**CS–2007–10**

# Addressing an Open Problem on Regex

*Cezar Câmpeanu and Nicolae Santean*

*Technical Report 10*

David R. Cheriton School of Computer Science
University of Waterloo

*2007*

# Addressing an Open Problem on Regex

Cezar Câmpeanu[1] and Nicolae Santean[2]

[1] Department of Computer Science and IT, University of Prince Edward Island
[2] School of Computer Science, University of Waterloo

**Abstract.** In this paper we revisit the semantics of extended regular expressions (regex), defined succinctly in the 90's (A. V. Aho) and rigorously in 2003 by Câmpeanu, Salomaa and Yu, when the authors reported an open problem, namely whether regex languages are closed under the intersection with regular languages. We give a positive answer to this question; 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 anthological languages, such as the mirror language, the language of palindromes and the language of balanced words are not regex languages.

**Keywords:** Extended Regular Expression, Regex Automata System, Regex

## 1 Introduction

Pattern expressions (PE) and extended regular expressions (regex) have recently been the subject of scrutiny for formal language theorists, in an effort to provide a conceptual basis to the existing implementations of regular expressions. Regular expressions are powerful programming tools present in many language implementations 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 that their theoretical counterpart, mainly due to the introduction of a new operator, that of back-referencing. This new feature allows to express patterns (repetitions) in words, thus, 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 "\1" is a reference to (copy of) the content of the first pair of parentheses.

The current implementation of extended regular expressions is plagued by many conceptual problems, which can readily be demonstrated on any system with the above-mentioned software. For example, the use of Perl 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 () and ()*. Moreover, by definition, a backreference should replicate the last match of its corresponding paranthesis if such mach has occurred, or *emptyset* otherwise. However, in the following runtime example this is not the case: $((b)|(a)) * c\backslash 2\backslash 3$ matches *babbcba* in some implementations. Here, the behaviour suggests that although in the last iteration the third parenthesis is not matched, it is still backreferenced with a value from a previous iteration. Thus, we observe implementation inconsistencies and lack of standard semantics in such pathological situations, where there are more than one possible interpretations of regex semantics. Moreover, these problems transcend the practice. For example, in neither [1] nor [2] — where extended regular expressions have been defined theoretically — there is a resolution on the meaning of $((a)|(b))\backslash 2\backslash 3$, for example.

This unfortunate status quo, of having a flawn regex implementation, as well an incomplete theoretical foundation, has recently lead to an increased research effort aiming at their better understanding. Among the most recent work we mention that of Câmpeanu et al., who have initiated a rigorous formalism for regex in [2], and an alternative to this formalism, given by pattern expressions, in [4].

The present paper continues this 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. In [3], we have answered this question positively for pattern expression languages, and it was believed for a while that this answer will extend naturally to regex languages, invoking the equivalence of these formalisms. However, we have recently found evidence that this is not the case. For example, we conjecture that the regex $((a^*b)\backslash 2)^*$ can not be converted into a pattern expression, due to the outer iteration which can change its pattern at every repetition. In fact, we believe that pattern expression languages are not even closed under the Kleene star - closure holding trivially for regex. Thus, the open problem in [2] has remained standing, despite our new results in [3]. Consequently, we revisit this problem and we prove rigorously that regex languages are closed under intersection with regular languages. In Section 2 we give a background on extended regular expressions. Section 3 presents a machine counterpart for regex: the regex automata system (RAS), and it shows their equivalence with regex. In order to prove the mentioned closure property, we first introduce a normalization of regex in Section 4, and use the RAS formalism to construct an automata system for the intersection of a regex langauge and a regular language. Section 5 presents a few consequences of this closure property and concludes the paper.

## 2 Notations and Definitions

Let $\Sigma$ be an alphabet, that is, a finite set of symbols (or letters). By $\Sigma^*$ we denote all words (strings of symbols) over $\Sigma$, and by $\varepsilon$ we denote the empty word, i.e., the word with no symbols. 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 symbols in $w$). A language $L$ is a subset of $\Sigma^*$. The cardinality of a set $X$ is denoted by $\#(X)$. For other notions and notations we refer the reader to [9, 8, 6].

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

**Definition 1.** *A regex over $\Sigma$ 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 $\varepsilon$). Beside 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 $i$'th pair of parentheses, where parentheses are indexed according to the occurrence sequence of their left parentheses.*

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 subword identical with the subword which has matched the parenthesis $i$ in $r$. There is one exception to this rule:
2. If the $i$'th 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 $i$'th pair of parentheses as given by its last iteration.

*Example 1.* 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 $aabaaabaab \notin L(r)$.

