On the Semantics of Atomic Subgroups in Practical Regular Expressions

Martin Berglund[1,3], Brink van der Merwe[2(✉)], Bruce Watson[1,3], and Nicolaas Weideman[2,3]

[1] Department of Information Science,
Stellenbosch University, Stellenbosch, South Africa
[2] Department of Computer Science,
Stellenbosch University, Stellenbosch, South Africa
abvdm@cs.sun.ac.za
[3] Center for AI Research, CSIR, Stellenbosch University, Stellenbosch, South Africa

Abstract. Most regular expression matching engines have operators and features to enhance the succinctness of classical regular expressions, such as interval quantifiers and regular lookahead. In addition, matching engines in for example Perl, Java, Ruby and .NET, also provide operators, such as atomic operators, that constrain the backtracking behavior of the engine. The most common use is to prevent needless backtracking, but the operators will often also change the language accepted. As such it is essential to develop a theoretical sound basis for the matching semantics of regular expressions with atomic operators. We here establish that atomic operators preserve regularity, but are exponentially more succinct for some languages. Further we investigate the state complexity of deterministic and non-deterministic finite automata accepting the language corresponding to a regular expression with atomic operators, and show that emptiness testing is PSPACE-complete.

1 Introduction

In this paper we study atomic subgroups, a generalization of the feature described by Jeffrey Friedl, in the first edition of his book on regular expressions, as follows [Fri97]:

> *"A feature I think would be useful, but that no regex flavor that I know of has, is what I would call possessive quantifiers. They would act like normal quantifiers except that once they made a decision that met with local success, they would never backtrack to try the other option. The text they match could be unmatched if their enclosing subexpression was unmatched, but they would never give up matched text of their own volition, even in deference to an overall match."*

In the five and a half years between the first and second edition of Friedl's book, possessive quantifiers were introduced, and in the process gave way to

A. Carayol and C. Nicaud (Eds.): CIAA 2017, LNCS 10329, pp. 14–26, 2017.
DOI: 10.1007/978-3-319-60134-2_2

atomic subgroups, making the prior a syntactic sugar for the latter. For example, `E*+` denotes a regular expression `E` with a possessive Kleene star applied, which may also be written as `(?>E*)`, where `?>` makes the surrounding parenthesis an "atomic subgroup". Atomic subgroups "lock up" the part of the pattern it contains once it has matched, a failure further on in the pattern is not allowed to backtrack into the atomic group, but backtracking past it to previous subexpressions works as usual. A common use of atomic subgroups is to prevent needless backtracking and thus speedup matching time. For example, while the matcher in Java will take exponential time in the length of the input string to establish that input strings of the form $a \ldots ab$ can not be matched by `(a|a)*`, by essentially trying each possible way of matching an a in the input string with respectively the first or the second a in `(a|a)`, matching happens in linear time when using `(?>a|a)*`, since the matcher "forgets" each time after using the first `a` in `(?>a|a)` to match an a, that it was also possible to use the second. Atomic subgroups are implemented in, among others, the Java, .NET, Python, Perl, PHP, and Ruby standard libraries, and in libraries such as Boost and PCRE.

Paper outline. In the next section we introduce the required notation followed by a section on the matching semantics of a-regexes (regular expressions with atomic subgroups). Then a section on the descriptional complexity of a-regexes and the complexity of deciding emptiness follows. After this, we briefly discuss how we arrived at our matching semantics definition, followed by our conclusions.

2 Definitions and Notation

An alphabet is a finite set of symbols. When not otherwise specified, Σ denotes an arbitrary alphabet. A regular expression over Σ is, as usual, an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ (ε denotes the empty string), or an expression of one of the forms $(E_1 | E_2)$, $(E_1 \cdot E_2)$, or (E_1^*), where E_1 and E_2 are regular expressions. Some parenthesis may be elided using the rule that the Kleene closure '$*$' takes precedence over concatenation '$\cdot$', which takes precedence over union '$|$'. In addition, outermost parenthesis may be dropped and $E_1 \cdot E_2$ abbreviated as E_1E_2. The language matched by an expression is defined in the usual way. Furthermore, an alphabet $S = \{s_1, \ldots, s_n\}$ used as an expression S is an abbreviation for $s_1 | \cdots | s_n$, and for any expression E we may write E^k as an abbreviation for $E \cdots E$, i.e. k copies of E (so $|E^k| = k|E|$, where $|E|$ denotes the number of symbols in $|E|$, i.e. the *size* of E). Regular expressions set in `typewriter` font are examples of the Java syntax (same as most other libraries), which is not fully described here.

For a set S let 2^S denote the powerset of S. For a string w and a set of strings S, let $w \setminus S = \{v \mid wv \in S\}$. A singleton set S and the single string may be used interchangeably. The union, concatenation and Kleene star of languages (over an alphabet Σ) is defined as usual. For a possibly infinite sequence $v_1, v_2, \ldots$ let $\mathrm{dedup}(v_1, v_2, \ldots)$ denote the list (always finite in the uses in this paper) resulting when only the first instance of each value in the sequence is retained (e.g. $\mathrm{dedup}(1, 2, 2, 1, 4, 3, 4) = 1, 2, 4, 3$). The concatenation of two sequences

$\sigma = v_1, \ldots, v_m$ and $\sigma' = v'_1, \ldots, v'_n$ is denoted by σ, σ' and defined to be the sequence $v_1, \ldots, v_m, v'_1, \ldots, v'_n$. For a string $w \in \Sigma^*$ and sequence $\sigma = v_1, \ldots, v_m$ with $v_i \in \Sigma^*$, we denote by $w\sigma$ the sequence $wv_1, \ldots, wv_m$.

