


Regular Expressions with Lookahead


Martin Berglund

(Department of Information Science, Stellenbosch University, South Africa

 <https://orcid.org/0000-0002-3692-6994>, pंबरlund@sun.ac.za)


Brink van der Merwe

(Computer Science Division, Stellenbosch University, South Africa

 <https://orcid.org/0000-0001-5010-9934>, abvdm@cs.sun.ac.za)

Steyn van Litsenborgh

(Computer Science Division, Stellenbosch University, South Africa

 <https://orcid.org/0000-0001-6513-1704>, steyn.van.l@gmail.com)

Abstract: This paper investigates regular expressions which in addition to the standard operators of union, concatenation, and Kleene star, have *lookaheads*. We show how to translate regular expressions with lookaheads (*REwLA*) to equivalent Boolean automata having at most 3 states more than the length of the *REwLA*. We also investigate the state complexity when translating *REwLA* to equivalent deterministic finite automata (DFA).

Keywords: Regular expressions, Lookahead expressions, Boolean automata

Categories: F.1.1, F.1.2, F.4.3

DOI: 10.3897/jucs.66330

1 Introduction

As regular expressions paved their way into everything from programming languages to databases, the standard notion was enriched with several constructs. To accommodate this, many implementations have resorted to using algorithms based on backtracking, which has the weakness that matching takes exponential time (in the length of the input string) in the worst case unless potentially memory expensive memoization techniques are added to backtracking matchers [Davis 2019]. Consequently, not only did this create a gap between regular expressions in theory and practice, but gave rise to security vulnerabilities when misused in practice [Staicu and Pradel 2018; Davis et al. 2018].

Arguably, the most common difference between how regular expressions are used in practice, in contrast to how they have been studied, is the fact that more often than not regular expressions are used in submatching mode. That is, programmers use them to find the first submatch, from the left, in an input string (often iterating this procedure). This point of view is complicated by the lookahead extension (typically implemented using backtracking), making it possible to supply a lookahead expression at any point in a regular expression, describing in addition to the conditions enforced by the regular expression (without the lookaheads) the constraint that the lookahead expression should match (or fail to match) a prefix of the remainder of the input string.

It is important to note that once we consider regular expressions in submatching mode, where it might be possible for a regular expression to return more than one submatch

starting at a given position in an input string, that a disambiguation policy, such as the commonly used greedy (used in Perl-compatible engines, the standard example being PCRE [Hazel 2015]) or the less often encountered posix policy [Stallman 2008], be specified in order to select one of these possible submatches.

The combination of submatching with lookaheads causes regular expressions to have an influence on the possible suffixes beyond the returned submatch. Thus, regular expressions with lookaheads describe what gets matched *and* what is left over after the submatch. In many regular expression engines, $(?=r)$ and $(?!r)$ denote the positive and negative lookahead of r , respectively.

Lookaheads are also used in parsing expression grammars (*PEGs*) [Ford 2004], and in [Chida and Kuramitsu 2017] Linear *PEGs*, a subclass of *PEGs* that parse regular languages, is defined, demonstrating a close link between *PEGs* and regular expressions with lookaheads (*REwLA*). Several automata-based regular expression engines (such as those found in RE2, Rust, and Golang) do not support lookahead assertions, claiming that they do not support constructs for which only backtracking solutions are known to exist [Cox 2010]. Although we are not aware of regular expression engines with support for lookaheads that is not based on a backtracking algorithm, the results presented here can be directly applied to implement such an engine.

Lookaheads are zero-length assertions (i.e. they do not consume characters in the input string) that specify what should or should not follow the current position in the input string. Consider for example the regular expression $[0-9]+(?=U)$. On input 123 the match fails, since 123 is followed by ε and not the character U. On input 123U we match 123 with remainder U, and on input 123U12 we match 123 with remainder U12. We thus consider this regular expression with lookahead U, to describe the language of one or more digits followed by strings starting with U as lookahead. This concept of describing what gets matched and what should follow a given match, is formalized in the notion of a lookahead language.

The anchors \wedge and $\$$ are used to force the regular expression to match from the beginning and match all the way to the end of the input string, respectively. We ignore the \wedge anchor, assuming all matches start from the beginning of the input string instead of skipping a prefix (the shortest possible), i.e. we assume that all regular expressions begin with the anchor symbol \wedge (which we do not indicate). The reason for ignoring \wedge follows from the observation that the submatching behavior of two regular expressions with lookaheads are equivalent, whether we check equivalence when both expressions are forced to match from the beginning of an input string, or both are allowed to skip a prefix (the shortest possible) before (possibly) succeeding with a sub or full match. To see this, let E and F be regular expressions not starting with \wedge . If $\wedge E$ and $\wedge F$ matches identical prefixes of any string, then E and F will certainly match identical substrings of any given string and vice versa. But in contrast, for the anchor $\$$ we have that if E and F denotes $\wedge(=?=ab)a\$$ and $\wedge(=?=ac)a\$$ respectively, then both expressions match no string, since $a\$$ forces strings that could be matched to be only the symbol a , whereas the lookaheads require at least two symbols. However, if E' and F' denote $\wedge(=?=ab)a$ and $\wedge(=?=ac)a$ respectively, then E' matches the first symbol of a string when it is a , but only when a is directly followed by a suffix starting with b , whereas F' match a first symbol if it is an a directly followed by a suffix starting with c .

REwLA were originally formalized by Morihita [Morihita 2012], and in the follow-up [Miyazaki and Minamide 2019] it was shown that a *REwLA* of size n can be converted into DFA with at most $(2^{2^n} + 1)$ states, and asymptotically at least $2^{2^{\Omega(\sqrt{n})}}$ states in the worst case. The lower bound $2^{2^{\Omega(\sqrt{n})}}$ is obtained by using the common

strategy (in these cases) of considering the language where when going back k symbols from the right of a word, we get a specific symbol, and then expressing k , which is required to be a product of distinct primes, by using intersection of languages and thus using a number of symbols (in the $REwLA$) that is the sum of the prime factors of k in order to describe k . It is interesting to compare DFA state complexity of $REwLA$ to that of regular expressions that are extended by intersection and complement, where we have that it is not bounded from below by an elementary function [Stockmeyer and Meyer 1973].

Our contributions are (i) showing how to translate $REwLA$ into alternating (and Boolean) automata, and (ii) various state complexity results when converting $REwLA$ to equivalent DFA, improving results by [Miyazaki and Minamide 2019] (for all but $REwLA$ of length 1 or 2).