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. 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[3]:

> *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 $\emptyset$ (remember 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 1.** *(Aho, 1990) The membership problem for regex is NP-complete.*

## 3 Regex Machines: Regex Automata Systems

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

Let $\Sigma$ be a finite alphabet and $\{u_1, v_1, \ldots, u_n, v_n\}$ be a set of $2n$ variable symbols, $n \geq 1$. By $\Sigma_i$ we denote the alphabet $\Sigma \cup \{u_1, v_1, \ldots, u_{i-1}, v_{i-1}\}$, for $i \in \{1, \ldots, n\}$ (thus, $\Sigma_1 = \Sigma$). A regex automata system (RAS) is a tuple $\mathcal{A} = (A_1, \ldots, A_n)$ of $n$ finite automata, such that $A_i = (\Sigma_i, Q_i, 0_i, \delta_i, F_i)$, with $Q_i = \{0_i, 1_i, \ldots, \#(Q_i)_i\}$, and which satisfies the following conditions: (1) *for all $i, j \in \{1, \ldots, n\}$, the variable symbol $u_i$ can occur as the label of at most one transition in automaton $A_j$*, and (2) *if $u_i \in \Sigma_j$ then $u_i \notin \Sigma_{j'}$, for all $j' \neq j$.* If we denote $Q = \cup_{i=0}^n Q_i$, our RAS $\mathcal{A}$ is described at each computation step by a configuration of the following form: $(q, w, S, x_1, x_2, \ldots, x_n)$, where $q \in Q$, $w \in \Sigma^*$, $S$ is a stack of elements taken from $Q$, and $\{x_i\}_{i=1}^n$ is a set of buffers capable of holding words in $\Sigma^*$: the buffer $x_i$ will store a word matching variable $u_i$.

The computation starts with an initial configuration $(0_0, w, \emptyset, \underbrace{\emptyset, \emptyset, \ldots, \emptyset}_{n})$, and the system transits from configuration to configuration as follows

$$(s, \alpha w, S^{(t)}, x_1^{(t)}, x_2^{(t)}, \ldots, x_n^{(t)}) \mapsto (q, w, S^{(t+1)}, x_1^{(t+1)}, x_2^{(t+1)}, \ldots, x_n^{(t+1)})$$

[3] All the proofs in this paper can easily be adapted to any other alternative semantics.

in one of the following circumstances:

1. **letter-transition:** $\alpha = a \in \Sigma$, $s \in Q_i$, $q \in \delta_i(s,a)$, $S^{(t+1)} = S^{(t)}$, $x_j^{(t+1)} = x_j^{(t)}a$ for all $j \geq i$ such that $u_i \preceq u_j$, and $x_j^{(t+1)} = x_j^{(t)}$ for all the other cases.
2. $v$**-transition:** $\alpha \in \Sigma^*$, $s \in Q_k$, $q \in \delta_k(s, v_i)$, $S^{(t+1)} = S^{(t)}$, $x_i = \alpha$, $x_j^{(t+1)} = x_j^{(t)}\alpha$ for all $j \geq k$ such that $u_k \preceq u_j$, and $x_j^{(t)} = x_j^{(t+1)}$ for all the other cases. Obviously, when $x_i = \emptyset$, this transition can not be performed.
3. $u$**-transition:** $\alpha = \varepsilon$, $s \in Q_i$, $r \in \delta_i(s, u_k)$, $q = 0_k$, $S^{(t+1)} = push(r, S^{(t)})$, $x_k^{(t+1)} = \varepsilon$, and $x_j^{(t+1)} = x_j^{(t)}$ for all $j \neq k$.
4. **context switch:** $\alpha = \varepsilon$, $s \in F_i$ $(i \neq n)$, $q = top(S^{(t)})$, $S^{(t+1)} = pop(S^{(t)})$, and $x_j^{(t)} = x_j^{(t+1)}$ for all $j$.

If $f \in F_n$, then the configuration $(f, \varepsilon, \emptyset, x_1, x_2, \ldots, x_n)$ is final. A computation is successful if it lands on a final configuration while it has consumed the entire input. Incidentally, at the end of a successful computation, the buffer $x_n$ will store exactly the initial input word. Notice that the stack $S$ may have at most $n-1$ elements, thus it 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 being 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, we require another manner of indexing parentheses in a regex; that is, inner pairs of parentheses will have indices smaller than those for parentheses which surround them. For example, the regex $(_1(_2a^*b)^*c)\backslash 2 + (_3a^*(_4b + ba))\backslash 3$ is re-indexed, and the back-references are adjusted, as follows: $(_2(_1a^*b)^*c)\backslash 1 + (_4a^*(_3b + ba))\backslash 4$. It is clear that this method of indexing does not change the interpretation of such regex. Notice also that there may be more than one way to index a regex according with this method, and we will enforce uniqueness by adding the second convention: *if two pairs of parentheses are not nested, than the left pair has a smaller index than the right one.*

Let $r$ be a regex with parentheses indexed according to this new convention. Without loss of generality, we assume that $r$ has an enclosing pair of parentheses, otherwise we consider the regex $(r)$ instead. The parentheses of $r$ are numbered as $1, 2, \ldots, n$, and obviously, the $n$'th pair of parentheses is the outermost one. To each pair of parentheses $(_i \ldots)$ we associate a variable symbol $u_i$, regardless on whether this pair is back-referenced or not (we reserve variable symbols in order to simplify the formalism). For each back-reference $\backslash i$ in $r$ we associate another variable symbol $v_i$. The interpretation of these two sets of variables becomes apparent during the matching of a word: $u_i$ will store the content of $i$'th parenthesis used in matching, whereas $v_i$ will enforce the matching of a sub-word with the already-instantiated content of $u_i$.