Remark 1. In many real-world systems the primary primitive for regular expression matching is a *substring finding* one, where an input string w is searched for the left-most longest substring which matches the expression. Here we take (mostly) the more classical view, concerning ourselves with the strings matched entirely by the expression (with the exception of Definition 4). When we write e.g. a^*b the corresponding Java regular expression is `^a*b$`, the caret and dollar sign being special operators which "anchor" the match to the ends of the string.

As usual we will need to consider finite automata in some of the following.

Definition 1. *A* non-deterministic finite automaton *(NFA) is a tuple* $A = (Q, \Sigma, q_0, \delta, F)$ *where: (i)* Q *is the finite set of* states; *(ii)* Σ *is the* input alphabet; *(iii)* $q_0 \in Q$ *is the* initial state; *(iv)* $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ *is the* transition relation; and *(v)* $F \subseteq Q$ *is the set of* final states.

The language $\mathcal{L}(A)$ *accepted by* A *is precisely the strings* $w = \alpha_1 \cdots \alpha_n$ *where* $\alpha_i \in \Sigma \cup \{\varepsilon\}$ *for all* i*, such that there exists states* $q_0, \ldots, q_n \in Q$*, where* q_0 *is the initial state,* $(q_i, \alpha_{i+1}, q_{i+1}) \in \delta$ *for each* i*, and* $q_n \in F$*.*

For brevity we may write e.q. A_Q to denote the states of A, A_δ for the transition function, and so on. Also, $|A|$ denotes the number of states in A.

Definition 2. *An* NFA with negative regular lookaheads *(NFA with lookaheads for short) is an NFA* $A = (Q, \Sigma, q_0, \delta, F)$*, where* δ *may contain transitions of the form* $(q, \alpha, \neg E, q') \in \delta$*, where* E *is a regular expression over* Σ*.*

The language is as in Definition 1 except a transition $(q, \alpha, \neg E, q') \in \delta$ *may only be used when the* remainder *of the input string is not in* $\mathcal{L}(E)$*.*

We use lookaheads to demonstrate the regularity of atomic subgroups in an intuitive way. For this purpose, note that NFA with lookahead can only represent regular languages, as the lookaheads may be implemented by complementation and intersection of regular languages (that is, a product automaton tracking all lookaheads in parallel with the main expression).

3 Regular Expression Semantics and Atomic Subgroups

Informally, atomic subgroups are defined in terms of the depth-first search nature of matchers (such as in e.g. Java), in that the implementation will discard the portion of the stack (recording decisions made) corresponding to the atomic subgroup upon exiting the group. That is, the matcher will not reconsider choices made within the atomic subgroup once it has started to match the expression immediately following the group, though it may reconsider the choices made before entering the atomic subgroup, in which case the atomic subgroup matching will also be reconsidered.

Definition 3. *An* a-regex *over Σ is an element of $\Sigma \cup \{\varepsilon, \emptyset\}$, or an expression of the form $(E_1 \,|\, E_2)$, $(E_1 \cdot E_2)$, (E_1^*), or $(\triangleright E_1)$, where E_1 and E_2 are a-regexes. A subexpression of the form $(\triangleright E)$, for an expression E, is referred to as an* atomic subgroup *(that is, where it is styled as `(?>E)` in e.g. Java we write $(\triangleright E)$).*

Before going into the definition proper let us first give some informal examples of the semantics of atomic subgroups (agreeing with those in practical software).

Example 1. The expression $(\triangleright b^*)b$ matches nothing, as the atomic subgroup will consume all bs available and refuse to give one up for the final b subexpression. Meanwhile, the expression $a^*(\triangleright ab \,|\, b^*)b$ will match $\{a^n b^2 \mid n \geq 1\}$. For example, on a^2b^2 the matcher will first have the $a*$ subexpression consume all as, then the $b*$ in the atomic subgroup "steals" all bs, making the match fail. However, as the atomic subgroup will not relinquish a b the matcher will backtrack past it into a^*, having it match one less a, after which reconsidering the atomic subgroup instead matches its preferred ab, leaving the final b to be matched by the end of the expression. Note that there exist E such that $\mathcal{L}(\triangleright E) \neq \mathcal{L}(E)$, and more precisely, $\mathcal{L}(\triangleright E) \subseteq \mathcal{L}(E)$ in general. For example $(\triangleright a \,|\, aa)$ does not match aa as it will always prefer to just match the first a without possibility of backtracking.

Example 2. A key use of atomic subgroups in practical matching is to limit ambiguity for performance reasons (e.g. avoiding pitfalls such as those formalized in [WvdMBW16]). Consider the following expression for matching email addresses, extracted from the RegExLib repository [Reg] (here slightly simplified):

```
[0-9a-z]([-.\w]*[0-9a-z])*@(([0-9a-z])+([-\w]*[0-9a-z])*\.)+[a-z]{2,9}
```

We do not give a complete explanation of the syntax and matching behavior of this expression, but there are two dangerous subexpressions here. Firstly, `([-.\w]*[0-9a-z])*` is (exponentially) ambiguous on the string $a \cdots a$ since both `[-.\w]` and `[0-9a-z]` represents subalphabets containing a, and thus aa can be matched in more than one way by `([-.\w]*[0-9a-z])*`. Using this regular expression, in e.g. a Java system, to validate that a user has provided a valid email address, would leave the system open to a regular expression denial of service attack. To make it safe one would replace this subexpression by `(?>([-.\w]*[0-9a-z])*)`. The refusal to backtrack, introduced by using `?>`, will have no effect on the language accepted, as the next symbol in the input sting must be an `@`, and the subgroup cannot read `@`. A similar problem, and solution, exist for the subexpression `([-\w]*[0-9a-z])*)`. This kind of performance concern apply especially in expressions using back references, which are necessarily very expensive to match in the face of ambiguity (unless P equals NP [Aho90]).