The outline of the paper is as follows. Next, we introduce the necessary definitions and notation, which is followed by a section describing how to translate $REwLA$ into alternating automata which gets converted into Boolean automata having at most three more states than the length of the corresponding $REwLA$. This is followed by [Section 4], where we give various state complexity results. After this, we provide some empirical results followed by our conclusions.

2 Definitions and Notation

Let Σ be a finite alphabet, ε the empty string, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $|w|$ be the length of w , in particular, $|\varepsilon| = 0$. We denote the empty set by \emptyset . For any set A , let 2^A be the power set of A .

A *lookahead language* is a subset of $\Sigma^* \times \Sigma^*$, which describes what is matched and what follows subsequently. For example, the language $\{(ab, c)\}$ contains the string “ab”, if it is followed precisely by “c”. Concatenation of lookahead languages is defined as $R \cdot S = \{(xy, z) \mid (x, yz) \in R, (y, z) \in S\}$. It is shown in [Miyazaki and Minamide 2019] that lookahead languages form a monoid under concatenation, with unit element $I = \{\varepsilon\} \times \Sigma^*$, and that concatenation for $R, S \subseteq I$ (which is equivalent to intersection in this case) is commutative and idempotent. The Kleene star operator is defined for lookahead languages in terms of concatenation and union as usual, with $R^* = \bigcup_{i=0}^{\infty} R^i$, assuming $R^0 = I$ and $R^{n+1} = R^n \cdot R$.

Next, we define lookahead assertions. As in [Miyazaki and Minamide 2019], we adopt the notation of *PEGs* [Ford 2004]. *PEGs* is a formalism consisting of a set of rules for recognizing strings in a language, and syntactically they look similar to context-free grammars, but they have a different interpretation and are closer to how string recognition tends to be done in practice by a recursive descent parser. As in *PEGs*, $\&R$ and $!R$ denote positive and negative lookaheads, respectively. We define the *positive lookahead* of a lookahead language R , denoted by $\&R$, as $\{\varepsilon\} \times \{xy \mid (x, y) \in R\}$. The positive lookahead operator satisfies the following relationships: $\&(\&R) = \&R$, $\&(R \cup S) = \&R \cup \&S$, and if $R \subseteq I$, then $\&(R \cdot S) = R \cdot \&S$. We define the *negative lookahead* of a lookahead language R , denoted as $!R$, by $\{\varepsilon\} \times (\Sigma^* \setminus \{xy \mid (x, y) \in R\})$. Positive and negative lookahead operators are related as follows: $\&R = !(!R)$.

Definition 1. *The set of regular expressions with lookaheads over the alphabet Σ , denoted as $REwLA(\Sigma)$ (or simply $REwLA$ if Σ is clear from the context), is defined inductively as follows, where r_1 and r_2 denote $REwLA$ already defined:*

(i) \emptyset , the empty language; (ii) ε , the empty string; (iii) a , for $a \in \Sigma$; (iv) $(r_1 | r_2)$; (v) $(r_1 \cdot r_2)$; (vi) (r_1^*) ; (vii) $(!r_1)$.

For $r \in REwLA(\Sigma)$, let $\|r\|$ denote the number of symbols from Σ that occur in r . We refer to $\|r\|$ as the length of the $REwLA$ r .

Remarks.

1. For $r \in REwLA$, $(\&r)$ denotes $(!(r))$.
2. Although character classes, for example $[A-Z]$ (denoting any letter in the uppercase alphabet in the English language) or $\backslash d$ (denoting any digit) should be considered from a practical point of view, we (beyond in some examples) leave considerations dealing with large alphabet sizes as future work, although we do use the metacharacter “.” as an abbreviation for $a_1 | \dots | a_n$, where $\Sigma = \{a_1, \dots, a_n\}$. We should point out that it is easy to verify that our state complexity results extend to the case where we allow character classes and count a character class as contributing one to the length of a $REwLA$.
3. Regular expressions are defined as usual, i.e. as in the definition above for $REwLA$, but without allowing $!r_1$ (and $\&r$). The set of regular expressions over the alphabet Σ is denoted by $Reg(\Sigma)$ or simply Reg if Σ is clear from the context. We thus regard $Reg \subset REwLA$.
4. Regular expressions set in *typewriter font* are examples of the Java syntax, which is mostly similar to other libraries. See [Davis et al. 2019] for an examination of how portable regular expressions are between programming languages.

Definition 2. The matching semantics of $REwLA$ is defined by the function $\mathcal{B} : REwLA \rightarrow \Sigma^* \times \Sigma^*$, which maps $REwLA$ to languages with lookaheads. The function \mathcal{B} is defined inductively as follows:

(i) $\mathcal{B}(\emptyset) = \emptyset$; (ii) $\mathcal{B}(\varepsilon) = I$; (iii) $\mathcal{B}(a) = \{a\} \times \Sigma^*$; (iv) $\mathcal{B}((r_1 | r_2)) = \mathcal{B}(r_1) \cup \mathcal{B}(r_2)$; (v) $\mathcal{B}(r_1 \cdot r_2) = \mathcal{B}(r_1) \cdot \mathcal{B}(r_2)$; (vi) $\mathcal{B}((r^*)) = \mathcal{B}(r)^*$; (vii) $\mathcal{B}(!r) = !\mathcal{B}(r)$.

Example 1. In this example we show how to use lookaheads to describe the intersections of regular languages, by using both Java regular expression syntax and $PEGs$ notation. The regular expression r given by

$$(?!.*[a-zA-Z])(?=.*\d)(?=.*[!@#$(), ;])\{8,\}$$$

can be used to validate passwords (on a web server) that satisfy each of the following conditions: (i) a password contains at least one letter from $[a-zA-Z]$, at least one digit, and at least one of the specified special symbols (these three conditions are validated individually by each of the three lookahead expressions), and (ii) should have a minimum length of eight, as encoded by the subexpression $\{8,\}$, the only part of the expression *not* in a lookahead (often called the “main” part). By making use of $PEGs$ notation, we can write this regular expression as the $REwLA$

$$\&(.*[a-zA-Z])\&(.*\d)\&(.*[!@#$(), ;])\{8,\}$$$

if we extend our $REwLA$ notation to allow the subexpressions $[!@#$(), ;]$ and $\{8,\}$. By making use of lookahead expressions and preceding these expressions by $.*$, we instruct the lookahead expression to match any character until it starts matching the expressions following the $.*$ expression. Given that a lookahead expression will only assert conditions and not consume any characters, the only subexpression in r consuming characters is the “main” part. All three lookahead expressions and the “main” part of

r each describe a regular language. A password is valid if a prefix of the password is in the language described by each of the lookaheads, and if it is matched by the “main” part of r .