For every parenthesis $u_i$ we associate a regular expression $r_i$ over $\Sigma_i$, such that substituting the variable $u_i$ with the corresponding regular expression $r_i$, and each variable $v_i$ with $\backslash i$, we obtain the original regex $r$ $(= r_n)$, corresponding to the variable $u_n$.

*Example 2.* Let $r = (_5(_2(_1a^*b)^*c)\backslash 1 + (_4a^*(_3b + ba))\backslash 4)$. 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: $u_1 \to (a^*b) = r_1$, $u_2 \to (u_1^*c) = r_2$, $u_3 \to (b + ba) = r_3$, $u_4 \to (a^*u_3) = r_4$, and $u_5 \to (u_2v_1 + u_4v_4) = r_5$. Notice that these regular expressions have no parentheses beside the enclosing ones.

As mentioned before, we denote $\Sigma_i = \Sigma \cup \{u_1, \ldots, u_{i-1}\} \cup \{v_1, \ldots, v_{i-1}\}$. Then the expression $r_i$ is a regular expression over $\Sigma_i$. If the variable $u_i$ is used in regex $r_j$, i.e., $|r_j|_{u_i} > 0$, we say that $u_i \preceq u_j$. This relation is transitive and antisymmetric, and it becomes reflexive when we add $u_i \preceq u_i$, for all $0 \leq i \leq n$. If $u_i \preceq u_j$, during the matching of an input word with regex $r$, each time we attempt to match a sub-word with the expression $r_i$, we have to consider updating the string which match $r_j$ as well, since the expression $r_i$ is a dependency of $r_j$.

Given the regex $r$, we now construct an equivalent RAS $\mathcal{A} = (A_1, \ldots, A_n)$, by associating to each expression $r_i$ an automaton $A_i = (\Sigma_i, Q_i, 0_i, \delta_i, F_i)$ recognizing the language $L(A_i) = L(r_i)$. One can easily see that indeed, $\mathcal{A}$ verifies the RAS conditions:

1. Each automaton $A_i$ uses only variable symbols $u_j$ and $v_j$ with $j < i$; in particular, $A_1$ has no variable symbols. This is due to the way we index parentheses in $r$, and to the fact that within the pair $(_i \ldots)$, one can not have a back-reference $\backslash i$ (we do not allow self-referencing in regex). Also, recall that a back-reference stands always to the right of its corresponding pair of parentheses, whereas the index of un-nested parentheses increases from left to the right.
2. In each automaton $A_i$, and for any index $j$ , the symbol $u_j$ is the label of at most one transition. Indeed, since the symbol $u_j$ corresponds to the $j$'th pair of parentheses in $r$, it is clear that there is at most one occurrence of $u_j$ in any expression $r_i$ (there is an unique pair of parentheses indexed with $j$).

Vice versa, given a RAS $\mathcal{A} = (A_1, \ldots, A_n)$, one can construct a corresponding regex $r$ by reversing the previous construction: for each $A_i$ we find the equivalent regular expression $r_i$ over the alphabet $\Sigma_i$, and starting with $r_n$ we recursively substitute each symbol $u_i$ by its corresponding regular expression $r_i$, and each symbol $v_i$ with the back-reference $\backslash i$, placing $\backslash i$ always at the end of each alternation. We eventually obtain a regex over $\Sigma$. The conditions governing the structure of $\mathcal{A}$ ensure that the obtained regex $r$ is well-formed.

**Theorem 2.** *RAS are equivalent with regex.*

*Proof. (sketch)* 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 $\mathcal{A} \to r$. These conversions are not unique, depending on the algorithms used to convert a finite automaton into a regular expression and vice versa. Given $r$ and $\mathcal{A} = (A_1, \ldots, A_n)$, we make the following remarks:

- In the definition of transitions in $\mathcal{A}$, case 1 corresponds to a match of a letter, case 2 corresponds to a match of a back-reference, case 3 corresponds to starting of a match for a parenthesis $k$, while case 3 corresponds to ending the match for parenthesis $i$ – marking the moment when $x_i$ (the value of variable $u_i$) can be used in a subsequent back-referencing.
- During a computation, $\mathcal{A}$ cannot transit along a transition labeled with a variable symbol $v_i$ for which $u_i$ has not been instantiated. This behaviour is consistent with the common understanding of regex evaluation, where we can not use a back-reference of a parenthesis which has not been matched (e.g, as a result of an alternation) – more precisely, we use $\emptyset$, equivalent with no match.
- The operation of $\mathcal{A}$ is non-deterministic, since it follows closely the non-deterministic matching of a word against the regex $r$.