Example 3. The example eliciting the quote from [Fri97] on the introductory page concerned writing a regular expression for rounding decimal numbers. The expression should match a decimal number if it; either has more than two digits on the right of the decimal point; and; if the third is non-zero, it has more than three. It would match 12.750001 with the intent of rounding to 12.75, and match 2.1250 to round to 2.125, but not match 2.125 (in almost all practical regular expression matchers the substring matched by a certain subexpression can be extracted after matching, which is used in this example). Friedl suggests the expression `([1-9][0-9]*\.([0-9][0-9]([1-9]|ε)))[0-9][0-9]*`, where `[x-z]` is shorthand for $x \,|\, \cdots \,|\, z$, with the intent of using the first parenthesized subexpression (i.e. `([1-9] [0-9]*\.([0-9] [0-9]([1-9]|ε)))`) to "capture" the rounded number. This is incorrect however, as the number 2.125 would get 2.12 captured with the 5 being used to satisfy the final `[0-9]`. It is non-trivial to rewrite without interfering with having the rounded substring be the one matched by the first subexpression. This suggested the invention of atomic subgroups, i.e. the ability to force the first subexpression to not not give up the trailing 5 once it has matched it in for example 2.125, even though this makes the overall match fail, realizing the intended language.

For classical regular expressions the language being accepted can be defined inductively in terms of operations on the languages accepted by the subexpressions, e.g. $\mathcal{L}(E_1 \cdot E_2) = \{wv \mid w \in \mathcal{L}(E_1), v \in \mathcal{L}(E_2)\}$, but this is not the case for a-regexes. Instead we have to opt for a left-to-right approach on a specified input string w, where a subexpression acts upon some prefix of the suffix of w left to be matched. This definition was arrived at by careful analysis of the Java implementation – see Sect. 5 for a discussion on this process.

Definition 4. *For any a-regex E and string w let $m(E, w)$ denote the sequence of (not necessarily strict) prefixes of w which E matches,* in order of priority. *Then for all w:*

- *$m(\varepsilon, w) = \varepsilon$, the list consisting of a single element, the empty string,*
- *$m(\alpha, w) = \alpha$ if $\alpha \in \Sigma$ and w starts with α, otherwise $m(\alpha, w)$ is empty,*
- *$m(E \,|\, E', w) = dedup(m(E, w), m(E', w))$ (the concatenation deduplicated),*
- *$m(E^*, w) = dedup(v_1\sigma_1, v_2\sigma_2, \ldots, v_n\sigma_n, \varepsilon)$ where $m(E, w) = v_1, \ldots, v_n$ and for each i, $\sigma_i = m(E^*, v_i \setminus w)$ if $v_i \neq \varepsilon$, and $\sigma_i = \varepsilon$ otherwise,*
- *$m(E \cdot E', w) = dedup(v_1 m(E', v_1 \setminus w), \ldots, v_n m(E', v_n \setminus w))$ where $m(E, w) = v_1, \ldots, v_n$,*
- *$m((\triangleright E), w) = v_1$ if $m(E, w)$ is non-empty and equal to $v_1, \ldots, v_n$, otherwise $m((\triangleright E), w)$ is empty.*

The language matched by E is denoted $\mathcal{L}_a(E)$ and defined as

$$\{w \mid w \in \Sigma^* \text{ occurs in } m(E, w)\}.$$

Remark 2. Note that setting $\sigma_i = \varepsilon$ when $v_i = \varepsilon$ in the definition of $m(E^*, w)$ above, is required in order to avoid infinite recursion in the definition. Regular expressions with subexpressions of the form E^*, such that $\varepsilon \in \mathcal{L}(E)$, are so-called *problematic regular expressions.* These are special enough that they are a source for differences in matching behavior in some implementations, and are considered in for example [SMV12, BvdM16].

Remark 3. We can define $m(E^{*?}, w)$, where $E^{*?}$ denotes the lazy Kleene star of E by moving the ε to the front in a definition otherwise similar to $m(E^*, w)$: $m(E^{*?}, w) = \text{dedup}(\varepsilon, v_1\sigma_1, v_2\sigma_2, \ldots, v_n\sigma_n)$. Intuitively, $E^{*?}$ repeats matching with E as few times as possible, whereas E^* does the opposite. Thus $m(E^{*?}, a^n) = \{\varepsilon, a, \ldots, a^n\}$ whereas $m(E^*, a^n) = \{a^n, a^{n-1}, \ldots, \varepsilon\}$.

Remark 4. Atomic subgroups may be compared to *cuts* [BBD+13], a proposed alternative to concatenation, denoted $R_1\,!\,R_2$ for expressions R_1 and R_2. The expression $R_1\,!\,R_2$ forces a "greedy" (i.e. longest possible prefix) match with R_1, whereas $(\triangleright R_1)R_2$ forces R_1 to pick the "first" match according to a priority implied by the syntactic details of the expression R_1. So, for example, whereas $\mathcal{L}((\triangleright\varepsilon\,|\,a)ab^*)$ equals ab^*, the cut expression $(\varepsilon\,|\,a)!ab^*$ would match aab^*. As such the cut is a normal operator on languages with two arguments, whereas atomic subgroups depend on the structure of the expressions.

