Theoretical Computer Science 410 (2009) 2336–2344

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On the intersection of regex languages with regular languages

Cezar Câmpeanu [a,*], Nicolae Santean [b]

[a] Department of Computer Science and Information Technology, University of Prince Edward Island, Canada
[b] Department of Computer and Information Sciences, Indiana University South Bend, IN, USA

ARTICLE INFO

Keywords:
Extended regular expression
Regex automata system
Regex

ABSTRACT

In this paper we revisit the semantics of extended regular expressions (regex), defined succinctly in the 90s [A.V. Aho, Algorithms for finding patterns in strings, in: Jan van Leeuwen (Ed.), Handbook of Theoretical Computer Science, in: Algorithms and Complexity, vol. A, Elsevier and MIT Press, 1990, pp. 255–300] and rigorously in 2003 by Câmpeanu, Salomaa and Yu [C. Câmpeanu, K. Salomaa, S. Yu, A formal study of practical regular expressions, IJFCS 14 (6) (2003) 1007–1018], when the authors reported an open problem, namely whether regex languages are closed under the intersection with regular languages. We give a positive answer; and for doing so, we propose a new class of machines — regex automata systems (RAS) — which are equivalent to regex. Among others, these machines provide a consistent and convenient method of implementing regex in practice. We also prove, as a consequence of this closure property, that several languages, such as the mirror language, the language of palindromes, and the language of balanced words are not regex languages.

1. Introduction

Regular expressions are powerful programming tools present in many scripting languages such as Perl, Awk, PHP, and Python, as well as in most programming languages implemented after year 2000. Despite a similar nomenclature, these practical regular expressions (called regex in our paper) are more powerful than the regular expressions defined in formal language theory, mainly due to the presence of the back-reference operator. This operation allows us to express patterns (repetitions) in words, therefore regex can specify languages beyond the regular family. For example, the regex $(a^*b)\backslash 1$ expresses all the double words starting with arbitrary many a's followed by a b: the operator "$\backslash 1$" is a reference to (copy of) the content of the first pair of parentheses.

The current implementations of extended regular expressions are plagued by many conceptual problems, which can readily be demonstrated on many systems. For example, the use of Perl[1] regex $((a)|(b)) * \backslash 2$ or $((a)|(b)) * \backslash 2\backslash 3$ leads to an erratic behavior due to its inherent semantic ambiguity. Furthermore, in Perl, the expression $()$ is considered to match the empty word, whereas it should arguably match the empty set; thus, there is no semantic difference between the Perl expressions $()$ and $()^*$. Moreover, in theory, a back-reference should replicate the last match of its corresponding parenthesis if such a match has occurred, or the $\emptyset$ otherwise. In the following Perl example this is not the case: $((a|b)|(b|a)) * c\backslash 2\backslash 3$ matches *babbbcbb*, but not *babbcab*, however, $((a|b)|(b|a)*) * c\backslash 2\backslash 3$ matches both in some implementations.[2] Here the behavior suggests that the second parenthesis matches always ε and never b's. Tested on *babbcba* and *abcba*, we discover that these words are matched, suggesting that non-determinism in these regex implementations is selective. Thus, we observe implementation inconsistencies, ambiguities and a lack of standard semantics. This unfortunate status quo of having flawed

* Corresponding author.
E-mail address: ccampeanu@upei.ca (C. Câmpeanu).

[1] Tested on more than ten different implementations of Perl 5.x on Solaris and Linux systems.

[2] Newer versions of Perl seem to have fewer such pathological cases, however, we found other cases of queer behavior that were not present in the previous versions.

regex implementations, as well as an incomplete theoretical foundation, has recently lead to an increased research effort aiming at their better understanding. Some of the problems of regex semantics have been addressed recently in the work of Câmpeanu, Kai Salomaa, and Sheng Yu, who have initiated a rigorous formalism for regex in [2]. In addition, Câmpeanu and Sheng Yu provide an alternative to this formalism, by introducing pattern expressions in [4].

The present paper continues their line of research, focusing on two matters: to deal with some pathological aspects of regex semantics and, most importantly, to answer an open problem stated in [2, Conclusion], namely whether regex languages are closed under the intersection with regular languages.

2. Definitions and notation

Let Σ be an alphabet, that is, a finite set of symbols (or letters). By Σ^* we denote all words (strings of symbols) over Σ, and by ε we denote the empty word, i.e., the word with no letters. If $w \in \Sigma^*$, we denote by $|w|_a$ the number of occurrences of symbol a in w, and by $|w|$ the length of w (the total number of letters in w). A language L is a subset of Σ^*. The cardinality of a set X is denoted by $\#(X)$. For other notions we refer the reader to [7–10].

An extended regular expression, or *regex* for brevity, is a regular expression with back-references [6]. This extension can be found in most programming languages and has been conceptualized in several studies, such as [1,2,4]. We give here a definition equivalent to the one found in [1, C. 5, Section 2.3, p. 261].

Definition 1. A regex over Σ is a well-formed parenthesized formula, consisting of operands in $\Sigma^* \cup \{\backslash i | i \geq 1\}$, the binary operators $\cdot$ and $+$, and the unary operator * (Kleene star). By convention, $()$ and any other form of "empty" expression is a regex denoting $\emptyset$ (consequently, $()^*$ will denote ε). Besides the common rules governing regular expressions, a regex obeys the following syntactic rule: every control character $\backslash i$ is found to the right of the ith pair of parentheses, where parentheses are indexed according to the occurrence sequence of their left parenthesis.

The language represented by a regex r is that of all words matching r in the sense of regular expression matching, with the additional semantic rules:

(1) During the matching of a word with a regex r, a control $\backslash i$ should match a sub-word that has matched the parenthesis i in r. There is one exception to this rule:
(2) If the ith pair of parentheses is under a Kleene star and '$\backslash i$' is not under the same Kleene star, then '$\backslash i$' matches the content of the pair of parentheses under the Kleene star, as given by its last iteration.

Example 2. The expression $r = (a^*)b\backslash 1$ defines the language $\{a^n b a^n \mid n \geq 0\}$. For the expression $r = (a^*b)^*\backslash 1$, $aabaaabaaab \in L(r)$ and $aabaaabaab \notin L(r)$.

Remark 3. Most programming languages use | instead of $+$ to avoid ambiguity between the $+$ sign and the $^+$ superscript. Since there is no danger of confusion, in this paper we will use $+$ for alternation.

