u_1, u_2 \in A^*$ such that $w = u_1 \cdot v \cdot u_2$. The notation $|K|$ stands for the size of a set K or the length of a word K.

We use regular expression as they are commonly defined (see, e. g., Yu [18]). For the alternation operations we use the symbol "|" and in an alternation $(s \mid t)$, we call the subexpressions s and t *options*. For any regular expression r, $\mathcal{L}(r)$ denotes the language described by r and REG denotes the set of regular languages. Let Σ be a finite alphabet of *terminal symbols* and let $X := \{x_1, x_2, x_3, \ldots\}$ be a countably infinite set of *variables* with $\Sigma \cap X = \emptyset$. For any word $w \in (\Sigma \cup X)^*$, $\mathrm{var}(w)$ denotes the set of variables that occur in w.

3 Patterns with Regular Operators and Types

In this section, we combine the pattern languages mentioned in Section 1 with regular languages and regular expressions. more precisely, we first define pattern languages, the variables of which are typed by regular languages and after that we add the regular operators of alternation and star.

Let $\mathrm{PAT} := \{\alpha \mid \alpha \in (\Sigma \cup X)^+\}$ and every $\alpha \in \mathrm{PAT}$ is called a *pattern*. We always assume that, for every $i \in \mathbb{N}$, $x_i \in \mathrm{var}(\alpha)$ implies $\{x_1, x_2, \ldots, x_{i-1}\} \subseteq \mathrm{var}(\alpha)$. For any alphabets A, B, a *morphism* is a function $h : A^* \to B^*$ that satisfies $h(vw) = h(v)h(w)$ for all $v, w \in A^*$. A morphism $h : (\Sigma \cup X)^* \to \Sigma^*$ is called a *substitution* if $h(a) = a$ for every $a \in \Sigma$. For an arbitrary class of languages $\mathfrak{L}$ and a pattern α with $|\mathrm{var}(\alpha)| = m$, an $\mathfrak{L}$*-type for* α is a tuple $\mathcal{T} := (T_{x_1}, T_{x_2}, \ldots, T_{x_m})$, where, for every i, $1 \leq i \leq m$, $T_{x_i} \in \mathfrak{L}$ and T_{x_i} is called the *type language of (variable)* x_i. A substitution h *satisfies* $\mathcal{T}$ if and only if, for every i, $1 \leq i \leq m$, $h(x_i) \in T_{x_i}$.

We recall that in Section 1, the mapping that is done by a substitution has been called a homomorphic replacement. However, here we prefer to use the terminology that is common in the context of Angluin's pattern languages.

Definition 1. *Let* $\alpha \in \mathrm{PAT}$*, let* $\mathfrak{L}$ *be a class of languages and let* $\mathcal{T}$ *be an* $\mathfrak{L}$*-type for* α*. The* $\mathcal{T}$-typed pattern language of α *is defined by* $\mathcal{L}_{\mathcal{T}}(\alpha) := \{h(\alpha) \mid h$ *is a substitution that satisfies* $\mathcal{T}\}$*. For any class of languages* $\mathfrak{L}$*,* $\mathcal{L}_{\mathfrak{L}}(\mathrm{PAT}) := \{\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha \in \mathrm{PAT}, \mathcal{T}$ *is an* $\mathfrak{L}$*-type for* $\alpha\}$ *is the* class of $\mathfrak{L}$-typed pattern languages.

We note that $\{\Sigma^*\}$-typed and $\{\Sigma^+\}$-typed pattern languages correspond to the classes of E-pattern languages and NE-pattern languages, respectively, as defined by Angluin [3] and Shinohara [16]. It is easy to see that $\mathcal{L}_{\mathrm{REG}}(\mathrm{PAT})$ is contained in the class of REGEX languages. The substantial difference between

these two classes is that the backreferences of a REGEX can refer to subexpressions that are not classical regular expressions, but REGEX. Hence, in order to describe larger classes of REGEX languages by means of the pattern-based formalism given in Definition 1, the next step could be to type the variables of patterns with languages from $\mathcal{L}_{\mathrm{REG}}(\mathrm{PAT})$ instead of REG and then using the thus obtained languages again as type languages and so on. However, this approach leads to a dead end:

Proposition 1. *For any class of languages* $\mathfrak{L}$, $\mathcal{L}_{\mathfrak{L}}(\mathrm{PAT}) = \mathcal{L}_{\mathcal{L}_{\mathfrak{L}}(\mathrm{PAT})}(\mathrm{PAT})$.