The idea of proving the equivalence of $r$ and $\mathcal{A}$ is as follows. Consider a successful computation in $\mathcal{A}$, for some input $w \in L(\mathcal{A})$:

$$(0_0, w, \emptyset, \emptyset, \emptyset, \ldots, \emptyset) \mapsto^* (f_i, \alpha, S^{(t)}, x_0^{(t)}, x_1^{(t)}, \ldots, x_n^{(t)}) \mapsto^* (f_n, \varepsilon, \emptyset, x_1^{(l)}, x_2^{(l)}, \ldots, x_n^{(l)}),$$

where $f_i \in F_i$. In other words, in this computation we emphasize a configuration immediately before a context-switch (case 4 in the description of transitions in $\mathcal{A}$). One can check that when this configuration has been reached, all buffers $x_j$, with $x_j \neq \emptyset$ and $u_j \preceq u_i$, hold words corresponding with $u_j$, that is, words which match the $j$'th pair of parentheses in $r$ (in other words, which match the expression $r_j$). Notice that when a variable $u_j$ is involved in an iteration, the buffer $x_j$ holds the last iterated value of $u_j$ at that point in computation. At the end of a successful computation, the set of buffers $\{x_i\}_{i=1}^n$ 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 with $i$-th pair of parentheses in $r$ will hold a sub-word which is reconstructed in the buffer $x_i$ during a successful computation of $\mathcal{A}$ on input $w$.

Although the remaining proof details require an intricate formalism, they are straightforward and will be omitted. $\square$

**Corollary 1.** *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 is equivalent with RAS, we use RAS to decide word membership. A RAS has one stack of depth bound by the number $n$ of pairs of parentheses in regex, and it has $n$ buffers, each holding a word at most as long as the input word. $\square$

From now on we assume without loss of generality that all components $A_i$ of a RAS $\mathcal{A} = (A_1, \ldots, A_n)$ are trim (all states and transitions are useful).

## 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 and a regular language, based on the equivalence of regex with RAS. We first give some additional definitions and results.

**Definition 2.** *We say that a regex $r$ is in free-star normal form if*

1. *every parenthesis 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 a free-star normal form regex is a regex where a pair of parentheses and its possible back-references occurs only in the same sub-expression under a star operator.

*Example 3.* The expression $(a)^*\backslash 1$, and $((a*)b\backslash 2)*\backslash 2$ are not in star-free normal form, while $((a)*)\backslash 1$, $(a)*(a)\backslash 2$, and $(((a)*)(a)b\backslash 4)*(((a)*)(a)b\backslash 6)\backslash 6$ are.

**Lemma 1.** *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. Our argument is based on the following straightforward equality: $u^* = (u^*u+\varepsilon)$. This equality isolates two cases: when the iteration actually occurs, case in which we know exactly what an eventual back-reference will replicate, and the situation when the iteration does not occur (zero-iteration), case in which an eventual back-reference would be set to $\emptyset$. The proof is done by induction on the number of parentheses that are back-referenced. If there are no back-references, there is nothing to prove. Let us assume that we have a regex where a parenthesis is starred and it is also back-referenced. We consider the most exterior pair of parentheses containing a starred pair of parentheses that is back-referenced. We do not ignore the case when a most exterior parenthesis has a star and it is also back-referenced: in this case, our back-reference can refer to this pair of parentheses. This means that our expression $r$ has the form $\alpha(\beta({}_i\gamma)_i\mu)^*\zeta\backslash\{i\}\rho$, where $\beta, \mu$ can be empty. The new expression is

$$\alpha(\beta({}_i\gamma)_i\mu)^*(\beta({}_{i+k}\gamma)_{i+k}\mu+\varepsilon)\zeta\backslash\{i+k\}\rho = \alpha(\beta({}_i\gamma)_i\mu)^*(\beta({}_{i+k}\gamma)_{i+k}\mu)\zeta\backslash\{i+k\}\rho,$$

where $k = |(\beta({}_i\gamma_i)\mu)|_{(} + |\beta|_{(} + 1$. In this case, we can drop the $\varepsilon$, since $\backslash\{i+k\}$ becomes $\emptyset$.

We can see that the starred parenthesis $i$ is no longer back-referenced in the sub-expression to the right of $)_i$. The expressions: $\alpha$, $\beta({}_{i+k}\gamma)_{i+k}\mu+\varepsilon$, and $\beta({}_i\gamma)_i\mu$ have fewer back-referenced parentheses, therefore, using our inductive hypothesis, we can equivalently write all these sub-expressions in a star-free normal form. This means that the original expression can be written in star-free normal form. $\square$

*Remark 1.* For a RAS obtained from a regex in star-free normal form, if a variable $u_i$ is instantiated within a loop of an automaton $A_j$, then its value can not be used by a transition labeled $v_i$ which does not belong to that loop.