There is a regex construct that exhibits a semantic ambiguity, which should arguably be reported as an error during the syntactic analysis preceding the regex parsing.[3] Consider the following example: $r = ((a) + (b))(ab + \backslash 2)$. Here, we have a back-reference to the second pair of parentheses, which is involved in an alternation. What happens when this pair of parentheses is not instantiated? We adopt the following convention[4]:

If a control $\backslash i$ refers to the pair of parentheses i which has not been instantiated due to an alternation, we assume that pair of parentheses instantiated with $\emptyset$, thus $\backslash i$ will match $\emptyset$ (note that $\emptyset$ concatenated with any word or language yields $\emptyset$).

It turns out that although regex languages are not regular, they are the subject of a pumping lemma similar to that for regular languages [2]. We finally mention the complexity of membership problem for regex:

Theorem 4 ([1]). *The membership problem for regex is NP-complete.*

This theorem is true regardless of the interpretation of regex. Notice the big gap between this complexity and the complexity of the membership problem for regular expressions.

3. Regex machines: Regex automata systems

In this section we propose a system of finite automata with computations governed by a stack, that addresses the membership problem for regex. The purpose of this automata system is twofold: to give a theoretically sound method for implementing regex in practice, and to prove the closure property of regex under intersection with regular languages.

[3] In most programming languages these expressions are called "*bad regex*", and the recommendation is to avoid such expressions.
[4] All the proofs in this paper can be adapted to any other alternative semantics.

First we give a definition for a Regex Automata System (RAS), independent of the concept of regex. Let Σ be a finite alphabet and $\{u_1, v_1, \dots, u_n, v_n\}$ be a set of $2n$ variable symbols, $n \geq 1$. For $k \in \{1, \dots, n\}$ we denote by Σ_k the alphabet $\Sigma \cup \{u_1, v_1, \dots, u_{k-1}, v_{k-1}\}$, (thus, $\Sigma_1 = \Sigma$).

Let $n > 0$ and let $\left\{A_k = (\Sigma_k, Q_k, 0_k, \delta_k, F_k)\right\}_{k \in \{1,\dots,n\}}$ be a system of finite automata satisfying the following conditions:

(1) *for any $k \in \{1, \dots, n\}$, the variable symbol u_h appears as the label of at most one transition, and in at most one automaton A_k, with $h < k$. When this occurs, we write $u_h \prec u_k$ and say that "the instantiation of u_h is included in that of u_k". We further denote by $\preceq$ the transitive and reflexive closure of $\prec$, as order over the set $\{u_1, \dots, u_n\}$.*
(2) *for any $k \in \{1, \dots, n\}$, the variable symbol v_k does not appear as the transition label of any automaton A_h with $u_h \preceq u_k$.*

These two conditions have an important role for the correct operation of a RAS and in the relationship between regex and RAS. Note that by the first condition, u_n cannot appear as a transition label of any automaton A_k with $1 \leq k \leq n$.

If we denote $Q = \bigcup_{k=1}^{n} Q_k$, then we define a regex automata system (RAS) as a tuple $\mathcal{A} = (A_1, \dots, A_n, \Gamma, V_1, \dots, V_n)$ of n finite automata A_k, a stack Γ of depth at most n and storing elements of Q, and n buffers V_k that store words in Σ^*, $1 \leq k \leq n$. For improving the formalism, we will make no distinction between a buffer and its content, or between the stack and its content.

Our RAS $\mathcal{A}$ is described at any moment of its computation by a configuration of the following form: $(q, w, \Gamma, V_1, V_2, \dots, V_n)$, where $q \in Q$, $w \in \Sigma^*$, Γ is the stack content (of elements of Q), and the buffer V_k stores a word in Σ^* that has the role of instantiating the variable u_k, for all $k \in \{1, \dots, n\}$.

The computation starts with an initial configuration $(0_n, w, \varepsilon, \underbrace{\emptyset, \emptyset, \dots, \emptyset}_{n})$, and the system transits from configuration to configuration

$$(s, \alpha w, \Gamma^{(t)}, V_1^{(t)}, V_2^{(t)}, \dots, V_n^{(t)}) \mapsto (q, w, \Gamma^{(t+1)}, V_1^{(t+1)}, V_2^{(t+1)}, \dots, V_n^{(t+1)})$$

in one of the following circumstances:

(1) **letter-transition:** $\alpha = a \in \Sigma$, $s \in Q_k$, $q \in \delta_k(s, a)$, $\Gamma^{(t+1)} = \Gamma^{(t)}$, $V_h^{(t+1)} = V_h^{(t)}a$ for all h such that $u_k \preceq u_h$, and $V_h^{(t+1)} = V_h^{(t)}$ for all the other cases.
(2) **v-transition:** $\alpha \in \Sigma^*$, $s \in Q_k$, $q \in \delta_k(s, v_h)$, $\Gamma^{(t+1)} = \Gamma^{(t)}$, $V_h^{(t)} = \alpha$, $V_l^{(t+1)} = V_l^{(t)}\alpha$ for all l such that $u_k \preceq u_l$, and $V_l^{(t)} = V_l^{(t+1)}$ for all the other cases. Obviously, when $V_h^{(t)} = \emptyset$ this transition cannot be performed.
(3) **u-transition:** $\alpha = \varepsilon$, $s \in Q_k$, $r \in \delta_k(s, u_h)$, $q = 0_h$, $\Gamma^{(t+1)} = push(r, \Gamma^{(t)})$, $V_h^{(t+1)} = \varepsilon$, and $V_l^{(t+1)} = V_l^{(t)}$ for all $l \neq k$.
(4) **context switch:** $\alpha = \varepsilon$, $s \in F_h$ $(h \neq n)$, $q = top(\Gamma^{(t)})$, $\Gamma^{(t+1)} = pop(\Gamma^{(t)})$, and $V_l^{(t)} = V_l^{(t+1)}$ for all l.

If $f \in F_n$, then a configuration $(f, \varepsilon, \varepsilon, V_1, V_2, \dots, V_n)$ is final. A computation is successful if it reaches a final configuration. At the end of a successful computation, the buffer V_n will store the initial input word, whereas the buffers V_k with $1 \leq k < n$ contains the last word that has instantiated the variable u_k.

The difference between the variables u_k and v_k sharing the same buffer V_k, is that the variable u_k is "instantiated" (or reinstantiated) with the content of the buffer, while v_k simply uses the buffer to match a portion of the input word with the last instantiation of u_k. Note that the stack Γ may have at most $n - 1$ elements. If $qq' \in \Gamma$ (the top of the stack is to the right) with $q \in Q_k$ and $q' \in Q_h$, we have that $u_h \prec u_k$. Thus, Γ can be viewed as part of the finite control of $\mathcal{A}$. What makes a RAS more powerful than a finite automaton is the set of n buffers, each capable of holding an arbitrary long input.