Proposition 1 demonstrates that typed pattern languages are invariant with respect to iteratively typing the variables of the patterns. This suggests that if we want to extend pattern languages in such a way that they can describe larger subclasses of the class of REGEX languages, then the regular aspect cannot completely be limited to the type languages of the variables. This observation brings us to the definition of $\mathrm{PAT}_{\mathrm{ro}} := \{\alpha \mid \alpha$ is a regular expression over $(\Sigma \cup X'),$ where X' is a finite subset of $X\}$, the set of *patterns with regular operators*. For the sake of convenience, in the remainder of this paper, whenever we use a regular expression over the alphabet $(\Sigma \cup X)$, we actually mean a regular expression over $(\Sigma \cup X')$, for some finite subset X' of X. In order to define the language given by a pattern with regular operators, we extend the definition of types to patterns with regular operators in the obvious way.

Definition 2. *Let* $\alpha \in \mathrm{PAT}_{\mathrm{ro}}$ *and let* $\mathcal{T}$ *be a type for* α. *The* $\mathcal{T}$-typed pattern language of α *is defined by* $\mathcal{L}_{\mathcal{T}}(\alpha) := \bigcup_{\beta \in \mathcal{L}(\alpha)} \mathcal{L}_{\mathcal{T}}(\beta)$. *For any class of languages* $\mathfrak{L}$, *we define* $\mathcal{L}_{\mathfrak{L}}(\mathrm{PAT}_{\mathrm{ro}}) := \{\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha \in \mathrm{PAT}_{\mathrm{ro}}, \mathcal{T}$ *is an* $\mathfrak{L}$*-type for* $\alpha\}$.

Patterns with regular operators are also used in the definition of pattern expressions (see [7] and Section 4) and have been called *regular patterns* in [4]. As an example, we define $\alpha := (x_1 \mathtt{a} x_1 \mid x_2 \mathtt{b} x_2)^* \in \mathrm{PAT}_{\mathrm{ro}}$ and $\mathcal{T} := (\mathcal{L}(\mathtt{c}^*), \mathcal{L}(\mathtt{d}^*))$. The language $\mathcal{L}_{\mathcal{T}}(\alpha)$ can be generated in two steps. We first construct $\mathcal{L}(\alpha) = \{\beta_1 \cdot \beta_2 \cdot \cdots \cdot \beta_n \mid n \in \mathbb{N}_0, \beta_i \in \{x_1 \mathtt{a} x_1, x_2 \mathtt{b} x_2\}, 1 \leq i \leq n\}$ and then $\mathcal{L}_{\mathcal{T}}(\alpha)$ is the union of all typed pattern languages $\mathcal{L}_{\mathcal{T}}(\beta)$, where $\beta \in \mathcal{L}(\alpha)$. Thus, $\mathcal{L}_{\mathcal{T}}(\alpha) = \{w_1 \cdot w_2 \cdot \cdots \cdot w_n \mid n \in \mathbb{N}_0, w_i \in \{\mathtt{c}^m \mathtt{a} \mathtt{c}^m, \mathtt{d}^m \mathtt{b} \mathtt{d}^m \mid m \in \mathbb{N}_0\}, 1 \leq i \leq n\}$.

It seems reasonable to assume that REG-typed patterns with regular operators are strictly more powerful than REG-typed patterns without regular operators. In the following proposition, we formally prove this intuition.

Proposition 2. $\mathcal{L}_{\{\Sigma^*\}}(\mathrm{PAT}) \subset \mathcal{L}_{\mathrm{REG}}(\mathrm{PAT}) \subset \mathcal{L}_{\mathrm{REG}}(\mathrm{PAT}_{\mathrm{ro}})$.

The invariance of typed patterns – represented by Proposition 1 – does not hold anymore with respect to patterns with regular operators. Before we formally prove this claim, we shall define an infinite hierarchy of classes of languages given by typed patterns with regular operators. The bottom of this hierarchy are the REG-typed pattern languages with regular operators. Each level of the hierarchy is then given by patterns with regular operators that are typed by languages from the previous level of the hierarchy and so on.

Definition 3. *Let* $\mathfrak{L}_{\mathrm{ro},0} := \mathrm{REG}$ *and, for every* $i \in \mathbb{N}$*, we define* $\mathfrak{L}_{\mathrm{ro},i} := \mathcal{L}_{\mathfrak{L}_{\mathrm{ro},i-1}}(\mathrm{PAT}_{\mathrm{ro}})$*. Furthermore, we define* $\mathfrak{L}_{\mathrm{ro},\infty} = \bigcup_{i=0}^{\infty} \mathfrak{L}_{\mathrm{ro},i}$*.*

It follows by definition, that the classes $\mathfrak{L}_{\mathrm{ro},i}$, $i \in \mathbb{N}_0$, form a hierarchy and we strongly conjecture that it is proper. However, here we only separate the first three levels of that hierarchy.