Lemma 1. *For a regular expression E the sequence $m(E, w)$ contains each prefix w' of w with $w' \in \mathcal{L}(E)$ precisely once, and $m(E, w)$ contains no other strings. As a direct effect it holds that $\mathcal{L}(E) = \mathcal{L}_a(E)$.*

Proof. Follows by induction on the number of operators appearing in E. □

In the remainder of the paper we simply use the notation $\mathcal{L}(E)$, instead of $\mathcal{L}_a(E)$, for an a-regex E. Let us consider some of the properties of a-regexes.

Lemma 2. *For a-regexes E and F we have the following properties.*

(i) $\mathcal{L}(EF) \subseteq \mathcal{L}(E)\mathcal{L}(F)$ (ii) $\mathcal{L}(E\,|\,F) = \mathcal{L}(E) \cup \mathcal{L}(F) = \mathcal{L}(F\,|\,E)$

(iii) $\mathcal{L}(E^*) \subseteq \mathcal{L}(E)^*$ (iv) $\mathcal{L}(\triangleright E) \subseteq \mathcal{L}(E)$

Also (i) is an equality if E is a regular expression. In addition, there exists E and F such that $\mathcal{L}(EF) \subsetneq \mathcal{L}(E) \cdot \mathcal{L}(F)$, $\mathcal{L}(E^*) \subsetneq \mathcal{L}(E)^*$ and $\mathcal{L}(\triangleright E) \subsetneq \mathcal{L}(E)$.

Proof. Follows from Definition 4, e.g. $abab \notin \mathcal{L}((\triangleright aba^*)^*) \subsetneq \mathcal{L}((\triangleright aba^*))^* \ni abab$ exemplifies property (iii). □

In addition to the language captured, let us make the ordered nature of the semantics which Definition 4 gives to each expression an explicit property.

Definition 5. *For a-regexes F and G, we define F and G to be* language equivalent*, denoted by $F \equiv_{\mathcal{L}} G$, if $\mathcal{L}(F) = \mathcal{L}(G)$, whereas F and G are* order equivalent*, denoted by $F \equiv_{\mathcal{O}} G$, if $m(F, w) = m(G, w)$ for all $w \in \Sigma^*$.*

Lemma 3. *The following language and order equivalences hold.*

(i) $(FG)H \equiv_{\mathcal{O}} F(GH)$ (ii) $(F\,|\,G)\,|\,H \equiv_{\mathcal{O}} F\,|(G\,|\,H)$

(iii) $(F^*)^* \equiv_{\mathcal{O}} F^*$ (iv) $F \equiv_{\mathcal{O}} G$ *implies* $F \equiv_{\mathcal{L}} G$

However, there exists some F and G fulfilling each of the following inequalities.

(v) $F\,|\,G \not\equiv_{\mathcal{O}} G\,|\,F$ (vi) $(\triangleright FG) \not\equiv_{\mathcal{L}} (\triangleright F)(\triangleright G)$ (vii) $F \equiv_{\mathcal{L}} G$ *but* $F \not\equiv_{\mathcal{O}} G$

Proof. Follows directly from Definition 4. For $(\triangleright FG) \not\equiv_{\mathcal{O}} (\triangleright F)(\triangleright G)$ take e.g. $F = \Sigma^*$ and $G = a$, which makes $\mathcal{L}((\triangleright F)(\triangleright G))$ empty. □

Order equivalence captures the semantics precisely: if two expressions are *not* order-equivalent contexts exist where replacing one with the other (as subexpressions of some expression) will result in different languages being accepted.

Lemma 4. *Let F and G be two a-regexes over Σ. Let E and E' be a-regexes over $\Sigma \cup \{\#\}$ (we assume $\# \notin \Sigma$) such that E' is obtained from E by replacing the subexpression F by the subexpression G. Then: (i) $F \equiv_{\mathcal{O}} G$ implies $E \equiv_{\mathcal{O}} E'$ for all E; and; (ii) $F \not\equiv_{\mathcal{O}} G$ implies $E \not\equiv_{\mathcal{L}} E'$ for* some *E.*

Proof. Statement (i) follows from Definitions 4 and 5, since having order equivalence means that $m(F, w) = m(G, w)$ for all w, and the sequences $m(F, w)$ and $m(G, w)$ entirely determine the influence of the subexpressions F and G on $m(E, w)$ and $m(E', w)$, respectively.