*Example 4.* Let $r = ((a^*b)^*c\backslash 2)^*\backslash 2\backslash 1$. We rewrite it in star free normal form as follows:
$r = ({}_1({}_2a^*b)_2^*c\backslash 2)_1^*\backslash 2\backslash 1$
$\sim (((a)^*b)^*c\backslash 2)^*\backslash 2\backslash 1$
$\sim (((a)^*b)^*(((a)^*b)+\varepsilon)c\backslash 5)^*\backslash 5\backslash 4$
$\sim (((a)^*b)^*(((a)^*b)+\varepsilon)c\backslash 5)^*((((a)^*b)^*(((a)^*b)+\varepsilon)c\backslash 11)|\varepsilon)\backslash 11\backslash 10$
$\sim (((a)^*b)^*(((a)^*b))c\backslash 5)^*((((a)^*b)^*(((a)^*b))c\backslash 11))\backslash 11\backslash 10.$

**Theorem 3.** *The family of regex languages is closed under the intersection with regular languages.*

*Proof.* Let $L = L(r)$ be a regex language and $R$ be a regular language accepted by a trim DFA $B = (\Sigma, Q_B, 0, \delta_B, F_B)$, with $m = \#(Q_B)$. We consider a RAS $\mathcal{C} = (C_1, C_2, \ldots C_n)$, $C_k = (Q_k, \Sigma_k, \delta_k, 0_k, F_k)$, such that $L(\mathcal{C}) = L(r)$, where $\mathcal{C}$ is obtained using the construction in Section 3 from regex $r$ assumed in star-free normal form (Lemma 1). We now construct a RAS which simulates the run of $\mathcal{C}$ in "parallel" with $B$. The simulation goes in parallel when $\mathcal{C}$ transits from state to state based on letters in $\Sigma$. When $\mathcal{C}$ meets a transition labeled with a variable name "$u$" or "$v$", $B$ is put on hold, and $\mathcal{C}$ calls the proper module which takes over the resolution of $u$, or advances with the content of $v$ in parallel with $B$. Whenever a module uses a transition labeled with a letter in $\Sigma$, $B$ is revived and advances again in parallel with $\mathcal{C}$. *This idea is facing the challenge of designing this simulator as a RAS.* The problem turned to be difficult because of increased descriptional complexity, and we will see that the newly constructed RAS uses significantly more variables than $\mathcal{C}$, each component having significantly more states than the corresponding component in $\mathcal{C}$.

One essential technique used in the proof is to index the newly introduced variables in such manner, that the subscripts themselves provide information on where the run of $B$ has paused, or where it should resume from, in terms of the states of $B$.

In order to construct a RAS for the intersection, we need to consider the family of functions $f_w : Q_B \to Q_B$ defined as $f_w(q) = \delta_B(q, w)$. Since $Q_B$ is finite, the number of functions $\{f_w \mid w \in \Sigma^*\}$ is finite. These functions, together with composition and $f_\varepsilon$ as identity, form a finite monoid: the transition monoid $T_B$ of $B$.

Thus, we partition $\Sigma^*$ into equivalent classes, given by the equivalence of finite index: $u \equiv v \Leftrightarrow f_u = f_v$.

We denote $W = \Sigma^*/\equiv$, the quotient of $\Sigma^*$ under $\equiv$, and denote the functions in $T_B$ by $\{f_c\}_{c \in W}$.

Thus, if $w_1, w_2 \in c$, then for all $i \in Q_B$,

$$\delta_B(i, w_1) = j \text{ iff } \delta_B(i, w_2) = j. \tag{1}$$

Please, note that if there is an $i \in Q_B$ such that $\delta_B(i, w_1) = j$ and $\delta_B(i, w_2) = j$, we may have $w_1 \not\equiv w_2$. We can only say that $w_1$ and $w_2$ are equivalent, if equation 1 holds for all $i \in Q_B$. Therefore, the transitions from a state $i$ in $B$ can be precisely determined for each class $c \in W$.

For $i, j \in Q_B$, denote by $B_{i,j}$ the automaton obtained from $B$ by setting $i$ to be the initial state and $j$ the only final state. Then, we anticipate that beside the normal indexing of variables in $\mathcal{C}$, we need extra information for subscripts: one component to keep track of states from $Q_B$, and another component to keep track of the class of $W$ for each instantiation of a variable. The $Q_B$ index is used to keep track of the states in $B$, while the $W$ component is used to syncronize the match for $u_k$ with $v_k$.

We anticipate that we need some minimal information for states in order to keep track of what variable is initialized and what variable is not. Moreover, if a variable resulting from $u_k$ is initialized, it must be syncronized with all future $v_k$-corresponding variables; therefore, beside $k$ we need to know the behavior of a transition with a $v_k$-related variable, and when this transition can be triggered. Therefore, the name of a state having a transition with a $v_k$-related variable must contain enough information to distinguish a specific case of variable(s) instantiation; in other words, must reflect the behavior for the $Q_B$ index of such a transition.

Since all transitions labeled with a variable resulting from $u_k$ correspond to an automaton of the new RAS, we need to establish the initial state of this automaton. The $Q_B$ component of this initial state must be stored in the name of the $u$-variable.

We denote $Init(k) = \{s \in Q \mid \exists i : \delta_i(s, u_k) \text{ is defined}\}$. Since $\mathcal{C}$ is a RAS, there exists exactly one transition for each $k$; thus $Init(k)$ will always have one element only. We are interested in the set of states having transitions with variables $u_k$ $(1 \le k \le n-1)$: $Init = \bigcup_{k=1}^{n-1} Init(k)$. Note that we may have variables $u_k, u_{k'}$, with $k \neq k'$, but $Init(k) = Init(k')$.