Theorem 1. $\mathfrak{L}_{\mathrm{ro},0} \subset \mathfrak{L}_{\mathrm{ro},1} \subset \mathfrak{L}_{\mathrm{ro},2} \subseteq \mathfrak{L}_{\mathrm{ro},3} \subseteq \mathfrak{L}_{\mathrm{ro},4} \subseteq \ldots$.

In the following section, we take a closer look at the class $\mathfrak{L}_{\mathrm{ro},\infty}$. We shall show that it coincides with the class of languages that are defined by the already mentioned pattern expressions and we formally prove it to be a proper subset of the class of REGEX languages.

4 Pattern Expressions

We define pattern expressions as introduced by Câmpeanu and Yu [7], but we use a slightly different notation.

Definition 4. *A* pattern expression *is a tuple* $(x_1 \to r_1, x_2 \to r_2, \ldots, x_n \to r_n)$*, where, for every* i*,* $1 \leq i \leq n$*,* $r_i \in \mathrm{PAT}_{\mathrm{ro}}$ *and* $\mathrm{var}(r_i) \subseteq \{x_1, x_2, \ldots, x_{i-1}\}$*. The set of all pattern expressions is denoted by* PE*.*

In [7], the language of a pattern expression $p := (x_1 \to r_1, x_2 \to r_2, \ldots, x_n \to r_n)$ is defined in the following way. Since, by definition, r_1 is a classical regular expression, it describes a regular language L. The language L is then interpreted as a type for variable x_1 in every r_i, $2 \leq i \leq n$. This step is then repeated, i. e., $\mathcal{L}_{(L)}(r_2)$ is the type for x_2 in every r_j, $3 \leq j \leq n$, and so on.

Definition 5. *Let* $p := (x_1 \to r_1, x_2 \to r_2, \ldots, x_n \to r_n)$ *be a pattern expression. We define* $L_{p,x_1} := \mathcal{L}(r_1)$ *and, for every* i*,* $2 \leq i \leq n$*,* $L_{p,x_i} := \mathcal{L}_{\mathcal{T}_i}(r_i)$*, where* $\mathcal{T}_i := (L_{p,x_1}, L_{p,x_2}, \ldots, L_{p,x_{i-1}})$ *is a type for* r_i*. The* language generated by p with respect to iterated substitution *is defined by* $\mathcal{L}_{\mathrm{it}}(p) := L_{p,x_n}$ *and* $\mathcal{L}_{\mathrm{it}}(\mathrm{PE}) := \{\mathcal{L}_{\mathrm{it}}(p) \mid p \in \mathrm{PE}\}$*.*

We illustrate the above definition with an example. Let

$$q := (x_1 \to \mathtt{a}^*, x_2 \to x_1(\mathtt{c} \mid \mathtt{d})x_1, x_3 \to x_1\mathtt{c}x_2)$$

be a pattern expression. According to the above definition, $\mathcal{L}_{\mathrm{it}}(q) = \{\mathtt{a}^k\mathtt{c}\mathtt{a}^m u \mathtt{a}^m \mid k, m \in \mathbb{N}_0, u \in \{\mathtt{c}, \mathtt{d}\}\}$. We note that in a word $\mathtt{a}^k\mathtt{c}\mathtt{a}^m u \mathtt{a}^m \in \mathcal{L}_{\mathrm{it}}(q)$, both $\mathtt{a}^k$ and $\mathtt{a}^m$ are substitution words for the same variable x_1 from the type language L_{q,x_1}. However, $k \neq m$ is possible, since, intuitively speaking, $\mathtt{a}^k$ is picked first from L_{q,x_1} as the substitution word for x_1 in $x_1\mathtt{c}x_2$ and then $\mathtt{a}^m$ is picked from L_{q,x_1} as substitution word for x_1 in $x_1(\mathtt{c} \mid \mathtt{d})x_1$ in order to construct the substitution word $\mathtt{a}^m u \mathtt{a}^m$ for x_2 in $x_1\mathtt{c}x_2$. Consequently, occurrences of the same variable in different elements of the pattern expression do not need to be substituted

by the same word. We shall later see that this behaviour essentially limits the expressive power of pattern expressions.

As mentioned before, the class of languages described by pattern expressions with respect to iterated substitution coincides with the class $\mathfrak{L}_{\mathrm{ro},\infty}$ of the previous section.

Theorem 2. $\mathfrak{L}_{\mathrm{ro},\infty} = \mathcal{L}_{\mathrm{it}}(\mathrm{PE})$.

In the following, we define an alternative way of how pattern expressions can describe languages, i. e., instead of substituting the variables by words in an iterative way, we substitute them uniformly.