In order to prove that RAS and regex are equivalent, we present a conversion of a regex into a RAS and vice versa. For our construction, the usual indexing of regex used in practice is not useful. We require another manner of indexing parentheses in a regex, to obey the following rules:

(1) *the entire expression is enclosed by a pair of parentheses;*
(2) *inner pairs of parentheses have an index smaller than those of the parentheses that surround them;*
(3) *if two pairs of parentheses enclosed in a outer one are not nested, then the left pair has a higher index than the right one.*

This order corresponds to inverse BFS (breadth-first search) order of traversing the parenthesis tree. We mention that the third condition above is not crucial, however, it helps the formalism.

One can easily transform a "classical" regex into one obeying the above rules. For example, the regex $({}_1({}_2a^*b)^*c)\backslash 2 + ({}_3a^*({}_4b + ba))\backslash 3$ is reindexed, and the back-references are adjusted as follows: $({}_5({}_4({}_2a^*b)^*c)\backslash 2 + ({}_3a^*({}_1b + ba))\backslash 3)$.

It is easy to observe that changing the rules of indexing in this manner and adjusting the back-references accordingly will not change the interpretation of the regex.

Let r be a regex with parentheses indexed according to this new convention. The parentheses of r are numbered as $1, 2, \dots, n$, and obviously, the nth pair of parentheses is the outmost one. To each pair of parentheses $({}_k\!-\!)$ we associate a variable symbol u_k, regardless of whether this pair is back-referenced or not. To each back-reference $\backslash k$ we associate another variable symbol v_k. These two sets of variables are used in the matching of a word as follows: u_k will store the content of the

kth parenthesis used in matching, whereas v_k will enforce the matching of an input sub-word with the already instantiated content of u_k.

To every pair of parentheses u_k we associate a regular expression r_k over Σ_k (the sub-expression enclosed by these parentheses), such that substituting the variable u_k with the corresponding regular expression r_k, and each variable v_k with $\backslash k$, we obtain the original regex r $(= r_n)$ corresponding to the variable u_n. We illustrate this breakdown in the following example.

Example 5. Let $r = (_5(_4(_2a^*b)^*c)\backslash 2 + (_3a^*(_1b + ba))\backslash 3)$. We have two sets of variables $\{u_1, u_2, u_3, u_4, u_5\}$ and $\{v_1, v_2, v_3, v_4, v_5\}$, and to each u_i we associate a regular expression as follows: $u_2 \to (a^*b) = r_2$, $u_4 \to (u_2^*c) = r_4$, $u_1 \to (b + ba) = r_1$, $u_3 \to (a^*u_1) = r_3$, and $u_5 \to (u_4v_2 + u_3v_3) = r_5$. Notice that these regular expressions have no other parentheses except the enclosing pair.

Denoting $\Sigma_k = \Sigma \cup \{u_1, \ldots, u_{k-1}\} \cup \{v_1, \ldots, v_{k-1}\}$, the expression r_k is a regular expression over Σ_k. If the variable u_h is used in regex r_k, i.e., $|r_k|_{u_h} > 0$, we say that $u_h \sqsubset u_k$. Note that if $u_h \sqsubset u_k$, then $u_h \not\sqsubset u_{k'}$, for all $k' \neq k$. In other words, once a variable u_h is used in an expression r_k, it will not be used again in another expression, for each pair of parentheses is transformed into an u-variable that "masks" the "inner" u-variable. This relation can be extended to an order relation $\sqsubseteq$ by applying the transitive and reflexive closures. During the matching of an input word with regex r, if $u_h \sqsubseteq u_k$, then each time we attempt to match a sub-word with the expression r_h, we have to consider updating the string that matches r_k as well, since the expression r_h is included in r_k. Notice the distinction between $\preceq$, defined in the context of a RAS, and $\sqsubseteq$ defined for a regex (and yet, the parallel between them is clear).

Anticipating Theorem 6, we outline here the parallel between regex and RAS. Given a regex r, we can construct an equivalent RAS $\mathcal{A} = (A_1, \ldots, A_n, \Gamma, V_1, \ldots, V_n)$, by associating to each expression r_k (in the breakdown of r as above) an automaton $A_k = (\Sigma_k, Q_k, 0_k, \delta_k, F_k)$ recognizing the language $L(A_k) = L(r_k)$. One can easily see that indeed, $\mathcal{A}$ verifies the RAS conditions. Vice versa, given a RAS $\mathcal{A} = (A_1, \ldots, A_n, \Gamma, V_1, \ldots, V_n)$, one can construct a corresponding regex r by reversing the previous construction: for each A_k we find the equivalent regular expression r_k over the alphabet Σ_k, and starting with r_n, we recursively substitute each symbol u_k by its corresponding regular expression r_k, and each symbol v_k with the back-reference $\backslash k$. We eventually obtain a regex over Σ. The conditions governing the structure of $\mathcal{A}$ ensure that the obtained regex r is indexed according to the new rules introduced in Section 3. From here, there is no problem to reindex r and adjust the back-references accordingly, to obtain a "classical" regex.

Note that there may be cases when this construction leads to back-references occurring to the left of the referenced parentheses. Indeed, $\mathcal{A}$ may have, in theory, transitions labeled with v_k that are triggered before u_k is instantiated (they can easily be detected). Those transitions are useless, however, in order to keep the definition of RAS simple, we did not impose restrictions for avoiding them. Consequently, the resulting regex r may have "orphan" back-references, which we agree to replace with $\emptyset$ and perform the proper simplifications.

Theorem 6. *RAS and regex are equivalent.*

Proof. We have already shown how a regex r can be associated with a RAS $\mathcal{A}$, by a two-way construction: $r \to \mathcal{A}$, and $\mathcal{A} \to r$. The outcome of these conversions is not unique, depending on the algorithms used to convert a finite automaton into a regular expression and vice versa. Given r and $\mathcal{A} = (A_1, \ldots, A_n, \Gamma, V_1, \ldots, V_n)$, we make the following remarks:

i) In the definition of transitions in $\mathcal{A}$ (Section 3), case (1) corresponds to a matching of an input letter, case (2) corresponds to a matching of a back-reference, while case (3) corresponds to ending the matching process for a parenthesis k – marking the moment when u_k has been instantiated, and can be used in a subsequent back-referencing (by a v_k).
ii) During a computation, $\mathcal{A}$ cannot transit along a transition labeled with a variable symbol v_k for which u_k has not been instantiated. This behavior is consistent with the common understanding of regex evaluation, where we cannot use a back-reference of a parenthesis which has not been matched yet (e.g, as a result of an alternation) – more precisely, we use $\emptyset$, equivalent to having no match.
iii) The operation of $\mathcal{A}$ is non-deterministic, since it follows closely the non-deterministic matching of a word by the regex r.
iv) $\preceq$ for $\mathcal{A}$ "coincides" with $\sqsubseteq$ for the corresponding r.