In general, suppose we have two regular expressions r_1 and r_2 , then the regular expression $(?=r_1\$)(?=r_2\$).*\$$ (where the subexpression $.*$ matches any string), represents the intersection of the languages described by r_1 and r_2 . One can similarly encode the negation of a (language described by a) regular expression using a negative lookahead.

Lemma 3. For $r \in REwLA$, $\mathcal{B}(\&r) = \&\mathcal{B}(r)$.

Proof. The result follows by noting that $\&r$ denotes $!(r)$, the identity $\&\mathcal{B}(r) = !(\mathcal{B}(r))$ and observing that $\mathcal{B}(!(r)) = !(\mathcal{B}(r))$. \square

Definition 4. The language of $r \in REwLA$, denoted $\mathcal{L}(r)$, is defined as $\{x \mid (x, \varepsilon) \in \mathcal{B}(r)\}$ (which is a regular language).

Note for $r, r' \in REwLA$, $\mathcal{B}(r) = \mathcal{B}(r')$ implies $\mathcal{L}(r) = \mathcal{L}(r')$, but the converse obviously does not hold in general. Also, we refer to $\mathcal{B}(r)$ as the lookahead language of r .

Remark 1. For $r \in REwLA$ we define the following additional iterative operators. The optional match ($r?$) abbreviates $(r \mid \varepsilon)$, and the positive closure (r^+) abbreviates (rr^*) .

The end of string anchor “\$” asserts that we are at the end of the input and can thus only match ε followed by ε . The “\$” expression can be represented by “!”. and we have that $\mathcal{B}(\$) = \{\varepsilon\} \times \{\varepsilon\}$. Furthermore, $\mathcal{B}(r\$) = \mathcal{L}(r) \times \{\varepsilon\}$.

Next, we define alternating automata (AFA)—a generalization of the standard notion of nondeterministic automata. Nondeterminism provides a computing device with the power of existential choice. The dual of such a device has universal choice and AFA have both existential and universal choice. Alternation was studied in [Kozen 1976; Chandra et al. 1981] in the context of Turing machines and in [Brzozowski and Leiss 1980; Chandra et al. 1981] for finite automata. In general, the transition function of an AFA on a given alphabet symbol from a given state, can be an arbitrary Boolean function over states. Based on our use cases, we consider the following two classes of automata (with our version of AFA contained in the class of Boolean automata, as usual, once we apply the ε -removal procedure described later): (i) AFA (with ε -transitions) having a single initial state and a transition function making use of Boolean formulas in the transition function which can be represented as a conjunction or disjunction over states, with all transitions from a given state making use of either only conjunction or only disjunction, and (ii) Boolean automata (without ε -transitions) where we replace the initial state by an arbitrary Boolean formula, and where we make use of arbitrary Boolean formulas over Q in the transition function. We denote by $\mathcal{B}(Q)$ the set of Boolean formulas over Q , inductively defined as usual with negation (“not”), conjunction (“and”), disjunction (“or”), over Q as variables, plus the constants *true* and *false*, all defined as usual. Let $\mathcal{B}^+(Q) \subset \mathcal{B}(Q)$ denote the *monotone* (or *positive*) Boolean functions over Q , i.e. Boolean formulas not making use of negation. For any Boolean formulas $f, g \in \mathcal{B}(Q)$ and $q \in Q$ let $f \llbracket q \leftarrow g \rrbracket$ denote the formula resulting when replacing all occurrences of q in f with a copy of the formula g . In cases where the distinction is unimportant we often identify Boolean formulas representing equivalent Boolean functions and use the terminology Boolean formulas and Boolean functions interchangeably.

Our definition of AFA is complicated by the inclusion of epsilon transitions. These are not usually included in definitions of AFA or Boolean automata, notably they are not considered in any of the papers listed above. They are however very helpful when compositionally constructing AFA from *REwLA*, as we will do in the next section.

Definition 5. An alternating finite automaton (AFA), with Q being the disjoint union of Q_{\exists} and Q_{\forall} , is a tuple $A = (Q_{\exists}, Q_{\forall}, \Sigma, q_0, \delta, F)$, where:

(i) Q_{\exists} is the finite set of existential states; (ii) Q_{\forall} is the finite set of universal states; (iii) Σ is the input alphabet; (iv) $q_0 \in Q$ is the initial state; (v) $\delta : Q \times \Sigma_{\varepsilon} \rightarrow 2^Q$ is the transition function; (vi) $F \subseteq Q$ is the set of final states.

For a Boolean automaton the state set is simply Q (there are no distinguished existential or universal states), we replace $q_0 \in Q$ by $q'_0 \in \mathcal{B}(Q)$, and the transition function $\delta : Q \times \Sigma_{\varepsilon} \rightarrow 2^Q$ by $\delta' : Q \times \Sigma \rightarrow \mathcal{B}(Q)$.

To simplify our discussion, we assume that if there is a transition on epsilon from a given state in an AFA, then there are no transitions on alphabet symbols from this state.

Next, we define the language accepted by an AFA in terms of accepting runs. We begin with a preliminary definition to handle ε -transitions.

Definition 6. Let $A = (Q_{\exists}, Q_{\forall}, \Sigma, q_0, \delta, F)$ be any AFA. The forall epsilon closure of a set of states $P \subseteq Q$, denoted $C_{\forall}(P)$, is the smallest set such that $P \subseteq C_{\forall}(P)$ and $\{q' \mid q \in C_{\forall}(P) \cap Q_{\forall}, (q, \varepsilon, q') \in \delta\} \subseteq C_{\forall}(P)$. Further, the full epsilon closure of P is the smallest set of sets $C_F(P) \subseteq 2^Q$, such that $C_{\forall}(P) \in C_F(P)$, and for $P' \in C_F(P)$, $q \in P' \cap Q_{\exists}$, and $(q, \varepsilon, q') \in \delta$, we have $((P' \setminus \{q\}) \cup C_{\forall}(\{q'\})) \in C_F(P)$.

That is, $C_{\forall}(P)$ is the set resulting when starting from P and inductively adding any state reachable on an ε -transition from a \forall -state already in P , while $C_F(P)$ contains the \forall -closure for every way to take any number (including zero) of ε -transitions from an \exists -state in a set in $C_F(P)$. We denote by $\overline{C}_F(P)$ the set of non-empty sets obtained by deleting states with epsilon outgoing transitions from the sets in $C_F(P)$. We assume that $\overline{C}_F(\{q\}) \neq \emptyset$ for all states q , to avoid having to define acceptance for the situation of having a state q' reachable after having read an input word, being such that all states reachable from q' on ε -transitions, having (only) epsilon outgoing transitions.