Definition 6. *Let* $p := (x_1 \to r_1, x_2 \to r_2, \ldots, x_n \to r_n) \in \mathrm{PE}$. *A word* $w \in \Sigma^*$ *is in the* language generated by p with respect to uniform substitution *(*$\mathcal{L}_{\mathrm{uni}}(p)$*, for short) if and only if there exists a substitution* h *such that* $h(x_n) = w$ *and, for every* i, $1 \leq i \leq n$, *there exists an* $\alpha_i \in \mathcal{L}(r_i)$ *with* $h(x_i) = h(\alpha_i)$.

For the pattern expression q from above, a word w is in $\mathcal{L}_{\mathrm{uni}}(q)$ if there is a substitution h with $h(x_3) = w$ and there exist $\alpha_1 \in \mathcal{L}(\mathtt{a}^*)$, $\alpha_2 \in \mathcal{L}(x_1(\mathtt{c} \mid \mathtt{d})x_1)$ and $\alpha_3 \in \mathcal{L}(x_1\mathtt{c}x_2)$, such that $h(x_1) = h(\alpha_1)$, $h(x_2) = h(\alpha_2)$ and $h(x_3) = h(\alpha_3)$. Since $\alpha_1 = \mathtt{a}^n$, $n \in \mathbb{N}_0$, $\alpha_2 = x_1 u x_1$, $u \in \{\mathtt{c}, \mathtt{d}\}$, and $\alpha_3 = x_1\mathtt{c}x_2$, this implies that w is in $\mathcal{L}_{\mathrm{uni}}(q)$ if there is a substitution h and an $\alpha := x_1\mathtt{c}x_1 u x_1$, $u \in \{\mathtt{c}, \mathtt{d}\}$, such that $w = h(\alpha)$ and h satisfies the type $(\mathcal{L}(\mathtt{a}^*))$. Thus, $\mathcal{L}_{\mathrm{uni}}(q) = \{\mathtt{a}^n\mathtt{c}\mathtt{a}^n u \mathtt{a}^n \mid n \in \mathbb{N}_0, u \in \{\mathtt{c}, \mathtt{d}\}\}$, which is a proper subset of $\mathcal{L}_{\mathrm{it}}(q)$.

For an arbitrary pattern expression $p := (x_1 \to r_1, x_2 \to r_2, \ldots, x_n \to r_n)$, the language $\mathcal{L}_{\mathrm{uni}}(p)$ can also be defined in a more constructive way. We first choose a word $u \in \mathcal{L}(r_1)$ and, for all i, $1 \leq i \leq n$, if variable x_1 occurs in r_i, then we substitute all occurrences of x_1 in r_i by u. Then we delete the element $x_1 \to r_1$ from the pattern expression. If we repeat this step with respect to variables $x_2, x_3, \ldots, x_{n-1}$, then we obtain a pattern expression of form $(x_n \to r'_n)$, where r'_n is a regular expression over Σ. The language $\mathcal{L}_{\mathrm{uni}}(p)$ is the union of the languages given by all these regular expression.

The language $\mathcal{L}_{\mathrm{it}}(q)$ can be defined similarly. We first choose a word $u_1 \in \mathcal{L}(r_1)$ and then we substitute all occurrences of x_1 in r_2 by u_1. After that, we choose a *new* word $u_2 \in \mathcal{L}(r_1)$ and substitute all occurrences of x_1 in r_3 by u_2 and so on until there are no more occurrences of variable x_1 in q and then we delete the element $x_1 \to r_1$. Then this step is repeated with respect to $x_2, x_3, \ldots, x_{n-1}$.

The above considerations yield the following proposition:

Proposition 3. *Let* $p := (x_1 \to r_1, x_2 \to r_2, \ldots, x_m \to r_m)$ *be a pattern expression. Then* $\mathcal{L}_{\mathrm{uni}}(p) \subseteq \mathcal{L}_{\mathrm{it}}(p)$ *and if, for every* i, j, $1 \leq i < j \leq m$, $\mathrm{var}(r_i) \cap \mathrm{var}(r_j) = \emptyset$, *then also* $\mathcal{L}_{\mathrm{it}}(p) \subseteq \mathcal{L}_{\mathrm{uni}}(p)$.

The interesting question is whether or not there exists a language $L \in \mathcal{L}_{\mathrm{uni}}(\mathrm{PE})$ with $L \notin \mathcal{L}_{\mathrm{it}}(\mathrm{PE})$ or vice versa. Intuitively, for any pattern expression p, it seems obvious that it is not essential for the language $\mathcal{L}_{\mathrm{it}}(p)$ that there exist occurrences of the same variable in different elements of p and it

should be possible to transform p into an equivalent pattern expression p', the elements of which have disjoint sets of variables and, thus, by Proposition 3, $\mathcal{L}_{\text{it}}(p) = \mathcal{L}_{\text{uni}}(p')$. Hence, for the language generated by a pattern expression with respect to iterated substitution, the possibility of using the same variables in different elements of a pattern expression can be considered as mere syntactic sugar that keeps pattern expressions concise. On the other hand, the question of whether or not, for every pattern expression p, we can find a pattern expression p' with $\mathcal{L}_{\text{uni}}(p) = \mathcal{L}_{\text{it}}(p')$, is not that easy to answer. The following lemma states that there are in fact languages that can be expressed by some pattern expression with respect to uniform substitution, but not by any pattern expression with respect to iterated substitution.