For statement (ii), take w such that $m(F, w) \neq m(G, w)$ and let $m(F, w) = v_1 \cdots v_n$ and $m(G, w) = v'_1 \cdots v'_{n'}$. If $\{v_1, \ldots, v_n\} \neq \{v'_1, \ldots, v'_{n'}\}$ the languages $\mathcal{L}(F)$ and $\mathcal{L}(G)$ already differ when restricted to prefixes of w, so just take $E = F$, $E' = G$ and we are done. Otherwise, let i be the smallest index with $v_i \neq v'_i$. As v_i and v'_i are both prefixes of w, we may assume without loss of generality that $w = v_i w_2 = v'_i w_1 w_2$, with $w_1 \neq \varepsilon$. Now construct $E = (\triangleright F(w_2\#\# \mid w_1 w_2\#))\#$, which makes $E' = (\triangleright G(w_2\#\# \mid w_1 w_2\#))\#$. Then $w\#\# \notin \mathcal{L}(E)$, while $w\#\# \in \mathcal{L}(E')$. To see, for example, why $w\#\# \notin \mathcal{L}(E)$, note that as a subexpression, F has to match either v_i or v'_i in order to make it through the atomic subgroups in E, when attempting to match $w\#\#$ with E. However, during this matching process, the a-regex F will in fact use v_i and not v'_i, since v_i appears before v'_i in $m(E, w)$. Since using v_i will cause both $\#$ end-markers to be used in the atomic subgroup, we have that $w\#\#$ is not matched by E. □

4 Automata Construction and Complexity Results

Despite the rather special semantics, adding atomic subgroups to regular expressions *does* in fact preserve regularity, though with a high state complexity.

Lemma 5. *For every a-regex E there exists a finite automaton A with $\mathcal{L}(E) = \mathcal{L}(A)$.*

Proof. We first consider the case where E contains no subexpression of the form F^*, with $\varepsilon \in \mathcal{L}(F)$. This restriction ensures that the constructed NFA contain no ε-loops, and thus each input string has only finitely many acceptance paths.

We inductively construct an NFA for each a-regex E, denoted by $M(E)$, with lookaheads and prioritized ε-transitions (a concept to be defined below), such that not only $\mathcal{L}(M(E)) = \mathcal{L}(E)$, but also such that $M(E)$ encodes (to be made precise below) for each string w the order in which prefixes w' of w with $w' \in \mathcal{L}(E)$, appear in $m(E, w)$. $M(E)$ has a single accept state with no outgoing transitions. With the exception of the final state, each state p in $M(E)$ has outgoing transitions of one of the following forms: (i) p has a single transition to a state q on a symbol from $\Sigma \cup \{\varepsilon\}$; (ii) p has transitions on ε to states q_1 and q_2, but $p \to q_1$ has higher priority (a concept used and defined next to ensure that each $w \in \mathcal{L}(E)$ has a unique accepting path in $M(E)$) than $p \to q_2$. Also, prioritized ε-transitions may have regular lookahead.

Given a string $w \in \mathcal{L}(G)$, we define an accepting path for w as usual, but whenever we encounter a state with transitions of type (ii), we always pick the higher priority transition if taking this transition will still make acceptance of w possible. By doing this, each $w \in \mathcal{L}(E)$ will have a unique accepting path in $M(E)$. Note that in terms of language accepted, the priorities on transitions play no role. Also, note that if w' and w'' are both prefixes of a string w, with $w', w'' \in \mathcal{L}(E)$, then the accepting paths $ap(w')$ and $ap(w'')$ of w' and w'' respectively, will be such that at some state p with prioritized outgoing ε-transitions, the one acceptance path will take the higher priority transition and the other the lower priority transition. The priorities on transitions at states with two outgoing ε-transitions can thus be used to define an ordering on all prefixes of w in $\mathcal{L}(E)$, denoted by the sequence $M(E, w)$. By constructing $M(E)$ inductively over the operators in E, we show that $M(E, w) = m(E, w)$ for all $w \in \Sigma^*$, which will also imply that we have $\mathcal{L}(M(E)) = \mathcal{L}(E)$. See Fig. 1 for examples.

The construction of $M(E)$, when $E = \emptyset$, ε or a, for $a \in \Sigma$, is as usual. Now suppose $M(E_i)$, for $i = 1, 2$, is already constructed, and $M(E_i, w) = m(E_i, w)$ for all $w \in \Sigma^*$. Also, assume p_i and q_i are the initial and final states in E_i. Next we describe the construction of (i) $M(E_1|E_2)$, (ii) $M(E_1E_2)$, (iii) $M({E_1}^*)$ and (iv) $M((\triangleright E_1))$, and leave it to the reader to verify from Definition 4 that $M(E, w) = m(E, w)$, for all $w \in \Sigma^*$, in each of these four cases.

(i) Create a new initial state p and final state q. In addition to these two states, we use the states and transitions as in E_1 and E_2. We add prioritized ε-transitions from p to p_1 and p_2, with $p \to p_1$ having higher priority. We also add ε transitions from q_1 and q_2 to q.

(ii) We use the states and transitions as in E_1 and E_2 and merge states q_1 and p_2. We use p_1 as initial and q_2 as final state.

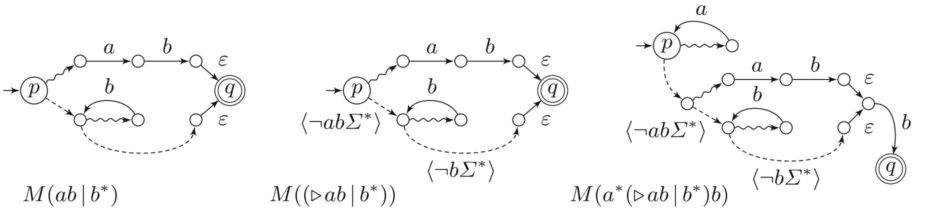

Fig. 1. NFA with lookahead constructed as defined in Lemma 5. Wavy and dashed lines represent high and low priority transitions respectively. Negative lookaheads are shown in angle brackets. On the left the expression $ab \,|\, b^*$, in the middle the same expression inside an atomic subgroup, getting the prioritized edges augmented by lookaheads on the low-priority case. Right is the full result for an expression discussed in Example 1.