A run of an AFA $A = (Q_{\exists}, Q_{\forall}, \Sigma, q_0, \delta, F)$ on a string $w = a_1 \dots a_n$ is a sequence of non-empty sets of states $Q_0, Q'_0, \dots, Q_n, Q'_n \subseteq Q_{\exists} \cup Q_{\forall}$ which fulfill the following conditions.

- As a base case $Q_0 = \{q_0\}$.
- For each i the set Q'_i is in the full epsilon closure of Q_i , i.e. $Q'_i \in \overline{C}_F(Q_i)$.
- Finally, Q_{i+1} corresponds to (one way of) reading the symbol a_{i+1} when in the states Q'_i . Specifically, Q_{i+1} is the smallest set containing the states dictated by
 - if $q \in Q'_i \cap Q_{\forall}$ then $\delta(q, a_{i+1}) \subseteq Q_{i+1}$, or
 - if $q \in Q'_i \cap Q_{\exists}$ then some $q' \in \delta(q, a_{i+1})$ is in Q_{i+1} .

A run $Q_0, Q'_0, \dots, Q_n, Q'_n$ is *accepting* if $Q'_n \subseteq F$.

We remove ε -transitions from an AFA and convert AFA to Boolean automata as follows. We begin by removing all states from Q having (only) epsilon outgoing transitions, replacing δ by $\delta' : Q \times \Sigma \rightarrow \mathcal{B}^+(Q)$, and replacing q_0 with $q'_0 \in \mathcal{B}^+(Q)$. Next we describe how to obtain q'_0 and δ' . For $P \subseteq Q$, let $\overline{C}_F^B(P)$ be the Boolean formula over Q obtained by replacing each $P' \in \overline{C}_F(P)$ by the conjunction over the states in P' , and then taking the disjunction over all conjuncts obtained for each $P' \in \overline{C}_F(P)$. We obtain q'_0 by taking $\overline{C}_F^B(\{q_0\})$. Similarly, $\delta'(q, a)$ is the disjunction or conjunction over

$\overline{C}_F^B(\{q'\})$, for all $q' \in \delta(q, a)$, depending on if q is an existential or universal state. Once we have a Boolean automaton with δ' making use only of positive Boolean formulas written in disjunctive normal form, we generalize the definition of a run $Q_0, Q'_0, \dots, Q_n, Q'_n$ by making each Q_i a formula, and each Q'_i is a way of choosing a single conjunction from Q_i (so Q'_i implies Q_i). A straightforward argument can be used to verify that the accepted language is not altered when applying the described ε -transition removal procedure.

Acceptance of strings by Boolean automata can also be considered from the point of view described next. We start with the initial Boolean formula, and then use the transition function that maps pairs of states and input symbols to Boolean formulas. As the input is read (from left to right), the automaton “builds” a Boolean formula, starting with the formula q'_0 , and on reading an input symbol $a \in \Sigma$, replacing every $q \in Q$ in the current formula by $\delta'(q, a)$, i.e. the formula f turns into the formula $f[\{q \leftarrow \delta'(q, a) \mid q \in Q\}]$. The input is accepted if the formula f constructed on reading the whole input is such that $f[\{q \leftarrow true \mid q \in F\}][\{q' \leftarrow false \mid q' \in (Q \setminus F)\}]$ evaluates to true (i.e. simplifies to true as no variables remain). When converting a Boolean automaton to an equivalent (minimal) DFA, we identify Boolean formulas with states of the DFA, with equivalent Boolean formulas (i.e. Boolean formulas defining the same Boolean function) being identified with the same state. In the case where we consider only formulas from $\mathcal{B}^+(Q)$, we thus obtain an upper bound on the number of states of the equivalent minimal DFA, of $\mathcal{D}(n)$ states, where n is the number of states in the corresponding Boolean automaton, and $\mathcal{D}(n)$ being the n th Dedekind number, since the n th Dedekind number is the number of monotone Boolean functions on n variables ([Kleitman 1969] and [Sloane 1964]). Note that $\mathcal{D}(n) < 2^{2^n}$ for $n \geq 1$, with 2^{2^n} being an upper bound on the number of states in the minimal DFA obtained when converting an arbitrary Boolean automaton with n states to a DFA.

3 Automata Construction for REwLA

In this section, we develop a translation from *REwLA* to finite automata by applying a bottom-up technique similar to [Thompson 1968], but producing an AFA (with “and” and “or” states) as the result. As we make use of standard Boolean automata and DFA in this section, we need a string representation of lookahead languages. We thus define three slightly different encodings of lookaheads as strings.

Definition 7. For a lookahead language L over the alphabet Σ , we define the following string encodings:

- The # string encoding is the string language $\{u\#v \mid (u, v) \in L\}$ where we assume $\# \notin \Sigma$.
- The \sim string encoding, used in [Miyazaki and Minamide 2019], is the string language $\{u\tilde{v}_1 \cdots \tilde{v}_k \mid (u, v) \in L, v = v_1 \cdots v_k \text{ where } v_i \in \Sigma \text{ for each } i\}$ over the alphabet $\Sigma \cup \tilde{\Sigma}$, where $\tilde{\Sigma}$ is Σ with each symbol decorated with a tilde and assumed to be disjoint from Σ .
- The null string encoding is the string language $\{uv \mid (u, v) \in L\}$. This case is discussed in Remark 2.

Example 2. Consider the lookahead word $(aab, bacb) \in L$, where L is a lookahead language over Σ . When using the # string encoding we obtain the string $aab\#bacb$, the \sim string encoding gives us the string $aab\tilde{b}\tilde{a}\tilde{c}\tilde{b}$, and finally the null encoding the string $aabbacab$.

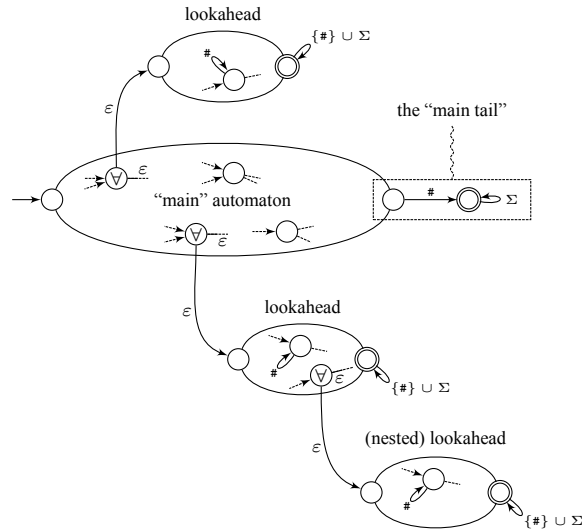


Figure 1: The structure of the alternating finite automata $\mathcal{A}(r)$ constructed from a $REwLA$ r by the procedure described in [Section 3].