Lemma 1. *There exists a language* $L \in \mathcal{L}_{\text{uni}}(\text{PE})$ *with* $L \notin \mathcal{L}_{\text{it}}(\text{PE})$.

From Lemma 1 we can conclude the main result of this section, i. e., the class of languages given by pattern expressions with respect to iterated substitution is a proper subset of the class of languages given by pattern expressions with respect to uniform substitution.

Theorem 3. $\mathcal{L}_{\text{it}}(\text{PE}) \subset \mathcal{L}_{\text{uni}}(\text{PE})$.

We conclude this section by mentioning that in Bordihn et al. [4], it has been shown that $\mathcal{H}^*(\text{REG}, \text{REG})$, a class of languages given by an iterated version of H-systems (see Albert and Wegner [2] and Bordihn et al. [4]), also coincides with $\mathcal{L}_{\text{it}}(\text{PE})$, which implies $\mathfrak{L}_{\text{ro},\infty} = \mathcal{L}_{\text{it}}(\text{PE}) = \mathcal{H}^*(\text{REG}, \text{REG}) \subset \mathcal{L}_{\text{uni}}(\text{PE})$.

In the following section, we take a closer look at the larger class $\mathcal{L}_{\text{uni}}(\text{PE})$ and compare it to the class of REGEX languages.

5 REGEX

We use a slightly different notation for REGEX compared to the one used in [5].

A REGEX is a regular expression, the subexpressions of which can be numbered by adding an integer index to the parentheses delimiting the subexpression (i. e., $(_n \ \ldots \ )_n$, $n \in \mathbb{N}$). This is done in such a way that there are no two different subexpressions with the same number. The subexpression that is numbered by $n \in \mathbb{N}$, which is called the n^{th} *referenced subexpression*, can be followed by arbitrarily many *backreferences* to that subexpression, denoted by $\backslash n$.

For example, $(_1 \ \mathtt{a} \mid \mathtt{b} \ )_1 \cdot (_2 \ (\mathtt{c} \mid \mathtt{a})^* \ )_2 \cdot (\backslash 1)^* \cdot \backslash 2$ is a REGEX, whereas $r_1 := (_1 \ \mathtt{a} \mid \mathtt{b} \ )_1 \cdot (_1 \ (\mathtt{c} \mid \mathtt{a})^* \ )_1 \cdot (\backslash 1)^* \cdot \backslash 2$ and $r_2 := (_1 \ \mathtt{a} \mid \mathtt{b} \ )_1 \cdot \backslash 2 \cdot (_2 \ (\mathtt{c} \mid \mathtt{a})^* \ )_2 \cdot (\backslash 1)^* \cdot \backslash 2$ is not a REGEX, since in r_1 there are two different subexpressions numbered by 1 and in r_2 there is an occurrence of a backreference $\backslash 2$ before the second referenced subexpression.

A formal definition of the language described by a REGEX can be found in [5]. Here, we stick to the more informal definition which has already been briefly outlined in Section 1 and that we now recall in a bit more detail.

For a REGEX r, the language described by r is denoted by $\mathcal{L}(r)$. A word w is in $\mathcal{L}(r)$ if and only if we can obtain it from r in the following way. We

move over r from left two right. We treat alternations and stars as it is done for classical regular expressions and we note down every terminal symbol that we read. When we encounter the i^{th} referenced subexpression, then we store the factor u_i that is matched to it and from now on we treat every occurrence of $\backslash i$ as u_i. However, there are two special cases we need to take care of. Firstly, when we encounter the i^{th} referenced subexpression for a second time, which is possible since the i^{th} referenced subexpression may occur under a star, then we overwrite u_i with the possible new factor that is now matched to the i^{th} referenced subexpression. This entails the late binding of backreferences, which has been described in Section 1. Secondly, if a backreference $\backslash i$ occurs and there is no factor u_i stored that has been matched to the i^{th} referenced subexpression, then $\backslash i$ is interpreted as the empty word.