(iii) Create a new final state q and relabel the old final state q_1 in E_1 as the new initial state. In addition to the state q, we use the states and transitions as in E_1. We add prioritized ε-transitions from q_1 to p_1 and q, with $q_1 \to p_1$ having higher priority.

(iv) We keep the states and transitions as in E_1, but for all states p' having prioritized ε-transitions to q'_1 and q'_2 (with $p' \to q'_1$ having highest priority), we add regular lookahead $\neg(E_1)_{p',q'_2}$ to $p' \to q'_2$, where $(E_1)_{p',q'_2}$ is obtained as follows. Let $(E_1)_{q'_1}$ be a regular expression for the language accepted by $M(E_1)$ when q'_1 is initial, then $(E_1)_{p',q'_2} = (E_1)_{q'_1}\Sigma^*$.

Next we discuss the modifications required for subexpression of the form ${E_1}^*$, with $\varepsilon \in \mathcal{L}(E_1)$. In the construction of ${E_1}^*$ given in (iii) above we end up with potentially infinitely many acceptance paths for some strings when $\varepsilon \in \mathcal{L}(E_1)$. This problem can be addressed by a procedure called *flattening*, described in the proof of Theorem 3 in [BvdM16]. According to Definition 4, in cases where ε is the next prefix that should be matched (by the subexpression E_1) based on priority of prefix matching, the process of matching with E_1 (again) is disallowed. Flattening ensures this behavior by replacing consecutive ε-transitions (on a path in the NFA) with a single ε-transition, while taking all lookaheads on a given path of ε-transitions and replacing them with a regular expression equivalent to the intersection of encountered lookaheads. Once we apply this procedure, ε-selfloops may be obtained, which are simply not used in the flattened version of $M({E_1}^*)$. It should be noted that applying the flattening procedure may produce states with more than two outgoing prioritized transitions. □

Remark 5. The proof above can allow for lazy Kleene closures by switching the priorities of the outgoing ε-transitions from state q_1 in $M(E_1^*)$.

Lemma 6. *For every a-regex E there exists an NFA A such that $\mathcal{L}(E) = \mathcal{L}(A)$ and $|A| \in 2^{\mathcal{O}((k+1)|E|)}$ where k is the number of atomic subgroups in E.*

Proof (sketch). A Boolean automaton with n states can be simulated by an NFA with $2^n + 1$ states [HK11], and can be used to implement lookaheads. Without a complete definition, note that Boolean automata may have transitions of the form $(q, \alpha, (p \wedge \neg p'))$, i.e., one can in q accept αw if w can be accepted from p but it *cannot* be accepted from p' ([HK11] does not permit $\alpha = \varepsilon$, but without ε-loops and each state having either transitions on symbols or ε, but not both, ε-transitions can be removed by replacing a state with the Boolean formula defining the transition on ε, in other Boolean formulas). A transition from q to p with lookahead $\neg F$ can be simulated in a Boolean automaton by constructing a Boolean automaton A with $\mathcal{L}(F) = \mathcal{L}(A)$ and $(q, \varepsilon, (p \wedge \neg A_{q_0})) \in A_\delta$.

To complete the proof we argue that the NFA with lookaheads $M(E)$ constructed in the proof of Lemma 5 can be converted into a Boolean automaton with $\mathcal{O}((k+1)|M(E)|)$ states, where k is the number of atomic subgroups in E. Notice that $M(E)$ has $\mathcal{O}(|E|)$ states as constructed. As only the language matters, prioritized transitions are treated as ε-transitions.

Consider a lookahead $\neg G\Sigma^*$ added to a transition when constructing $M(\triangleright F)$ in Lemma 5. Notice that there will exist some $q \in M(F\Sigma^*)_Q$ such that $\mathcal{L}(G\Sigma^*)$ is accepted by $M(F\Sigma^*)$ when starting in q (choosing the q which corresponds to the higher-priority choice). As such, let $\{E_1, \ldots, E_k\}$ be all the subexpressions such that each $(\triangleright E_i)$ occurs in E. Then construct the disjoint union of all these automata $A = M(E) \cup M(E_1\Sigma^*) \cup \cdots \cup M(E_k\Sigma^*)$ taking $A_{q_0} = M(E)_{q_0}$. Then, for each transition $(q, \alpha, \neg G\Sigma^*, p)$ in A find the state r such that A accepts $\mathcal{L}(G\Sigma^*)$ when started in r (as noted above r was originally in $M(E_i\Sigma^*)_Q$ for the E_i most closely enclosing this transition), and replace the transition by $(q, \alpha, (p \wedge \neg r))$. The intersected lookaheads created by the flattening at the end of Lemma 5 can be handled by a conjunction of lookaheads in the formula. □

Theorem 1. *The class of languages matched by a-regexes is precisely the class of regular languages.*

Proof. This follows from the combination of Lemma 1, as a regular expression is an a-regex representing the same language, and Lemma 5, demonstrating that a finite automaton can describe the language of an a-regex. □

We now demonstrate that a-regexes are exponentially more succinct than regular expressions for some languages. We start with two utility lemmas, which demonstrate that we can perform a limited set subtraction and intersection of languages using atomic subgroups.