The idea of proving the equivalence of r and $\mathcal{A}$ is as follows. Consider a successful computation in $\mathcal{A}$, for some input $w \in L(\mathcal{A})$: $(0_0, w, \emptyset, \emptyset, \emptyset, \ldots, \emptyset) \mapsto^* (f_k, \alpha, \Gamma^{(t)}, V_0^{(t)}, V_1^{(t)}, \ldots, V_n^{(t)}) \mapsto^* (f_n, \varepsilon, \emptyset, V_1^{(l)}, V_2^{(l)}, \ldots, V_n^{(l)})$, where $f_k \in F_k$. In other words, in this computation we emphasize a configuration immediately before a context switch (case (4) in the description of transitions in $\mathcal{A}$ in Section 3). One can check that when this configuration has been reached, all buffers V_h, with $V_h \neq \emptyset$ and $u_h \preceq u_k$, will have as a suffix the word V_k corresponding to u_k, that is, the word that matches the kth pair of parentheses in r (or equivalently, which matches the expression r_k). Notice that when a variable u_k is involved in an iteration, the buffer V_k is "reset" at the beginning of each iteration and will eventually hold the last iterated value of u_k at that point of computation. At the end of computation, the set of buffers $\{V_k\}_{k=1}^n$ provide the matching sub-words used for parsing w according to r.

The converse argument works similarly. Given a word w in $L(r)$, one can construct a matching tree [2] for w, and the node corresponding to the kth pair of parentheses in r will hold a sub-word reconstructed in the buffer V_k during a successful computation of $\mathcal{A}$ on input w. □

Corollary 7. *The membership problem for regex has $O(mn)$ space complexity, where n is the number of pairs of parentheses in the regex and m is the length of the input word.*

Proof. Since regex and RAS are equivalent, we use RAS to decide the word membership. A RAS has at most as many buffers as number of pairs of parentheses in the regex (n), and the size of a buffer is at most m (the size of the input). Notice that the depth of the stack is at most n. □

Remark 8. We may have different semantic variations of regex, such as:

(1) Some regex implementations consider non-instantiated variables as ε, e.g., in UNIX Bourne shell interpretation. To adapt a RAS to this interpretation, we start the computation with the initial configuration $(0_n, w, \varepsilon, \underbrace{\varepsilon, \varepsilon, \dots, \varepsilon}_{n})$.

(2) We may consider that each reinstantiation of a variable u_k resets the values of all previously instantiated variables u_h such that $u_h \preceq u_k$. In this case, for step 3 we set $x_h^{(t+1)} = \emptyset$ or $x_h^{(t+1)} = \varepsilon$, for all $h \neq k$ such that $u_h \preceq u_k$, depending on the initial values for uninstantiated variables.

All the results of this paper can easily be adapted without effort to any regex semantics, including the ones implemented in the current programming environments.

From now on we assume without loss of generality that all components A_k of a RAS $\mathcal{A} = (A_1, \dots, A_n)$ are trim (all states and transitions are useful), and that no transition v_k can be triggered before a preceding transition u_k (these situations can be detected and such transitions v_k can be removed).

4. Main result: Intersection with regular languages

In this section we present a construction of a RAS that recognizes the intersection of a regex language with a regular language, based on the equivalence of regex with RAS. Because the orders $\preceq$ and $\sqsubseteq$ coincide for a regex and its corresponding RAS, we will only use $\preceq$. We now give some additional definitions and results.

Definition 9. We say that a regex r is in star-free normal form if

(1) every pair of parentheses included in a starred sub-expression is not back-referenced outside that sub-expression, i.e., in a sub-expression to the right of that starred sub-expression;
(2) all star operations are applied to parentheses.

This definition says that, in a star-free normal form regex, a pair of parentheses and its possible back-references occur only in the same sub-expression under a star operator. In other words, the following situation is avoided:

$$(\dots ({}_k-) \dots)^* \dots \backslash k \dots .$$

Example 10. The expressions $({}_1a)^*\backslash 1$ and $({}_2({}_1a^*)b\backslash 1)^*\backslash 1$ are not in star-free normal form, while $({}_2({}_1a)^*)\backslash 2$ and $({}_4({}_2({}_1a)^*)({}_3a)b\backslash 3)^*$ are.

Lemma 11. *For every regex r there exists an equivalent regex r′ in star-free normal form.*

Proof. The second condition can easily be satisfied, therefore we only consider expressions where star is applied to parentheses.

For the first condition, let u be a sub-expression under a star operator, which includes a pair of parentheses back-referenced outside u. The situation can be generically expressed as $\underbrace{(\dots ({}_k-) \dots)}_{u}{}^* \dots \backslash k \dots$. Our argument is based on the following straightforward equality: $u^* = (u^*u + \varepsilon)$. Then, we can rewrite the regex as follows:

$$\Big(\underbrace{(\dots (-) \dots)^*}_{u^*} \underbrace{(\dots ({}_h-) \dots)}_{u} + \varepsilon \Big) \dots \backslash h \dots$$

without changing the accepted language. Notice that we have adjusted the back-reference $\backslash k$ to a new value $\backslash h$, to account for the introduction of new pairs of parentheses during the process. The idea is to isolate two cases: when the iteration actually occurs, the case when we know exactly what an eventual back-reference will duplicate, or when the iteration does not occur (zero-iteration) and an eventual back-reference is set to $\emptyset$. A proof by induction on the number of parentheses that are back-referenced is straightforward. □

Remark 12. For a RAS obtained from a regex in star-free normal form, if a variable u_h is instantiated within a loop of an automaton A_k, then its value cannot be used by a transition labeled v_h, unless v_k belongs to the same loop.