We also define an alternative way of how a REGEX describes a language, that shall be useful for our proofs. The *language with necessarily initialised subexpressions* of a REGEX r, denoted by $\mathcal{L}_{\text{nis}}(r)$, is defined in a similar way as $\mathcal{L}(r)$ above, but if a backreference $\backslash i$ occurs and there is currently no factor u_i stored that has been matched to the i^{th} referenced subexpression, then instead of treating $\backslash i$ as the empty word, we interpret it as the i^{th} referenced subexpression, we store the factor u_i that is matched to it and from now on every occurrence of $\backslash i$ is treated as u_i. For example, let $r := ((_1\ \mathtt{a}^*\ )_1 \mid \varepsilon) \cdot \mathtt{b} \cdot \backslash 1 \cdot \mathtt{b} \cdot \backslash 1$. Then $\mathcal{L}(r) := \{\mathtt{a}^n\mathtt{b}\mathtt{a}^n\mathtt{b}\mathtt{a}^n \mid n \in \mathbb{N}_0\}$ and $\mathcal{L}_{\text{nis}}(r) := \mathcal{L}(r) \cup \{\mathtt{b}\mathtt{a}^n\mathtt{b}\mathtt{a}^n \mid n \in \mathbb{N}_0\}$.

We can note that the late binding of backreferences as well as non-initialised referenced subexpressions is caused by referenced subexpression under a star or in an alternation. Next, we define REGEX that are restricted in this regard.

Definition 7. *A* REGEX r *is* alternation confined *if and only if the existence of a referenced subexpression in the option of an alternation implies that all the corresponding backreferences occur in the same option of the same alternation. A* REGEX r *is* star-free initialised *if and only if every referenced subexpression does not occur under a star. Let* REGEX_{ac} *and* $\text{REGEX}_{\text{sfi}}$ *be the sets of* REGEX *that are alternation confined and star-free initialised, respectively. Furthermore, let* $\text{REGEX}_{\text{sfi,ac}} := \text{REGEX}_{\text{ac}} \cap \text{REGEX}_{\text{sfi}}$.

We can show that the condition of being alternation confined does not impose a restriction on the expressive power of a star-free initialised REGEX. The same holds with respect to their languages with necessarily initialised subexpressions. Furthermore, for every star-free initialised REGEX r, the language $\mathcal{L}(r)$ can also be given as the language with necessarily initialised subexpressions of a star-free initialised REGEX and vice versa. This is formally stated in the next lemma, which shall be useful for proving the main result of this section.

Lemma 2.

$$\mathcal{L}(\text{REGEX}_{\text{sfi}}) = \mathcal{L}(\text{REGEX}_{\text{sfi,ac}}) = \mathcal{L}_{\text{nis}}(\text{REGEX}_{\text{sfi}}) = \mathcal{L}_{\text{nis}}(\text{REGEX}_{\text{sfi,ac}})\,.$$

In the following, we take a closer look at the task of transforming a pattern expression p into a REGEX r, such that $\mathcal{L}_{\text{uni}}(p) = \mathcal{L}(r)$. Although, this is

possible in general, a few difficulties arise, that have already been pointed out by Câmpeanu and Yu in [7] (with respect to $\mathcal{L}_{\mathrm{it}}(p)$).

The natural way to transform a pattern expression into an equivalent REGEX is to successively substitute the occurrences of variables by referenced subexpressions and appropriate backreferences. However, this is not always possible. For example, consider the pattern expression $q := (x_1 \to (\mathtt{a} \mid \mathtt{b})^*, x_2 \to x_1^* \cdot \mathtt{c} \cdot x_1 \cdot \mathtt{d} \cdot x_1)$. If we simply transform q into $r_q := (_1\ (\mathtt{a} \mid \mathtt{b})^*\ )_1^* \cdot \mathtt{c} \cdot \backslash 1 \cdot \mathtt{d} \cdot \backslash 1$, then we obtain an incorrect REGEX, since $\mathcal{L}_{\mathrm{uni}}(q) \neq \mathcal{L}(r_q)$. This is due to the fact that the referenced subexpression is under a star. To avoid this, we can first rewrite q to $q' := (x_1 \to (\mathtt{a} \mid \mathtt{b})^*, x_2 \to (x_1 \cdot x_1^* \mid \varepsilon) \cdot \mathtt{c} \cdot x_1 \cdot \mathtt{d} \cdot x_1)$, which leads to $r_{q'} := ((_1\ (\mathtt{a} \mid \mathtt{b})^*\ )_1 \cdot (\backslash 1)^* \mid \varepsilon) \cdot \mathtt{c} \cdot \backslash 1 \cdot \mathtt{d} \cdot \backslash 1$. Now we encounter a different problem: $\mathcal{L}_{\mathrm{uni}}(q')$ contains the word `cabadaba`, but in $\mathcal{L}(r_{q'})$ the only word that starts with `c` is `cd`. This is due to the fact that if we choose the second option of $((_1\ (\mathtt{a} \mid \mathtt{b})^*\ )_1 \cdot (\backslash 1)^* \mid \varepsilon)$, then all $\backslash 1$ are set to the empty word. However, we note that the language with necessarily initialised subexpressions of $r_{q'}$ is exactly what we want, since $\mathcal{L}_{\mathrm{nis}}(r_{q'}) = \mathcal{L}_{\mathrm{uni}}(q)$. Hence, we can transform any pattern expression p to a REGEX r_p that is star-free initialised and $\mathcal{L}_{\mathrm{uni}}(p) = \mathcal{L}_{\mathrm{nis}}(r_p)$.