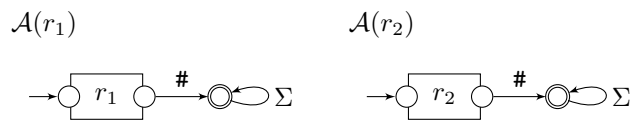
In this section we construct automata which match the $\#$ string encoded language of a $REwLA$. As we will see it is straightforward to convert this to either of the other two encodings. The automaton constructed to accept the $\#$ string encoding of a language matched by a $REwLA$ will be of the form shown in [Fig. 1] (when all lookaheads are positive lookaheads). Specifically, the automaton consists of a “main” part and one subautomaton for each lookahead subexpression. In [Fig. 1], r has three lookaheads, where one is nested within one of the others. Every \forall (“and”) state (note that we denote “or” states as blank circles) has precisely two outgoing edges, both labeled ε . The first stays inside the current subautomaton and the other is the *unique* transition entering the subexpression corresponding to the lookahead. The “main” subautomaton is responsible for matching the part of the expression not contained in any lookahead, with the addition that where the expression would accept, the “main” subautomaton instead reads a single $\#$ and then reads any $\#$ -free suffix (this part of the “main” subautomaton is called the “main tail”). In contrast, in the lookaheads every “or”-state, without epsilon outgoing transitions, has a self-loop labeled $\#$ and contain no other $\#$ -labeled transitions (i.e. if a lookahead subautomaton accepts a string w then it also accepts a string v derived from w by adding and removing any number of instances of the symbol $\#$). As will be pointed out later in this section, subautomata for negative lookaheads have a different structure than the lookahead subautomata in [Fig. 1] in terms of which states are respectively “and” and “or” states, and also in terms of which states are accepting.

Remark 2. *It is important to keep in mind that the real-world use-case will have no $\#$ marker or \sim string encoding. Rather the expression is matched against a full string, where the main and lookahead parts are initially undistinguished. This is a problem for [Miyazaki and Minamide 2019], as changing the \sim -encoding to a null encoding (by replacing \tilde{a} by a for $a \in \Sigma$, on a transition in the corresponding DFA) makes the*

DFA constructed non-deterministic, requiring another round of determinization and can thus only be used to establish a triple-exponential bound on the state complexity. As our construction first produces an AFA, one can simply replace the $\#$ -labeled transition in the main tail by an ε -transition (and remove the other $\#$ -labeled loops), and in a practical implementation it will be straightforward to extract a potential $\#$ position from an automaton run.

The null encoding being important in practice, and the \sim -encoding preexisting, raises the question of what the motivation for the $\#$ encoding is. It is used here as a standin for a finitely decorating transducer; a string transducer which adds a constant (in our case precisely one) number of symbols to its output to indicate a simple kind of parse. Specifically, here the transducer would take a null-encoded string as input and produce a $\#$ -encoded string as output, adding a single symbol to indicate the end of the main match. Much of this work overlaps with the idea of captures in regular expressions, in effect a parse that records which substrings are last matched by a subexpression, which is well modelled by a finitely decorating transducer, e.g. as in [Berglund et al. 2018]. Such simple transducers have many attractive properties (drawing upon the theory of transducers with origins [Bojańczyk 2014]) which can be leveraged for parsing algorithms. While fully developing these ideas are beyond the scope of this paper, we work with the $\#$ string encoding to make it possible in the future to incorporate these concepts.

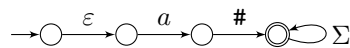
We will from here inductively define the alternating finite automaton that accepts the string language which is the $\#$ encoding of the lookahead language matched by a given $REwLA$, by defining the effect each operation has on the AFA constructed for the subexpressions involved. For any $REwLA$ r let $\mathcal{A}(r)$ denote the AFA constructed for r by this procedure. From here let r_1 and r_2 be two $REwLA$ for which $\mathcal{A}(r_1)$ and $\mathcal{A}(r_2)$ have been inductively constructed, with the automata being schematically drawn as follows:



Main tails will often be removed when constructing a new automaton from one or two subautomata, but the newly constructed automaton will always have a unique main tail.

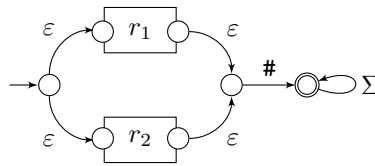
3.1 Terminal symbols

The AFA $\mathcal{A}(a)$ for the expression a consisting of a single terminal symbol $a \in \Sigma$ is constructed as follows. The epsilon transition from the initial state is included to ensure consistency, i.e. all constructed AFA will have only epsilon outgoing transitions from their respective initial states.



3.2 Union

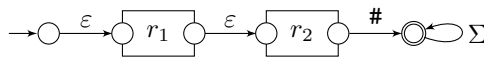
For any $REwLA$ r_1 and r_2 the AFA $\mathcal{A}(r_1 | r_2)$ is constructed as follows.



Note here that this is schematically almost the same as union in a Thompson construction, with the change that the main tail of r_1 and r_2 is cut off and a new common main tail is added (keeping the tail unique). Further, remember that $\mathcal{A}(r_1)$ and $\mathcal{A}(r_2)$ may contain lookaheads, which contain final states and have associated “and” states, but the union is only concerned with the initial and *main tail states*.

3.3 Concatenation

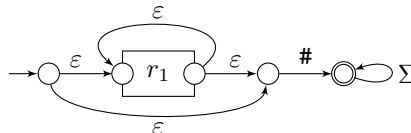
For any *REwLA* r_1 and r_2 the AFA $\mathcal{A}(r_1 \cdot r_2)$ is constructed as follows.



Thus, we remove the main tail of $\mathcal{A}(r_1)$ and replace the #-transition in the main tail of $\mathcal{A}(r_1)$ with an ϵ -transition to the initial state of $\mathcal{A}(r_2)$. Note that this construction is not applicable to AFA concatenation in general, since we do nothing to any number of final states which correspond to lookaheads inside $\mathcal{A}(r_1)$ and $\mathcal{A}(r_2)$. The *REwLA* concatenation definition indeed operates in this way, adding a suffix language only to the *main part* of $\mathcal{A}(r_1)$, and this is also *fortunate*, as AFA concatenation in general will in some cases cause an exponential blowup [M. Hospodár and G. Jirásková 2018]!