Example 13. Let $r = ((a^*b)^*c\backslash 1)^*\backslash 1\backslash 2$. We rewrite it in star-free normal form as follows:

$$\big({}_{13}\big({}_{12}\big({}_{10}\big({}_8({}_4a)_4^*b\big)_8^*\big({}_7({}_3a)_3^*b\big)_7 + \varepsilon\big)_{10}c\backslash 7\big)_{12}^*$$
$$\big({}_{11}\big({}_9\big({}_6({}_2a)_2^*b\big)_6^*\big({}_5({}_1a)_1^*b\big)_5 + \varepsilon\big)_9c\backslash 5\big)_{11}\backslash 5\backslash 11\big)_{13}.$$

Let $B = (\Sigma, Q_B, 0_B, \delta_B, F_B)$ be a trim DFA. We consider the family of functions $\tau_w^B : Q_B \to Q_B$ defined as $\tau_w^B(q) = \delta_B(q, w)$. Since Q_B is finite, the number of functions $\{\tau_w^B \mid w \in \Sigma^*\}$ is also finite. These functions, together with composition and

τ^B_ε as identity, form a finite monoid: the transition monoid $\mathcal{T}_B$ of B. We partition Σ^* into equivalent classes, given by the equivalence relation of finite index $u \equiv_B v \Leftrightarrow \tau^B_u = \tau^B_v$ and let $W_B = \Sigma^*/\equiv_B$ be the quotient of Σ^* under $\equiv_B$. The transition functions τ^B, can now be indexed by elements of W_B, i.e., $\{\tau^B_c\}_{c \in W_B}$.

For every $c \in W_B$, we can construct a DFA $D_c = (\Sigma, Q_c, \delta_c, 0_c, F_c)$ such that $L(D_c) = c$. We can repeat the above construction for each automaton D_c, obtaining an equivalence relation $\equiv_c$, the functions $\{\tau^c_w\}_{w \in \Sigma^*}$, and the set of equivalence classes $W_c = \Sigma^*/\equiv_c$. Let $W_0 = W_B$. The above relations are of finite index, and, iterating again the above construction, we can define the following equivalence relations: let $\equiv_0$ be identical with $\equiv_B$, and

$$x \equiv_{l+1} y \quad \text{iff} \quad x \equiv_l y \text{ and there is } c \in W_l \text{ such that } x \equiv_c y.$$

These classes induced by B have the following property:

For any $l > 1$, if $w_1, w_2 \in c_l \in W_l$, then there is a unique $c_{l-1} \in W_{l-1}$ such that $w_1, w_2 \in c_{l-1}$, and we have both

$$\delta_{c_l}(i, w_1) = \delta_{c_l}(i, w_2) \quad \text{and} \quad \delta_{c_{l-1}}(i, w_1) = \delta_{c_{l-1}}(i, w_2). \tag{1}$$

In what follows, we consider two classes c and c' to be distinct if $c \in W_j$ and $c' \in W_{j'}$, with $j \neq j'$, thus we will make a difference between a class c and the language represented by the class $L(c)$. If $c \in W_j$, we denote $\Lambda(c) = j$. For $c \in W_l$ we define the function

$$\tau_c : \Big(Q_B \cup \bigcup_{c \subseteq c'} Q_{c'}\Big) \longrightarrow \Big(Q_B \cup \bigcup_{c \subseteq c'} Q_{c'}\Big)$$

by $\tau_c(i) = \delta_{c'}(i, w)$ for $w \in c$, where $i \in Q_B$ or $i \in Q_{c'}$, $c \subseteq c'$, $\Lambda(c') \leq \Lambda(c)$. Functions τ_c are well defined, based on property (1).

Theorem 14. *The family of regex languages is closed under the intersection with regular languages.*

Proof. Let r be a regex in star-free normal form (Lemma 11) such that the occurrence of u_k in the dependency tree of the automata system is in a level lower than or equal to the levels of any occurrence of the corresponding v_k (otherwise we surround u_k by the required number of parentheses) and let $B = (\Sigma, Q_B, 0_B, \delta_B, F_B)$ be a trim DFA with $m = \#(Q_B)$. We consider a RAS $\mathcal{C} = (C_1, C_2, \dots C_n)$, such that $L(\mathcal{C}) = L(r)$, $C_k = (Q_k, \Sigma_k, \delta_k, 0_k, F_k)$, where $\mathcal{C}$ is obtained by using the construction in Section 3 from regex r. Let l be the number of levels of the dependency tree of the automata system $\mathcal{C}$ and consider the equivalence classes W_j, $0 \leq j \leq l$, induced by the automaton B. Denote $Level_0 = \{k \mid u_k \not\preceq u_h,$ and for any $1 \leq h \leq n-1$, u_k is a label in $C_n\}$ and $Level_{j+1} = \{h \mid u_h \preceq u_k, k \in Level_j\}$ for $0 \leq j < l$. Thus, for a word $w \in L(\mathcal{C}) \cap L(B)$, for each $k \in Level_j$, at the end of the computation we can consider that $V_k \in L(c_k)$ and $c_k \in W_j$ ($j = \Lambda(c_k)$). For this computation, when we update a buffer V_h, we also update all buffers V_k such that $u_h \preceq u_k$. Hence, for $u_h \preceq u_k$, processing a word in a class c_h using the automaton C_h requires updates of words processed by automaton C_k, but updates of words processed by automaton C_k may or may not require updates of words processed by automaton C_h.

Hence, in the RAS $\mathcal{A}$ constructed for the intersection $L(\mathcal{C}) \cap L(B)$, a buffer V_k for $\mathcal{C}$ may turn into a set of buffers in $\mathcal{A}$, where variables are in sets of variables resulting from u_k and v_k respectively, considering all possible class instantiations.

The indices of new variables u_- and v_- of the RAS $\mathcal{A}$ must contain the index k of the module C_k to which they are related, and information about the instantiation classes $c_h \in W_{\Lambda(c_h)}$, where $u_h \preceq u_k$, $h < k$. For all k, $1 \leq k < n$, $k \in Level_j$, all $c \in W_l$, and $d \in \{1, 2\}$ we define:

$\mathcal{S}^k_{icd} = \{(\alpha_k, \alpha_{k-1}, \dots, \alpha_1) \mid \alpha_k = icd,$ and for all h, $1 \leq h < k$, $\alpha_h = 0$ or $\alpha_h = 1$ or there is $1 \leq h' < h \leq k$, $u'_h \prec u_h \preceq u_k$ such that $\alpha_h = i_h c_h d_h$, $\alpha_{h'} = i_{h'} c_{h'} d_{h'}$, $i_{h'} \in Q_{c_h}$ and $c_{h'} \in W_{\Lambda(c_h)+1}$, $d_h, d'_h \in \{1, 2\}$, $d_{h'} \leq d_h\}$.

For $k \in Level_j$ and $j > 1$, we denote $\mathcal{S}_k = \{S \in \mathcal{S}^k_{icd} \mid d = 1, 2, c \in W_j, i \in Q_{c'}, c' \in W_{j-1}\}$. For $k \in Level_j$ and $j = 1$, we denote $\mathcal{S}_k = \{S \in \mathcal{S}^k_{icd} \mid d = 1, 2, c \in W_{j-1}, i \in Q_B\}$ and $\mathcal{S}_n = \mathcal{S}_{n-1}$.