Lemma 3. *For every pattern expression p, there exists a star-free initialised* REGEX r *with* $\mathcal{L}_{\mathrm{uni}}(p) = \mathcal{L}_{\mathrm{nis}}(r)$.

We recall that Lemma 2 states that every star-free initialised REGEX r can be transformed into a star-free initialised REGEX r' with $\mathcal{L}_{\mathrm{nis}}(r) = \mathcal{L}(r')$. Consequently, Lemmas 2 and 3 imply that every pattern expression p can be transformed into a star-free initialised REGEX r with $\mathcal{L}_{\mathrm{uni}}(p) = \mathcal{L}(r)$. For example, the pattern expression q introduced on page 10 can be transformed into the REGEX $t_q := ((_1\ (\mathtt{a} \mid \mathtt{b})^*\ )_1 \cdot (\backslash 1)^* \cdot \mathtt{c} \cdot \backslash 1 \cdot \mathtt{d} \cdot \backslash 1 \mid \mathtt{c} \cdot (_2\ (\mathtt{a} \mid \mathtt{b})^*\ )_2 \cdot \mathtt{d} \cdot \backslash 2)$, which finally satisfies $\mathcal{L}_{\mathrm{uni}}(q) = \mathcal{L}(t_q)$.

Theorem 4. $\mathcal{L}_{\mathrm{uni}}(\mathrm{PE}) \subseteq \mathcal{L}(\mathrm{REGEX}_{\mathrm{sfi}})$.

In the remainder of this section, we show the converse of Theorem 4, i. e., every star-free initialised REGEX r can be transformed into a pattern expression that describes the language $\mathcal{L}(r)$ with respect to uniform substitution. However, this cannot be done directly if r is not alternation confined. As an example, we consider $r := ((_1\ (\mathtt{a} \mid \mathtt{b})^*\ )_1 \mid (_2\ \mathtt{c}^*\ )_2) \cdot (\backslash 1)^* \cdot \backslash 2$. Now the natural way to transform r into a pattern expression is to substitute the first and second referenced subexpression and the corresponding backreferences by variables x_1 and x_2, respectively, and to introduce elements $x_1 \to (\mathtt{a} \mid \mathtt{b})$ and $x_2 \to \mathtt{c}^*$, i. e., $p_r := (x_1 \to (\mathtt{a} \mid \mathtt{b}), x_2 \to \mathtt{c}^*, x_3 \to (x_1 \mid x_2) \cdot (x_1)^* \cdot x_2)$. Now $\mathcal{L}_{\mathrm{uni}}(p_r)$ contains the word `cccababababccc`, whereas every word in $\mathcal{L}(r)$ that starts with `c` does not contain any occurrence of `a` or `b`, thus, $\mathcal{L}_{\mathrm{uni}}(p_r) \neq \mathcal{L}(r)$. So in order to transform star-free initialised REGEX into equivalent pattern expressions, again Lemma 2 is very helpful, which states that we can transform every star-free initialised REGEX into an equivalent one that is also alternation confined.

Theorem 5. $\mathcal{L}(\mathrm{REGEX}_{\mathrm{sfi}}) \subseteq \mathcal{L}_{\mathrm{uni}}(\mathrm{PE})$.

From Theorems 4 and 5, we can conclude that the class of languages described by pattern expressions with respect to uniform substitution coincides with the class of languages given by regular expressions that are star-free initialised.

Corollary 1. $\mathcal{L}(\mathrm{REGEX}_{\mathrm{sfi}}) = \mathcal{L}_{\mathrm{uni}}(\mathrm{PE})$.