3.4 Kleene closure

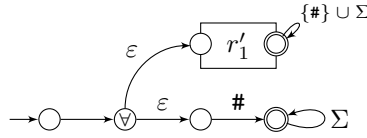
Finally, among the classic regular expression operators, for any *REwLA* r_1 the AFA $\mathcal{A}(r_1^*)$ is constructed as follows.



Thus, we again mimic the Thompson construction and ignore the final states in lookaheads.

3.5 Lookahead

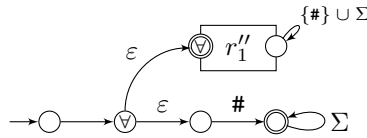
Although it follows from Definition 2 and Lemma 3 that it is not required to consider positive lookahead, we discuss this case to simplify the explanation of the construction of the negative lookahead case. For any *REwLA* r_1 we construct $\mathcal{A}(\&r_1)$ by building the following AFA.



The r'_1 subautomaton is derived from $\mathcal{A}(r_1)$ by adding a $\#$ -labeled self-loop to each “or”-state which is not a state with epsilon outgoing transitions and which does not have a $\#$ -labeled self-loop, ensuring the invariant noted in [Fig. 1] that when a lookahead subautomaton accepts a string, it will accept every string derivable by just adding or removing $\#$ symbols. Furthermore, the $\#$ -transition on the main tail in $\mathcal{A}(r_1)$ is replaced by an ε -transition in the lookahead subautomaton in $\mathcal{A}(\&r_1)$. The leading initial state is somewhat extraneous, but ensures all AFA have an “or” state as its initial state.

3.6 Negative lookahead

Finally, for any *REwLA* r_1 the automaton $\mathcal{A}(!r_1)$ is constructed by building the following AFA.



In this case, r''_1 is the negation of r'_1 with r'_1 as in the positive lookahead construction above (i.e. $w \in \mathcal{L}(r''_1)$ if and only if $w \notin \mathcal{L}(r'_1)$). As is well known [Chandra et al. 1981] (although this argument should be extended to also allow for ε -transitions) one can negate the alternating finite automaton r'_1 in linear time by first making r'_1 complete by adding a sink reject state, and then without adding any additional states or transitions replacing each “or”-state with an “and”-state, and vice versa, and making each non-final state final, and vice versa. Note since states in the lookahead of the AFA for $\mathcal{A}(\&r_1)$ and $\mathcal{A}(!r_1)$, with a self-loop on $\{\#\} \cup \Sigma$, only have the self-loop as outgoing transition, it does not matter if we regard these states as a universal or existential.

Theorem 1. *For any REwLA r we have $w \in \mathcal{L}(\mathcal{A}(r))$ if and only if $w = u\#v$ for some $(u, v) \in \mathcal{B}(r)$ (as in Definition 2). That is to say, the AFA construction is correct.*

Proof. We prove this by structural induction on the expression r , but rather than enumerate every case (as they are very similar) we demonstrate some key cases and leave the remainder as an exercise to the reader. Note that $\mathcal{A}(r)$ accepts only strings with precisely one $\#$, as the “main” part accepts only such strings (and is in a logical conjunction, i.e. a series of “ands”, with the lookahead parts), and specifically that all the lookahead parts do not care about $\#$ symbols, staying in the same state (after possibly taking ε -transitions) upon reading one.

Start by noting that $\mathcal{B}(a) = \{a\} \times \Sigma^*$, and that $\mathcal{A}(a)$ (as shown in the “terminal symbol” case) accepts precisely this language (by reading a , reading $\#$, then looping on Σ in an accepting state).

Next, we consider the lookahead case. By definition $\mathcal{B}(\&r_1) = \{(\varepsilon, xy) \mid (x, y) \in \mathcal{B}(r_1)\}$, so we need to show that $\mathcal{L}(\mathcal{A}(\&r)) = \{\#xy \mid x\#y \in \mathcal{L}(\mathcal{A}(r_1))\}$. The lookahead construction does precisely this:

1. If $x\#y$ is accepted by $\mathcal{A}(r_1)$ then the new “main” part in $\mathcal{A}(\&r_1)$ matches $\#\Sigma^*$ and the new lookahead will match $\#xy$, not caring about the marker moving (as lookaheads by construction do not “care” about the $\#$ symbol at all).
2. In the other direction, if $\mathcal{A}(\&r_1)$ matches $\#x$ (note that the $\#$ is always leading by construction) then $\mathcal{A}(r_1)$ matches $y\#z$ for some $yz = x$. This is the case as the lookahead subautomaton in $\mathcal{A}(\&r_1)$ is $\mathcal{A}(r_1)$ made not to care about $\#$ symbols.

As a final example case, we consider concatenation. By definition if $(x, yz) \in \mathcal{B}(r_1)$ and $(y, z) \in \mathcal{B}(r_2)$ then $(xy, z) \in \mathcal{B}(r_1 \cdot r_2)$. As such we need to prove that if $x\#yz \in \mathcal{A}(r_1)$ and $y\#z \in \mathcal{A}(r_2)$ then $xy\#z \in \mathcal{A}(r_1 \cdot r_2)$. This is the case as $x\#yz \in \mathcal{A}(r_1)$ means that $\mathcal{A}(r_1)$ reaches the main tail after reading x , and the concatenation construction attaches this state (of $\mathcal{A}(r_1)$ before the main tail) to the initial state in $\mathcal{A}(r_2)$, which then goes on to reach its main tail after y , read a $\#$ and accept z looping on its final state. From the perspective of the lookahead subautomata in $\mathcal{A}(r_1)$ and $\mathcal{A}(r_2)$, they accept the same suffix of xyz (and yz) as before the concatenation, ignoring the fact that $\#$ has moved, since lookaheads ignore all $\#$ markers by construction.

The arguments for the remaining cases (union and negative lookahead) follow the same structure and can be easily derived by the reader. \square

4 State Complexity Considerations

The alternating finite automaton construction of [Section 3] establishes upper bounds on the state complexity of *REwLA*, depending on the string encoding used (recall Definition 7). By using this construction, we begin by considering the number of states required in Boolean automata (obtained by removing states with epsilon outgoing transitions from $\mathcal{A}(r)$) for *REwLA* r , before moving on to state complexity for DFA constructed from *REwLA* (obtained by converting Boolean automata to DFA).

Theorem 2. *A Boolean automaton, using only monotone Boolean functions in its transitions function and accepting the language of a REwLA r , can be constructed with at most $\|r\| + 1$ states for the null string encoding, and at most $\|r\| + 2$ and $\|r\| + 3$ states respectively for the \sim and $\#$ string encodings.*

Proof. First we give an argument for the $\#$ string encoding case. We argue inductively over the various constructions given in the previous section. The induction assumption is that in addition to $\|r\|$ states, we need the following three special states: (i) a non-accepting state having only a $\#$ outgoing transition (to a state of type (ii)), (ii) an accepting state having a self-loop on Σ , and (iii) possibly an accepting state having a self-loop on $\{\#\} \cup \Sigma$.