Lemma 7. *For a-regexes F and G over the alphabet Σ with $\varepsilon \notin \mathcal{L}(G)$ we have that $\mathcal{L}((\triangleright(F\Sigma^* \,|\, \varepsilon))G) = \mathcal{L}(G) \setminus \mathcal{L}(F\Sigma^*)$.*

Proof. From $\mathcal{L}(\triangleright F\Sigma^*) = \mathcal{L}(F\Sigma^*)$ (both consist of all strings which have a prefix in $\mathcal{L}(F)$) and $\varepsilon \notin \mathcal{L}(G)$ it follows that $\mathcal{L}((\triangleright(F\Sigma^* \,|\, \varepsilon))G) \cap \mathcal{L}(F\Sigma^*) = \emptyset$. To complete the proof we need to show that if $w \in \mathcal{L}(G)$ but $w \notin \mathcal{L}(F\Sigma^*)$, then $w \in \mathcal{L}((\triangleright(F\Sigma^* \,|\, \varepsilon))G)$. This is indeed the case since $w \notin \mathcal{L}(F\Sigma^*)$ implies that ε, and not $F\Sigma^*$, is used when matching a string w with $(\triangleright(F\Sigma^* \,|\, \varepsilon))G$. □

Lemma 8. *Let $E_1, \ldots, E_n$ be a-regexes over the alphabet Σ and $\# \notin \Sigma$. Then there is an a-regex E over the alphabet $\Sigma \cup \{\#\}$ such that $|E| \leq cn|\Sigma| + \sum_{i=1}^{n} |E_i|$, for some constant c, and $\mathcal{L}(E) = (\mathcal{L}(E_1) \cap \ldots \cap \mathcal{L}(E_n))\#$.*

Proof. Let $\Gamma = \Sigma \cup \{\#\}$. The language equality when replacing Σ by Γ in Lemma 7 becomes:

$$\mathcal{L}((\triangleright(F\Gamma^* \,|\, \varepsilon))G) = \mathcal{L}(G) \setminus \mathcal{L}(F\Gamma^*) \tag{1}$$

Let $E_1' = (\triangleright(E_1\#\Gamma^* \,|\, \varepsilon))\Sigma^*\#$ be the lhs of (1) when setting $F = E_1\#$ and $G = \Sigma^*\#$. Then from (1) we have $\mathcal{L}(E_1') = \mathcal{L}(\Sigma^*\#) \setminus \mathcal{L}(E_1\#\Gamma^*) = (\mathcal{L}(\Sigma^*) \setminus \mathcal{L}(E_1))\#$.

Next, let $E_{1,2} = (\triangleright(E_1'\Gamma^* \,|\, \varepsilon))E_2\#$, again forming the lhs of (1) when taking $F = E_1'$ and $G = E_2\#$. Again from (1) we have $\mathcal{L}(E_{1,2}) = \mathcal{L}(E_2\#) \setminus \mathcal{L}(E_1') = (\mathcal{L}(E_1) \cap \mathcal{L}(E_2))\#$. The result now follows by repeating this construction. □

Using the above lemmas we can now demonstrate a lower bound on the worst-case state complexity of an a-regex.

Theorem 2. *There exists a sequence $F_1, F_2, \ldots$ of a-regexes of increasing size such that the number of states in a minimal DFA for $\mathcal{L}(F_n)$ is in $2^{2^{\Omega(\sqrt{|F_n|})}}$.*

Proof. By using Lemma 8 we obtain a sequence of a-regexes F_n with $\mathcal{L}(F_n) = \Sigma^* a((\Sigma^{p_1})^+ \cap \ldots \cap (\Sigma^{p_1})^+)\#$ and $|F_n| \in \Theta(p_1 + \ldots + p_n)$, where $a \in \Sigma$, $|\Sigma| \geq 2$ and $p_1, \ldots, p_n$ the first n prime numbers. Note that $\mathcal{L}(F_n) = \Sigma^* a(\Sigma^{r_n})^+\#$, where $r_n = p_1 \cdot p_2 \cdot \ldots \cdot p_n$. Let $D(L_n)$ be the complete minimal DFA for $\mathcal{L}(F_n)$ and $s(n)$ the number of states in $D(F_n)$. Thus $s(n) = 2^{r_n+1} + 2$ (which can for example be verified by using derivatives). By showing that $r_n \in 2^{\Omega(\sqrt{p_1+\ldots+p_n})}$ we thus have that $s(n) \in 2^{2^{\Omega(\sqrt{|F_n|})}}$. To obtain that $r_n \in 2^{\Omega(\sqrt{p_1+\ldots+p_n})}$, make use of the results stating that; the sum of the first n prime numbers is asymptotically equal to $(n^2 \ln n)/2$; and; the product of the first n prime numbers (the so called primorial function), is equal to $e^{(1+o(1))n \ln n}$. □

For NFA the lower bound on worst-case state complexity indirectly established in Theorem 2 ($2^{\Omega(\sqrt{n})}$ for NFA) can be improved upon.