In Sections 3 and 4 and in the present section, we have investigated several proper subclasses of the class of REGEX languages and their mutual relations. We conclude this section, by summarising these results:

$$\mathcal{L}_{\{\Sigma^*\}}(\mathrm{PAT}) \subset \mathcal{L}_{\mathrm{REG}}(\mathrm{PAT}) \subset \mathfrak{L}_{\mathrm{ro},1} \subset \mathfrak{L}_{\mathrm{ro},2} \subseteq \mathfrak{L}_{\mathrm{ro},3} \subseteq \ldots \subseteq \mathfrak{L}_{\mathrm{ro},\infty} = \mathcal{H}^*(\mathrm{REG}, \mathrm{REG}) = \mathcal{L}_{\mathrm{it}}(\mathrm{PE}) \subset \mathcal{L}_{\mathrm{uni}}(\mathrm{PE}) = \mathcal{L}(\mathrm{REGEX}_{\mathrm{sfi}}) \subseteq \mathcal{L}(\mathrm{REGEX}) .$$

6 REGEX with a Bounded Number of Backreferences

It is a well known fact that the membership problem for REGEX languages is NP-complete (cf. Aho [1] and Angluin [3]). Furthermore, Aho states that it can be solved in time that is exponential only in the number of backreferences in the following way. Let k be the number of referenced subexpressions in a REGEX r and let w be an input word. We first choose k factors $u_1, u_2, \ldots, u_k$ of w and then try to match r to w in such a way that, for every i, $1 \leq i \leq k$, the i^{th} referenced subexpression is matched to u_i. This is done with respect to all possible k factors of w. For this procedure we only need to keep track of the k possible factors of w, thus, time $\mathrm{O}(|w|^{2k})$ is sufficient. However, this approach is incorrect, since it ignores the possibility that the referenced subexpressions under a star (and their backreferences) can be matched to a different factor in every individual iteration of the star. On the other hand, if we first iterate every expression under a star that contains a referenced subexpression an arbitrary number of times, then, due to the late binding of backreferences, we introduce arbitrarily many new referenced subexpressions and backreferences, so there is an arbitrary number of factors to keep track of.

The question whether the membership problem for REGEX can be solved in time that is exponential only in the number of backreferences is not a secondary one, since a positive answer yields the polynomial time solvability of the membership problem for languages given by REGEX with a bounded number of backreferences.

We give a positive answer to that question, by showing that for any REGEX r, a nondeterministic two-way multi-head automaton (see Holzer et al. [11] for a survey) can be constructed that accepts exactly $\mathcal{L}(r)$ with a number of input heads that is bounded by the number of referenced subexpressions in r and a number of states that is bounded by the length of r.

Lemma 4. *Let r be a* REGEX *with k referenced subexpressions. There exists a nondeterministic two-way $(3k+2)$-head automaton with $\mathrm{O}(|r|)$ states that accepts $\mathcal{L}(r)$.*

Since we can solve the acceptance problem of a given two-way multi-head automaton M and a given word w in time that is exponential only in the number of input heads, we can conclude the following result:

Theorem 6. *Let $k \in \mathbb{N}$. The membership problem for* REGEX *with at most k referenced subexpressions can be solved in polynomial time.*

References

1. A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 255–300. MIT Press, 1990.
2. J. Albert and L. Wegner. Languages with homomorphic replacements. *Theoretical Computer Science*, 16:291–305, 1981.
3. D. Angluin. Finding patterns common to a set of strings. In *Proc. 11th Annual ACM Symposium on Theory of Computing*, pages 130–141, 1979.
4. H. Bordihn, J. Dassow, and M. Holzer. Extending regular expressions with homomorphic replacement. *RAIRO Theoretical Informatics and Applications*, 44:229–255, 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*, 410:2336–2344, 2009.
7. C. Câmpeanu and S. Yu. Pattern expressions and pattern automata. *Information Processing Letters*, 92:267–274, 2004.
8. B. Carle and P. Narendran. On extended regular expressions. In *Proc. LATA 2009*, volume 5457 of *LNCS*, pages 279–289, 2009.
9. D. D. Freydenberger. Extended regular expressions: Succinctness and decidability. In *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011*, volume 9 of *LIPIcs*, pages 507–518, 2011.
10. J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, Sebastopol, CA, third edition, 2006.
11. M. Holzer, M. Kutrib, and A. Malcher. Complexity of multi-head finite automata: Origins and directions. *Theoretical Computer Science*, 412:83–96, 2011.
12. L. Kari, G. Rozenberg, and A. Salomaa. L systems. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 5, pages 253–328. Springer, 1997.
13. S.C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 3–41. Princeton University Press, 1956.
14. K. S. Larsen. Regular expressions with nested levels of back referencing form a hierarchy. *Information Processing Letters*, 65:169–172, 1998.
15. G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Synchronized regular expressions. *Acta Informatica*, 39:31–70, 2003.
16. T. Shinohara. Polynomial time inference of extended regular pattern languages. In *Proc. RIMS Symposia, Kyoto*, volume 147 of *LNCS*, pages 115–127, 1982.
17. K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11, 1968.
18. S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 2, pages 41–110. Springer, 1997.