The projections $\pi_h : \mathcal{S}_k \longrightarrow (Q_{c_h} W_j \{1, 2\} \cup \{0, 1\})$, $c_h \in W_{j-1}$, are defined for $1 \leq h \leq k$ by $\pi_h(S) = \alpha_h$, where $S = (\alpha_k, \dots, \alpha_1)$.

The components of the RAS $\mathcal{A}$, corresponding to the variables resulting from the variable u_k are $A_{k,S}$, where $1 \leq k < n$, and $S \in \mathcal{S}_k$ verifies that $\pi_h(S) \neq 1$ for all $1 \leq h \leq k$.

The states of $A_{k,S}$ are in $Q_k \times Q_{c_k} \times \mathcal{S}_k$, $\pi_k(S) = i_k c_k d_k$, and variable labels are in $Q_h \times Q_{c_h} \times \mathcal{S}_h$ where $u_h \prec u_k$, $c_h \in W_{\Lambda(c_k)+1}$. Given a projection h of $S \in \mathcal{S}_k$ and α_h, we have the following interpretation:

- if $\alpha_h = i_h c_h d_h$, then c_h is a class for D_{c_k}, $i_h \in Q_{c_k}$, and d_h is 1 if this is the first instantiation of a variable resulting from u_h, and 2 if it is another (re)instantiation of a variable resulting from u_h.
- if $\alpha_h = 0$, then all variables resulting from u_h are not instantiated yet.
- if $\alpha_h = 1$, then at least one variable resulting from u_h has been instantiated and another one resulting from u_h is to be (re)instantiated. The information about previous instantiation is erased. This value is only possible for states.

Only transitions with variables of type u change instantiation classes for buffers and each such transition must be unique. We know when a transition with a variable resulting from v_k is possible, because the name of the states contains the last instantiation class.

There is only one state in Q where a variable u_k is instantiated, thus we denote by $init(k)$ the state in Q that has an outward transition labeled with u_k and by $Init = \{init(k) \mid 1 \leq k < n\}$. For the new modules, the states having transitions with variables resulting from u_k should only be allowed if one component is *init*(k) and the k component of the S name is 0 (first instantiation) or 1 (reinstantiation). If the k component is not in $\{0, 1\}$, then the previous instantiation of the variable resulting from u_k is considered, and there is no transition with variables resulting from u_k.

For reinstantiating a variable resulting from u_h in a module for a variable resulting from u_k, $u_h \prec u_k$, we need to consider all possible (re)instantiations of variables resulting from u_h as well as for some of the variables resulting from $u_{h'}$, with $u_{h'} \preceq u_h$. To achieve this, we define the set $E(S, h)$, where $S \in \mathcal{S}_k$, and c, c' are such that $S \in S^k_{icd}$, $c \in W_{\Lambda(c')+1}$ and $i \in Q_{c'}$.

$E(S, h) = \{S' \in \mathcal{S}^k_{icd} \mid \pi_h(S') = 1$ and for all $1 \le h'' < h' \le h$, if $u_{h''} \preceq u_{h'} \preceq u_h$ and $\pi_{h''}(S') = 1$, then $\pi_{h'}(S') = 1$, otherwise $\pi_{h'}(S) = \pi_{h'}(S')\}$.

Note that if $S' \in E(S, h)$, then $\pi_{h'}(S) = \pi_{h'}(S')$, for all $h' < k$ and $u_{h'} \not\preceq u_k$. The following set contains all cases for the reinstantiation of the new variables: $Choice(S) = \bigcup_{u_h \prec u_k} E(S, h)$. In this set, one component h and only some of the components h' with $u_{h'} \preceq u_h$ are set to 1, preparing them for a reinstantiation. For state names, the components of S which are reinstantiated must be 1, and after the u-transitions, they must be different from 0 or 1. The next set describes this situation: $Follow(S, h) = \{(\alpha_k, \ldots, \alpha_1) \mid \pi_h(S) \in \{0, 1\}, \alpha_h \notin \{0, 1\}$, and for all $1 \le h' < k$, $\alpha_{h'} \ne 1$, if $\pi_{h'}(S) = 0$, $\alpha_{h'} \in \{0, Q_{c'} W_{\Lambda(c_{h'})} 1\}$, and if $\pi_{h'}(S) = 1$, then $\alpha_{h'} \in \{Q_{c'} W_{\Lambda(c_{h'})} 2\}$, $\Lambda(c') + 1 = \Lambda(c_{h'})$, otherwise $\alpha_{h'} = \pi_{h'}(S)\}$.

Now we are ready to give the formal definitions for the modules of $\mathcal{A}$.

(1) For all k such that C_k does not have transitions labeled with variables $S = (i_k c_k d_k, \underbrace{\alpha_{k-1}, \ldots, \alpha_1}_{(k-1)})$:

$$A_{k,S} = \left(Q_k \times Q_{c_k}, \Sigma, (0_k, 0_{c_k}), \delta_{k,S}, F_{k,c_k}\right),$$

where $F_{kc_k} = F_k \times F_{c_k}$, and for all $(p, r) \in Q_k \times Q_{c_k}$ and $a \in \Sigma$:

$$\delta_{k,S}\big((p, r), a\big) = \{(q, \delta_{c_k}(r, a)) \mid q \in \delta_k(p, a)\}.$$

This is the case when back-references are not processed, thus the construction is the usual automata Cartesian product [7], $C_k \times D_{c_k}$. Note also that this corresponds to the case of a most inner pair of parentheses (i.e., with no dependencies).

(2) For all $k \in \{2, \ldots, n-1\}$ (case $k = 1$ does not involve any dependency) and $S \in \mathcal{S}_k$ with $S = (i_k c_k d_k, \underbrace{\alpha_{k-1}, \ldots, \alpha_1}_{(k-1)})$, we have:

$$\begin{aligned} A_{k,S} = \Big(& Q_k \times Q_{c_k} \times \mathcal{S}_k,\ \Sigma \cup \{u_{k',S'} \mid k' < k,\ S' \in \mathcal{S}_{k-1}\} \\ & \cup \{v_{k',S'} \mid k' < k,\ S' \in \mathcal{S}_{k-1}\},\ (0_k, 0_{c_k}, \underbrace{\alpha_{k-1}, \ldots, \alpha_1}_{k-1}),\ \delta_{k,S},\ F_{k,S}\Big), \end{aligned}$$

where $F_{k,S} = F_k \times F_{c_k} \times \{S \in \mathcal{S}_k \mid \pi_h(S) = \alpha_h, 1 \le h \le k\}$, and