We begin by making sure the statement holds for the base case where we consider the AFA for a REwLA consisting of a single alphabet symbol (with no lookaheads), and then remove the initial state, since it has only an epsilon outgoing transition. Thus, for the base case we need a single state and two more states (of types (i) and (ii) respectively) for the main tail. Next we consider the inductive case where a Boolean automaton is constructed for a REwLA of a union, concatenation, Kleene closure, lookahead or negative lookahead, given Boolean automata are already constructed for the subexpressions used by any of these operators, and assuming that the Boolean automata for these subexpressions satisfy the inductive hypothesis. After applying the constructions outlined in the previous section for the various operators, we remove states with epsilon outgoing transitions. Note that a non-accepting state with a self-loop on $\{\#\} \cup \Sigma$, is a sink reject state, and can

simply be removed with all transitions going to it. Also, states (i) and (ii) are required for the main tail, and state (iii) in the construction of lookaheads. All states of type (iii) can be merged into a single state this type, and there will be only one state of each of the types (i) and (ii).

In using the constructed Boolean automaton for the # string encoding, we construct the Boolean automaton for the null string encoding by using the automaton for the # string encoding, replacing # on transitions by ε (except #-loops which we simply remove), and observing that this will make it possible to remove state (i) and merge states (ii) and (iii). A similar procedure is used for the \sim string encoding, except that we add to transitions in lookaheads and negative lookaheads also \sim versions of alphabet symbols, which will have the effect that we can no longer merge states (ii) and (iii) as in the null string encoding case, since where the one will have a loop on Σ , the other will have a loop on $\Sigma \cup \bar{\Sigma}$. \square

It is interesting to observe that in contrast to regular expressions with intersection and negation, *REwLA* do not suffer from non-elementary state complexity when converted to DFA, although they do contain a form of intersection and negation.

Our upper bound in the next theorem, for the \sim string encoding case should be compared to the upper bound of $2^{2^{\|r\|}} + 1$, given in [Miyazaki and Minamide 2019]. Recall that $\mathcal{D}(n)$ denotes the n -th Dedekind number, with $\mathcal{D}(n) < 2^{2^n}$ for $n \geq 1$. The value of $\mathcal{D}(n)$ is only known for $0 \leq n \leq 8$ [Sloane 1964], and using these values for $\mathcal{D}(2)$, $\mathcal{D}(3)$ and $\mathcal{D}(4)$, we obtain that the bound by [Miyazaki and Minamide 2019] is better for $\|r\| = 1$ or 2, where ours (stated in the next theorem) is better from 3 onwards. This can be seen by noting that $(2^{2^3} + 1) > \mathcal{D}(4)$ and observing that from the definition of Dedekind numbers follows that $\mathcal{D}(n+1) \leq \mathcal{D}(n) \cdot \mathcal{D}(n)$ (whereas $2^{2^{n+1}} = 2^{2^n} \cdot 2^{2^n}$), since we can obtain all monotone Boolean functions on $(n+1)$ variables from the product of the monotone Boolean functions that always exclude the $(n+1)$ st variable from the sets of states in its domain with those monotone Boolean functions that always include it in its sets of states. Also, when considering the number of states for the three types of non-equivalent *REwLA* of size 1, i.e. a , $!a$ and $\&a$, we obtain that the optimal upper bound for $\|r\| = 1$ for the null, \sim and # string encodings are 3, 3 and 4 respectively, where our bounds in the next theorem are 3, 6 and 20 respectively. Using these precise bounds for the case $\|r\| = 1$ in combination with the argument used in the proof of the next theorem, we obtain that the bound in the previous theorem is tight for the null encoding case, but it could be off by 1 in the \sim and # cases.

Theorem 3. *A minimal (complete) DFA accepting the encoding of the lookahead language matched by a REwLA r has a number of states bounded from above as follows, depending on the string encoding used: (i) $\mathcal{D}(\|r\|)$ for null, (ii) $\mathcal{D}(\|r\| + 1)$ for \sim , and finally, (iii) $\mathcal{D}(\|r\| + 2)$ for the # string encoding.*

Proof. We apply to the result of the previous theorem the observation that when converting Boolean automata to equivalent DFA, we may regard the states of the DFA as Boolean formulas over the states of the original Boolean automaton (with equivalent Boolean formulas being identified). We can ignore a sink accept state in the Boolean automaton, since it corresponds to the Boolean formula True (and similarly for a sink reject state, although sink reject states are already excluded from the result of our previous theorem). Furthermore, since we do not make use of negation in the transition functions of our Boolean automata, we have Dedekind numbers (instead of $2^{2^{\|r\|}}$) in our bounds. \square

Remark 3. In [Miyazaki and Minamide 2019] an asymptotic lower bound for DFA state complexity of $2^{2^{\Omega(\sqrt{\|r\|})}}$ when converting *REwLA* to DFA, when using the \sim string encoding, is given. The class of examples constructed (in their Section 3.6) has no negative lookaheads, so it also provides an asymptotic lower bound for the case without negative lookaheads. While their lower bound class of examples is constructed for the \sim string encoding, the construction forces all lookahead words (u, v) to have $v = \varepsilon$, so the construction works the same for the null string encoding and requires one additional state for the $\#$ string encoding. We offer no improvement on this lower bound here.

To wrap-up this section we parameterize *REwLA* over the non-negative integers. Our next definition is inspired by the definitions and results from [Keeler and Salomaa 2020]. This definition and the following theorem also generalizes the reason for why better DFA state complexity is possible for the password example from the introduction, compared to the general case as stated in Theorem 3. We state the next definition and final theorem of this section, only for the $\#$ string encoding case. The fact that the next definition only applies to the $\#$ string encoding case follows from the fact that $\mathcal{A}(r)$ accepts the $\#$ string encoding of tuples in $\mathcal{B}(r)$.

Definition 8. A *REwLA* r is m -universally bounded if every input string accepted by $\mathcal{A}(r)$ can be accepted by a run $Q_0, Q'_0, \dots, Q_n, Q'_n$ such that we have $\max_{0 \leq i \leq n} |Q'_i| \leq m$.

That is, we can check string acceptance while keeping track of at most m states after epsilon closure in each intermediary step (which means $\mathcal{A}(r)$ can be implemented such that it nondeterministically rejects any run which features more than m states without this changing the accepted language).