Since in any RAS we are allowed to have only one transition with each $u$-variable, and we are also allowed to reinstantiate variables, we must have only one state changing the instantiation of an $u$-variable. Thus, we distinguish between states following the first instantiation of an $u$-variable and states following its reinstantiation. Moreover, we give different names to $u$-variables in order to ensure the unicity of such transitions. Hence, the variables for the newly defined RAS, $\mathcal{A}$, will be denoted by $u_{k,S}$ and $v_{k,S}$, with $k \in \{1, \ldots, n\}$, where $S$ is a set related to $W$ and defined as follows.

For $k \in \{1, \ldots, n\}$, let $S_k \in (Q_B W \{1,2\} \cup \{0\})^k$, $S_k = \{(i_k c_k m_k, \ldots, i_h c_h m_h, \ldots, i_1 c_1 m_1) \mid i_h \in Q_B \cup \{\varepsilon\}, c_h \in W \cup \{\varepsilon\}, m_h \in \{0,1,2\}$; if $u_h \not\preceq u_k$, then $i_h = c_h = \epsilon$ and $m_h = 0$; else, if $u_h \preceq u_k$ and $m_h = 0$, then $i_h = c_h = \varepsilon\}$. The intention behind $S_k$ is to name the states in such way that we do not allow two simultaneous instantiations of variable $u_{k,S}$, for different values of the $k$ component of $S$ in any computation of $\mathcal{A}$. The $m_h$ component will store the following information for a variable $u_h$: (a) if $m_h = 0$ then variable $u_h$ has not been instantiated yet; (b) if $m_h = 1$ then variable $u_h$ has been instantiated once; and (c) if $m_h = 2$ then variable $u_h$ has been reinstantiated.

In $\mathcal{C}$, the variable $v_k$ must use the last instantiation of $u_k$; therefore, we must also ensure that the transitions labeled with $v_{k,S}$ are preceded by $u_{k,S}$ in every computational path in $\mathcal{A}$ (Remark 1). Thus, we store the information of an $S \in S_k$ into the name of the states for all automata resulting from variable $u_k$. The projection $\pi_h : S_k \longrightarrow (Q_B W\{1,2\} \cup \{0\})$ is defined by $\pi_h(S) = i_h c_h m_h$, where $S = (i_k c_k m_k, \ldots, i_h c_h m_h, \ldots, i_1 c_1 m_1)$.

For $S \in S_k$, denote by $E(S,h) = S'$, where: (a) if $\pi_h(S) = 0$ then $S' = S$ and (b) if $\pi_h(S) \neq 0$ then $S'$ is such that $\pi_h(S') = 1$ and for all $h' \neq h$, $\pi_{h'}(S') = \pi_{h'}(S)$. The RAS $\mathcal{A}$, accepting the intersection $L(\mathcal{C}) \cap L(B)$ is constructed as follows:

1. for all $k$ and $S \in S_k$, such that $C_k$ does not have transitions labeled with variables (i.e., for which $S = (i_k c_k m_k, \underbrace{0, \ldots, 0}_{k-1})$):
   $A_{k,S} = \Big(Q_k \times Q_B, \Sigma, (0_1, i), \delta_{k,S}, F_{k,j}\Big)$, where $i = i_k$; $j = f_d(i)$ where $d = c_k$;
   for all $(p,l) \in Q_k \times Q_B$ and for all $a \in \Sigma$: $\delta_{k,S}\big((p,l), a\big) = \{(q,e) \mid q \in \delta_k(p,a), e = \delta_B(l,a)\}$ and $F_{k,j} = F_k \times \{j\}$;

   *This is the case when back-references are not processed, thus the construction is the usual automata Cartesian product [6],* $C_k \times B_{i,f_d(i)}$. *Notice that the definition of* $\delta_{k,S}$ *is independent of* $S$*, thus these automata share a same transition table, essentially. Moreover, in order to make the construction uniform we may consider each state in* $Q_k \times Q_B$ *as an element of* $Q_k \times Q_B \times S_{k-1}$*, where il all these cases* $S_{k-1} = \{(\underbrace{0, \ldots, 0}_{k-1})\}$ ;
2. for all $k \in \{2, \ldots, n-1\}$ and $S \in S_k$, denoting $i = i_k$, $d = c_k$, and $j = f_d(i)$, we have:

$$A_{k,S} = \Big(Q_k \times Q_B \times S_{k-1},\ \Sigma \cup \{u_{k',S'} \mid k' < k,\ S' \in S_{k-1}\} \\ \cup \{v_{k',S'} \mid k' < k,\ S' \in S_{k-1}\},\ (0_k, i, \underbrace{0, \ldots, 0}_{k-1}),\ \delta_{k,S},\ F_{k,j}\Big),$$

   where $F_{k,j} = F_k \times \{j\}$, and
   - for all $(p, l, S') \in Q_k \times Q_B \times S_{k-1}$, $a \in \Sigma$:

$$\delta_{k,S}\big((p,l,S'), a\big) = \{(q, j', S') \mid q \in \delta_k(p,a), j' = \delta_B(l,a),\ q \notin Init\} \\ \cup \{(q, j', T) \mid q \in \delta_k(p,a), j' = \delta_B(l,a),\ T \in E(S',h), q \in Init(h)\}$$