i) **letter-transition:** for all $(p, i, S') \in Q_k \times Q_{c_k} \times \mathcal{S}_k$ and $a \in \Sigma$:

$$\begin{aligned} \delta_{k,S}\big((p, i, S'), a\big) &= \{(q, \delta_{c_k}(i, a), S') \mid q \in \delta_k(p, a) - Init\} \\ &\cup \{(q, \delta_{c_k}(i, a), T') \mid q \in \delta_k(p, a) \cap Init, T' \in Choice(S')\} \end{aligned}$$

ii) u-**transition:** for all $(p, i, S') \in Q_k \times Q_{c_k} \times \mathcal{S}_k$, such that there is $k', k' < k$ with $\pi_{k'}(S') \in \{0, 1\}$, $p \in init(k')$, and for all $T' \in Follow(S', k')$ s.t. $\pi_{k'}(T') = icd$:

$$\begin{aligned} \delta_{k,S}\big((p, i, S'), u_{k',T'}\big) &= \{(q, \tau_c(i), T') \mid q \in \delta_k(p, u_{k'}) - Init\} \\ &\cup \{(q, \tau_c(i), T'') \mid q \in \delta_k(p, u_{k'}) \cap Init, T'' \in Choice(T')\} \end{aligned}$$

iii) v-**transition:** for all $(p, i, S') \in Q_k \times Q_{c_k} \times \mathcal{S}_k$, $k' < k$ and $\pi_{k'}(S') = icd$:

$$\begin{aligned} \delta_{k,S}\big((p, i, S'), v_{k',S'}\big) &= \{(q, \tau_c(i), S') \mid q \in \delta_k(p, v_{k'}) - Init\} \\ &\cup \{(q, \tau_c(i), T') \mid q \in \delta_k(p, v_{k'}) \cap Init, T' \in Choice(S')\}. \end{aligned}$$

Note that after each transition triggered by $u_{k',T'}$ we reach states where the transition with $v_{k',T'}$ is possible, but the transitions with $v_{k',T''}$, $\pi_{k'}(T') \ne \pi_{k'}(T'')$ are not defined. This ensures a correlation between $u_{k',S'}$ and $v_{k',S'}$, which mimics the correlation between $u_{k'}$ and $v_{k'}$ in C_k.

(3) $A_{n,F_B} = \left(Q_n \times Q_B \times \mathcal{S}_n,\ \Sigma \cup \{u_{k',S'} \mid k' < n,\ S' \in \mathcal{S}_n\} \cup \{v_{k',S'} \mid k' < n,\ S' \in \mathcal{S}_n\},\ (0_n, \underbrace{0, \ldots, 0}_{(n-1)}),\ \delta_{n,F_B},\ F_{n,F_B}\right)$,

where $F_{n,F_B} = F_n \times F_B \times \mathcal{S}_n$, and

i) **letter-transition:** for all $(p, i, S') \in Q_k \times Q_B \times \mathcal{S}_n, a \in \Sigma$:

$$\delta_{n,F_B}\big((p, i, S'), a\big) = \{(q, \delta_B(i, a), S') \mid q \in \delta_n(p, a)\} \cup \{(q, \tau_c(i), T') \mid q \in \delta_n(p, a) \cap \mathit{Init}, T' \in \mathit{Choice}(S')\}$$

ii) u**-transition:** for all $(p, i, S') \in Q_n \times Q_B \times \mathcal{S}_n$, such that there is $k', k' < k$ with $\pi_{k'}(S') \in \{0, 1\}$ $p \in \mathit{init}(k')$, and for all $T' \in \mathit{Follow}(S', k')$ s.t. $\pi_{k'}(T') = \mathit{icd}$:

$$\delta_{n,F_B}\big((p, i, S'), u_{k',T}\big) = \{(q, \tau_c(i), T') \mid q \in \delta_k(p, u_{k'})\} \cup \{(q, \tau_c(i), T'') \mid q \in \delta_n(p, u_{k'}) \cap \mathit{Init}, T'' \in \mathit{Choice}(T')\}$$

iii) v**-transition:** for all $(p, i, S') \in Q_n \times Q_B \times \mathcal{S}_k, k' < n$ and $\pi_{k'}(S') = \mathit{icd}$:

$$\delta_{n,F_B}\big((p, i, S'), v_{k',S'}\big) = \{(q, \tau_c(i), S') \mid q \in \delta_n(p, v_{k'})\} \cup \{(q, \tau_c(i), T') \mid q \in \delta_n(p, v_{k'}) \cap \mathit{Init}, T' \in \mathit{Choice}(S')\}.$$

Considering that A_{n,F_B} is the "main" automaton of our newly constructed RAS, the dependence between the constituent automata is straightforward. Let H be the number of automata in $\mathcal{A}$.

We make the following observations that justify the correctness of our construction:

(1) $\mathcal{A}$ is indeed a RAS. The transitions with $u_{(k',T')}, \pi_{k'}(T') = \mathit{icd}$, are only possible for states (p, i, S') in $Q_k \times Q_{\mathcal{C}_k} \times \mathcal{S}_k$ with $\pi_{k'}(S') \in \{0, 1\}$. Since p is unique for k', so is (p, i, S') for $u_{k',S'}$.

(2) If in the RAS $\mathcal{A}$ we consider only the first component of each state and ignore the S-subscript, we observe that a computation in $\mathcal{A}$ for an input word w is successful if and only if there exists a successful computation for w in this reduced version of $\mathcal{A}$, since all automata $A_{k,S}$ are identical with C_k, for all S (we have a surjective morphism from $A_{k,S}$ to C_k). For a fixed k, the buffers $V_{k,S}$ in $A_{k,S}$ are not simultaneously used, therefore it does not matter if for each k we use one buffer or several buffers.

The subtle point in this construction is to avoid the danger of using a back-reference $v_{k,S}$ corresponding to a variable $u_{k,S}$ that does not represent the last u_k instance, i.e., it is not $u_{k,S}$, but rather $u_{k,S'}$ for some index S' with $\pi_k(S) \neq \pi_k(S')$. However, this problem is avoided by using a RAS obtained from a regex in star-free normal form. This guarantees that every time we reinstantiate $u_{k',S'}$, we update the projection k' of S'; therefore, all the other variables $u_{k',S''}$ with $\pi_{k'}(S') \neq \pi_k(S'')$ are not on the path for $u_{k',S'}$. Indeed, $\pi_{k'}(S'') = \pi_{k'}(S')$, for all states following $u_{k',S'}$ in a successful computation path, since star is only applied to a reinstantiated variable (variable between parentheses). Thus, only the transitions with $v_{k',S'}$ are possible. The synchronization is done using the k' projection of the index S'.