Theorem 3. *For every integer $k \geq 1$ there exists an a-regex E_k of size $\mathcal{O}(k)$ such that a state minimal NFA (and thus also every regular expression) for $\mathcal{L}(E_k)$ contains $2^{\Omega(k)}$ states. Furthermore, E_k contains only one atomic subgroup.*

Proof. Let $\Sigma = \{0, 1\}$ and for $k \geq 1$ let $F = \Sigma^*(0\Sigma^{k-1}1 \,|\, 1\Sigma^{k-1}0)$ and $G = \Sigma^{2k}$ in Lemma 7. Then if $E_k = (\triangleright(F\Sigma^* \,|\, \varepsilon))G$, we have, via Lemma 7, that

$$\mathcal{L}(E_k) = \mathcal{L}(G) \setminus \mathcal{L}(F\Sigma^*) = \Sigma^{2k} \setminus \mathcal{L}(\Sigma^*(0\Sigma^{k-1}1 \,|\, 1\Sigma^{k-1}0)\Sigma^*) = \{ww \,|\, w \in \Sigma^k\}.$$

To complete the proof, note that from the pigeon-hole principle it follows that no NFA with fever than 2^k states can accept the language $\mathcal{L}(E_k)$ (for a detailed argument see the proof of Theorem 6 in [BBD+13] where a language very similar to $\mathcal{L}(E_k)$ is considered). □

Finally we show that deciding emptiness of a-regexes is PSPACE-complete.

Theorem 4. *The problem of deciding whether $\mathcal{L}(E) = \emptyset$, for an a-regex E, is PSPACE-complete.*

Proof. First we show that deciding emptiness is PSPACE-hard. With $\Gamma = \Sigma \cup \{\#\}$, where $\# \notin \Sigma$, and $E_1' = (\triangleright(E_1\#\Gamma^* \,|\, \varepsilon))\Sigma^*\#$, we have from the proof of Lemma 8 that $\mathcal{L}(E_1') = (\mathcal{L}(\Sigma^*) \setminus \mathcal{L}(E_1))\#$. Thus $\mathcal{L}(E_1') = \emptyset$ precisely when $\mathcal{L}(E_1) = \Sigma^*$. Since deciding if $\mathcal{L}(E_1) = \Sigma^*$ for a regular expression E_1 is PSPACE-hard, we have that deciding emptiness for a-regexes is PSPACE-hard.

It can be decided whether $\mathcal{L}(E) = \emptyset$ *in* PSPACE by constructing the Boolean automaton described in the proof of Lemma 6 (polynomial in the size of E), this automaton can then be converted into an alternating finite automaton and emptiness-checked in PSPACE using results from [HK11]. □

5 On Arriving at the Semantics

The atomic subgroups semantics defined in this paper should agree with most common regular expression libraries, but the reference point primarily used has been the Java implementation (where they are called "independent subgroups"). The priorities of Definition 4 follow from the depth-first search implementation which Java and many others use (or at least simulate the effects of), semantics which are treated at length in [BvdM16], where the specifics of the Java implementation are also described. For atomic subgroups specifically a further analysis of the Java source code (version 8u40-25) was performed. In so doing we informally deduced the stack-discarding behavior which causes the atomic subgroup semantics in Java. However, the source code for the matcher itself is close to 6000 lines, supported by several other classes, and has little documentation, making a truly formal proof of equivalence fall outside the scope of this paper.

To corroborate the semantics of Definition 4 without a formal proof an implementation computing $m(E, w)$ for any expression E and string w was created. This was compared to Java using both a full match (i.e. verifying that $w \in m(E, w)$ if and only if E matches w in Java), and by comparing the preferred prefixes (where the prefix of a string w matched by an expression E in Java is compared with the first element in $m(E, w)$). All strings $w \in \{a, b, c\}^*$ with $|w| \leq 5$ were tested against all expressions with up to three operations, with no discrepancies found between Java and the definition.

6 Conclusions and Future Work

While this paper gives formal definitions and some key results on the previously only informally documented atomic subgroups, numerous open questions remain. Specifically, the complexity of the uniform membership problem (it is linear in the non-uniform case due to the regularity of a-regexes) remains open ($\mathcal{O}(n^3)$ appears likely). Also, the worst-case bounds on the minimum number of states required to accept the language matched by an a-regex are not tight, with DFA having the span between $2^{2^{\Omega(\sqrt{|E|})}}$ and $2^{2^{\mathcal{O}((k+1)|E|)}}$ (where k is the number of atomic subgroups in E), and NFA between $2^{\Omega(|E|)}$ and $2^{\mathcal{O}((k+1)|E|)}$.

References

[Aho90] Aho, A.: Algorithms for finding patterns in strings. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. A, pp. 255–300. MIT Press (1990)

[BBD+13] Berglund, M., Björklund, H., Drewes, F., van der Merwe, B., Watson, B.: Cuts in regular expressions. In: Béal, M.-P., Carton, O. (eds.) DLT 2013. LNCS, vol. 7907, pp. 70–81. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38771-5_8

[BvdM16] Berglund, M., van der Merwe, B.: On the semantics of regular expression parsing in the wild. Theor. Comput. Sci. (2016). doi:10.1016/j.tcs.2016.09.006

[Fri97] Friedl, J.: Mastering regular expressions, 1st edn. O'Reilly & Associates Inc. (1997)

[HK11] Holzer, M., Kutrib, M.: Descriptional and computational complexity of finite automata—a survey. Inf. Comput. **209**(3), 456–470 (2011)

[Reg] RegexAdvice.com. Regular expression library. http://regexlib.com. Accessed 9 Jan 2017

[SMV12] Sakuma, Y., Minamide, Y., Voronkov, A.: Translating regular expression matching into transducers. J. Appl. Logic **10**(1), 32–51 (2012)

[WvdMBW16] Weideman, N., van der Merwe, B., Berglund, M., Watson, B.: Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In: Han, Y.-S., Salomaa, K. (eds.) CIAA 2016. LNCS, vol. 9705, pp. 322–334. Springer, Cham (2016). doi:10.1007/978-3-319-40946-7_27