   - for all $(p, i', S') \in Q_k \times Q_B \times S_{k-1}$, $k' < k$, $d \in W$, $\pi_{k'}(S') = i_{k'} c_{k'} m_{k'}$, $j' = f_d(i')$, and $T' \in S_{k-1}$ such that $\pi_h(S') = \pi_h(T')$ for all $h \neq k'$ and $\pi_{k'}(T') = i' d \max\{2, m_{k'} + 1\}$:

$$\delta_{k,S}\big((p, i', S'), u_{k',T'}\big) = \{(q, j', T') \mid q \in \delta_k(p, u_{k'}) - Init\} \\ \cup \{(q, j', T) \mid q \in \delta_k(p, u_{k'}) \cap Init(h),\ T \in E(T', h)\}$$

- for all $(p, i'', S') \in Q_k \times Q_B \times S_{k-1}$, $k' < k$, $\pi_{k'}(S') = i'dm_{k'}$, $m_{k'} > 0$, and $j' = f_d(i'')$:

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

*Notice that after each transition triggered by $u_{k',S'}$ we reach states where the transition with $v_{k',S'}$ are possible, but transitions with $v_{k',S''}$, $\pi_{k'}(S') \neq \pi_{k'}(S'')$ are not defined, since $v_{k',S''}$ is a match for $\emptyset$. This ensures a synchronization between $u_{k',S'}$ and $v_{k',S'}$, which mimics the synchronization between $u_{k'}$ and $v_{k'}$ in $C_k$.*

3. $A_{n,0,F_B} = \Big(Q_n \times Q_B \times S_{n-1},\ \Sigma \cup \{u_{k',S'} \mid k' < n,\ S' \in S_{n-1}\} \cup \{v_{k',S'} \mid k' < n,\ S' \in S_{n-1}\},\ (0_n, \underbrace{0, \ldots, 0}_{n-1}),\ \delta_{n,0,F_B},\ F_{n,F_B}\Big)$,
   where $F_{n,F_B} = F_n \times F_B$, and
   - for all $(p, l, S') \in Q_k \times Q_B \times S_{n-1}$, $a \in \Sigma$:

$$\begin{aligned}\delta_{n,0,F_B}\big((p, l, S'), a\big) = &\{(q, j', S') \mid q \in \delta_n(p, a), j' = \delta_B(l, a),\ q \notin Init\} \\ &\cup \{(q, j', T) \mid q \in \delta_n(p, a), j' = \delta_B(l, a),\ T \in E(S', h), q \in Init(h)\}\end{aligned}$$

   - for all $(p, i', S') \in Q_n \times Q_B \times S_{n-1}$, $k' < n$, $d \in W$, $\pi_{k'}(S') = i_{k'}c_{k'}m_{k'}$, $j' = f_d(i')$, and $T' \in S_{n-1}$ such that $\pi_h(S') = \pi_h(T')$ for all $h \neq k'$ and $\pi_{k'}(T') = i'd\max\{2, m_{k'} + 1\}$:

$$\begin{aligned}\delta_{n,0,F_B}\big((p, i', S'), u_{k',T'}\big) = &\{(q, j', T') \mid q \in \delta_n(p, u_{k'}) - Init\} \\ &\cup \{(q, j', T) \mid q \in \delta_n(p, u_{k'}) \cap Init(h),\ T \in E(T', h)\}\end{aligned}$$

   - for all $(p, i'', S') \in Q_n \times Q_B \times S_{n-1}$, $k' < n$, $\pi_{k'}(S') = i'dm_{k'}$, $m_{k'} > 0$, and $j' = f_d(i'')$:

$$\begin{aligned}\delta_{n,0,F_B}\big((p, i'', S'), v_{k',S'}\big) = &\{(q, j', T') \mid q \in \delta_n(p, v_{k'}) - Init\} \\ &\cup \{(q, j', T) \mid q \in \delta_n(p, v_{k'}) \cap Init(h),\ T \in E(T', h)\}.\end{aligned}$$

Considering that $A_{n,0,F_B}$ 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 which justify the correctness of our construction:

1. $\mathcal{A}$ is indeed a RAS. The transitions with $u_{k',S'}$ are only possible for states $(p, i, S'')$ in $Q_k \times Q_B \times S_{k-1}$ with $\pi_h(S') = \pi_h(S'')$ for all $h \neq k'$, and $\pi_{k'}(S'') = 1$ or $\pi_{k'}(S'') = 0$. Notice that if $\pi_{k'}(S') > 0$, then $\pi_{k'}(S'') = 1$ (a reinstantiation); whereas if $\pi_{k'}(S') = 0$, then $\pi_{k'}(S'') = 0$ (the first instantiation). This ensures the uniqueness of the transition labeled with $u_{k',S'}$ in $A_{k,S}$.