(3) For every transition: $((s, i, S), \alpha w, \Gamma^{(t)}, V_1^{(t)}, V_2^{(t)}, \ldots, V_H^{(t)}) \mapsto_{\mathcal{A}} ((q, j, T), w, \Gamma^{(t+1)}, V_1^{(t+1)}, V_2^{(t+1)}, \ldots, V_H^{(t+1)})$, we have that $i = j$ and $\alpha = \varepsilon$, or $\delta_B(i, \alpha) = j$ (in the case when α is matching a variable $\tau_c(i) = j, \alpha \in L(c)$ or α is a letter).

In conclusion, for every transition

$$((0_n, 0_B, \underbrace{0, \ldots, 0}_{n-1}), w, \emptyset, \underbrace{\emptyset, \emptyset, \ldots, \emptyset}_{H}) \mapsto^*_{\mathcal{A}} (q, \varepsilon, \emptyset, V_1, V_2, \ldots, V_H)$$

we have that: $\delta_B(0_B, w) = q'$ and

$$(0, \alpha w, \Gamma^{(t)}, V_1^{(t)}, V_2^{(t)}, \ldots, V_n^{(t)}) \mapsto_{\mathcal{C}} (q, w, \Gamma^{(t+1)}, V_1^{(t+1)}, V_2^{(t+1)}, \ldots, V_n^{(t+1)}),$$

which means that $w \in L(\mathcal{A})$ iff $w \in L(B)$ and $w \in L(\mathcal{C})$.

Thus, the automata system $\mathcal{A}$ recognizes the intersection of $L(\mathcal{C})$ and $L(B)$, proving that the intersection is a regex language. □

5. Consequences and conclusion

We use Theorem 14 to show that a few remarkable languages, such as the mirror language, are not regex languages. In [3,4] it was proved that the following languages satisfy neither the regex, nor the PE pumping lemma:

$$L_1 = \{(aababb)^n(bbabaa)^n \mid n \geq 0\}, \qquad L_2 = \{a^n b^n \mid n \geq 0\},$$
$$L_3 = \{a^{2n} b^n \mid n \geq 0\}, \qquad L_4 = \{a^n b^n c^n \big| n \geq 0\},$$
$$L_5 = \{\{a, b\}^n c^n \mid n \geq 0\}, \qquad L_6 = \{\{a, b\}^n c \{a, b\}^n \mid n \geq 0\}.$$

Since the pumping lemmas for regex and PE are essentially the same, it is clear that all these languages are not regex languages. This helps us to infer that some other languages, more difficult to control, are not regex languages – as the following result shows.

Corollary 15. *The following languages are not regex languages:*

$$L_7 = \{ww^R \mid w \in \Sigma^*\}, \qquad L_8 = \left\{w \mid w = w^R\right\}, \qquad L_9 = \left\{w \mid |w|_a = |w|_b\right\},$$
$$L_{10} = \left\{w \mid |w|_b = 2|w|_a\right\}, \qquad L_{11} = \left\{w \mid |w|_a = |w|_b = |w|_c\right\},$$
$$L_{12} = \left\{w \mid |w|_a + |w|_b = |w|_c\right\}, \qquad L_{13} = \left\{ucv \mid |u|_a + |u|_b = |v|_a + |v|_b\right\}.$$

Proof. We observe that: $L_7 \cap (aababb)^*(bbabaa)^* = L_8 \cap (aababb)^*(bbabaa)^* = L_1$, $L_9 \cap a^*b^* = L_2$, $L_{10} \cap a^*b^* = L_3$, $L_{11} \cap a^*b^*c^* = L_4$, $L_{12} \cap (a+b)^*c^* = L_5$, and $L_{13} \cap (a+b)^*c(a+b)^* = L_6$. If any of $L_7, \dots, L_{13}$ was a regex language, so would be its corresponding intersection, leading to a contradiction.

We should mention that none of the languages $L_7, \dots, L_{13}$ could be proven to be non-regex by pumping lemma alone. As a theoretical application of the closure property, some previous results involving elaborate proofs, such as Lemma 3 in [2], are immediately rendered true by Theorem 14. Consequently, we also infer that the new family of regex languages is not closed under shuffle with regular languages.

To conclude, in this paper we have defined a machine counterpart of regex, Regex Automata Systems (RAS), and used them to give an answer to an open problem reported in [2], namely, whether regex languages are closed under the intersection with regular languages. We have provided a positive answer to this question, and used this closure property to show that several anthological languages, such as the mirror language, the language of palindromes or the language of balanced words, are not regex – thus revealing some of the limitations of regex unforeseen before. Regex automata systems have also a practical impact: they give a rigorous method for implementing regex in programming languages and they avoid semantic ambiguities.

It remains open whether regex languages are closed under intersection. We conjecture that they are not, since in the proof for the closure under the intersection with regular languages we used in a crucial manner the transition monoid of a DFA, and its corresponding equivalence of finite index. Other open problems include the relation between regex and other formalisms, such as the pattern expressions [4] or grammar system [5].

References

[1] A.V. Aho, Algorithms for finding patterns in strings, in: Jan van Leeuwen (Ed.), Handbook of Theoretical Computer Science, in: Algorithms and Complexity, vol. A, Elsevier and MIT Press, 1990, pp. 255–300.
[2] C. Câmpeanu, K. Salomaa, S. Yu, A formal study of practical regular expressions, IJFCS 14 (6) (2003) 1007–1018.
[3] C. Câmpeanu, N. Santean, On pattern expression languages, Technical Report CS-2006-20, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2006.
[4] C. Câmpeanu, S. Yu, Pattern expressions and pattern automata, IPL 92 (2004) 267–274.
[5] Erzsébet Csuhaj-Varjú, Jürgen Dassow, Jozef Kelemen, Gheorghe Păun, Grammar systems, in: A Grammatical Approach to Distributed and Cooperation, Institute of Mathematics, Gordon and Breach Science Publishers, The Romanian Academy of Sciences, Bucureşti, Romania.
[6] J.E.F. Friedl, Mastering Regular Expressions, O'Reilly & Associates, Inc., Cambridge, 1997.
[7] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, Reading Mass, 2006.
[8] A. Salomaa, Theory of Automata, Pergamon Press, Oxford, 1969.
[9] A. Salomaa, Formal Languages, Academic Press, New York, 1973.
[10] S. Yu, Regular languages, in: A. Salomaa, G. Rozenberg (Eds.), Handbook of Formal Languages, Springer Verlag, 1997, pp. 41–110.