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 same $k$, the buffers $x_{k,S}$ in $A_{k,S}$ are not used simultaneously; therefore, it does not make any difference whether for each $k$ we use one buffer or $\#S_k$ 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 since we use a RAS obtained from a regex in free-star normal form. This guarantees that each time we re-instantiate $u_{k',S'}$ we update the projection $k'$ of $S'$; subsequently, all the other variables $u_{k',S''}$ with $\pi'_k(S') \neq \pi_k(S'')$ are either a match for $\emptyset$ or 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 parantheses). Thus, only the transitions with $v_{k',S'}$ are possible. The syncronization is done using the $k'$ projection of the index $S'$.

3. For every transition

$$((s,i,S),\alpha w, S^{(t)}, x_1^{(t)}, x_2^{(t)}, \ldots, x_H^{(t)}) \mapsto_{\mathcal{A}} ((q,j,T), w, S^{(t+1)}, x_1^{(t+1)}, x_2^{(t+1)}, \ldots, x_H^{(t+1)})$$

we have that $i = j$ and $\alpha = \varepsilon$, or $\delta_B(i,\alpha) = j$, (we choose $B$ to be a DFA).

In conclusion, for every transition

$$((0_n, 0_B, \underbrace{0,\ldots,0}_{n-1}), w, \emptyset, \underbrace{\emptyset,\emptyset,\ldots,\emptyset}_{H}) \mapsto^*_{\mathcal{A}} ((q,q'), \varepsilon, \emptyset, x_1, x_2, \ldots, x_H)$$

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

$$(0, \alpha w, S^{(t)}, x_1^{(t)}, x_2^{(t)}, \ldots, x_n^{(t)}) \mapsto_{\mathcal{C}} (q, w, S^{(t+1)}, x_1^{(t+1)}, x_2^{(t+1)}, \ldots, x_n^{(t+1)}),$$

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. □

The above construction involves a RAS and a DFA. If we want to recover a regex from the resulting RAS $\mathcal{A}$, we notice that there is an "or" between parenthesis $u_{k,S}$ and $u_{k,S'}$; however, this "or" extends to the last back-reference $v_{k,S}$. The parenthesis represented by variable $u_{k,S}$ is the regex for the intersection of $L(B_{i,j})$ with the language matched by the parenthesis $u_k$, where $j = f_c(i)$. The same idea applies to back-references.

## 5 Consequences and Conclusion

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

$$L_1 = \{(aababb)^n(bbabaa)^n \mid n \geq 0\},\ \ L_2 = \{a^nb^n \mid n \geq 0\},$$

$$L_3 = \{a^{2n}b^n \mid n \geq 0\},\ \ L_4 = \{a^nb^nc^n \big| n \geq 0\},$$

$$L_5 = \{\{a,b\}^nc^n \mid n \geq 0\},\ \ L_6 = \{\{a,b\}^nc\{a,b\}^n \mid n \geq 0\}.$$

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 2.** *The following languages are not regex languages:*

$$L_7 = \{ww^R \mid \ \ w \in \Sigma^*\},\ \ L_8 = \big\{w \mid \ \ w = w^R\big\},\ \ L_9 = \big\{w \mid \ \ |w|_a = |w|_b\big\},$$

$$L_{10} = \big\{w \mid \ \ |w|_b = 2|w|_a\big\},\ \ L_{11} = \big\{w \mid \ \ |w|_a = |w|_b = |w|_c\big\},$$

$$L_{12} = \big\{w \mid \ \ |w|_a + |w|_b = |w|_c\big\},\ \ L_{13} = \big\{ucv \mid \ \ |u|_a + |u|_b = |v|_a + |v|_b\big\}.$$

*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 $L_7, \ldots, L_{13}$ was a regex language, so would be its corresponding intersection, leading to a contradiction.

We should mention that none of the languages $L_7, \ldots, L_{13}$ could be proven to be non-regex by pumping lemma alone. We should also mention a theoretical application of the closure property, that some previous results involving elaborate proofs, such as Lemma 3 in [2], are immediately rendered true by Theorem 3.

To conclude, in this paper we have defined a machine counterpart of regex, namely 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 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.

## References

1. A. V. Aho: *Algorithms for Finding Patterns in Strings.* In: Jan van Leeuwen (edt.), *Handbook of Theoretical Computer Science*, Vol.A: Algorithms and Complexity, Elsevier and MIT Press (1990) 255–300.
2. C. Câmpeanu, K. Salomaa and S. Yu: A Formal Study of Practical Regular Expressions. *IJFCS*, **14(6)** (2003) 1007–1018.

3. C. Câmpeanu and 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 and S. Yu: Pattern Expressions and Pattern Automata. *IPL*, **92** (2004) 267–274.
5. J.E.F. Friedl: *Mastering Regular Expressions*, O'Reilly & Associates, Inc., Cambridge, (1997).
6. J.E. Hopcroft, R. Motwani, and J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading Mass, (2006).
7. M.E. Lesk: Lex - a Lexical Analyzer Generator.*Computer Science Technical Report*, AT&T Bell Laboratories, Murray Hill, N.J, **39** (1975).
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 and G. Rozenberg (eds.), *Handbook of Formal Languages*, Springer Verlag (1997) 41–110.