Note that all *REwLA* r are m -universally bounded with m equal to the number of states in $\mathcal{A}(r)$ not having epsilon outgoing transitions.

Example 3. The password example in the introduction, here in *PEGs* notation,

$$\& (. * [a-zA-Z]) \& (. * \backslash d) \& (. * [! @ \# \$ () , ; :]) . \{ 8 , \} \$$$

is 4-universally bounded, given that this *REwLA* begins with three positive lookaheads, followed by a main part, and thus at any step a run will contain (at most) one state from each of the three positive lookaheads and one state from the main part. In general, a *REwLA* with k positive lookaheads, with none of the lookaheads in a Kleene star, will be $(k + 1)$ -universally bounded, since at any step a run will contain at most one state from each of the k lookaheads and one state from the main part.

In the final result of this section we state the DFA state complexity for m -universally bounded *REwLA*.

Theorem 4. A minimal (complete) DFA accepting the $\#$ string encoding of an m -universally bounded *REwLA* r with k lookaheads, has a number of states bounded from above by $2^{\sum_{i=1}^m (\|r\|_i + k + 2)}$.

Proof. This result follow from the definition of m -universally boundedness and the argument used in the proof of Theorem 2, to obtain that we need $(\|r\| + k + 2)$ states in $\mathcal{A}(r)$ without epsilon outgoing transitions, if we do not merge the accept states in lookaheads with loops on $\Sigma \cup \{\#\}$, as is done in the proof of Theorem 2. Thus, for NFA

state complexity, at most $\sum_{i=1}^l (\|r\|_i^{k+2})$ states is required, since we have $(\|r\|_i^{k+2})$ number of choices for a set of i states at each step of a run, and therefore the DFA state complexity is at most $2^{\sum_{i=1}^m (\|r\|_i^{k+2})}$ states. \square

5 Empirical Results

Although we implemented our *REwLA* to AFA algorithm in Java, by extending an existing symbolic automata package [D’Antoni 2015], we focus in this short section on how often the features studied in this paper are used by developers.

We used the polyglot regular expression (with lookahead) corpus of [Davis et al. 2019], consisting of 537,806 *REwLA* from 8 different programming languages. Of the 8 programming languages, 6 support lookaheads. Of the 505,455 *REwLA* that belong to programming languages that support lookaheads, positive lookaheads occurred in 1.28% of *REwLA*, while negative lookaheads occurred in 1.23% of the *REwLA*. Of these 12,665 *REwLA* that contain lookaheads, lookaheads used in a nested fashion occurred in only 1.38% of them. Furthermore, we found that in 92% of these 12,665 *REwLA*, lookaheads were not contained in Kleene starred (or plus) subexpressions (empirically motivating the m -universally bounded restriction studied at the end of Section 4).

As far as we know, every previous empirical regular expression study solely focused on regular expressions in the context of full matching. To better understand how regular expression libraries are used and whether substring matching is used in practice as often as we think, we analyzed the usage of the `java.util.regex` package of 68,164 Java projects obtained through RepoReaper, which provides a curated list of GitHub projects. Of the 68,164 projects, the `java.util.regex` package was used in 13,608 of them. Of the 39,264 regular expressions analyzed, 27.66% were used in the functions `String.matches` and `Pattern.matches`, that perform complete matching. We found that 11% of the *REwLA* that were compiled using `Pattern.compile` and then used to find a submatch, were enclosed in the full matching anchors (`^..$`). Whether this was intentional or not is obviously unknown. Furthermore, we found that 47% of the *REwLA* were used for submatching and that for 11.22% of these, repeated submatching was performed.

6 Conclusions and Future Work

We showed how to translate *REwLA* into AFA with ε -transitions followed by an epsilon removal procedure to obtain Boolean automata without ε -transitions from the AFA. The Boolean automata consist of a number of states that is at most 1, 2, or 3 more, depending on the string encoding used, than the length of the corresponding *REwLA*. Our subclass of Boolean (and alternating) automata, in contrast to Boolean automata in general, behaves well under our definition of concatenation (i.e. the size of the concatenated Boolean automaton is the sum of the sizes of the two individual Boolean automata).

For future work, we will investigate *REwLA* in the context of large alphabet sizes, i.e. showing how to extend *REwLA* to the symbolic case, and also consider additional occurring features found in regular expression matching libraries (in conjunction with lookaheads), such as lookbehinds. Furthermore, we will consider the addition of the greedy disambiguation policy to *REwLA*, which at a minimum will partition a lookahead string matched by a *REwLA* in a unique way in its main and lookahead parts. Our state complexity result when converting *REwLA* into DFA, as stated in Theorem 3, deserves

further investigation to see if we can improve on these bounds. Additionally, if we could remove one state from the Boolean state complexity result of Theorem 2 in the case of the \sim string encoding (and perhaps also in the $\#$ string encoding case), then after using the result from Theorem 2 in the proof of Theorem 3, we will end up with a result in Theorem 3 that is better than the corresponding result from [Miyazaki and Minamide 2019], even in the two remaining cases where $\|r\| = 1$ or 2 . Of course, these two cases can be handled separately in an exhaustive way by considering all possible *REwLA* of lengths 1 and 2. Finally, we would like to focus in more detail on the investigation started in the last part of [Section 4] on m -universally bounded *REwLA*.

Acknowledgement

We would like to thank Michal Hospodár for fruitful discussions on how to add ε -transitions to the definition of alternating automata.

References

- [Berglund et al., 2018] Berglund, M., Bester, W., and van der Merwe, B. (2018). Formalising boost posix regular expression matching. In *International Colloquium on Theoretical Aspects of Computing*, pages 99–115.
- [Bojańczyk, 2014] Bojańczyk, M. (2014). Transducers with origin information. In *International Colloquium on Automata, Languages, and Programming*, pages 26–37.
- [Brzozowski and Leiss, 1980] Brzozowski, J. A. and Leiss, E. (1980). On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35.
- [Chandra et al., 1981] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. (1981). Alternation. *J. ACM*, 28(1):114–133.
- [Chida and Kuramitsu, 2017] Chida, N. and Kuramitsu, K. (2017). Linear Parsing Expression Grammars. In Drewes, F., Martín-Vide, C., and Truthe, B., editors, *Language and Automata Theory and Applications*, pages 275–286.
- [Cox, 2010] Cox, R. (2010). Regular Expression Matching in the Wild. <https://swtch.com/~rsc/egexp/regexp1.html> Accessed on 2020-02-07.
- [D’Antoni, 2015] D’Antoni, L. (2015). Library for symbolic automata. <https://github.com/lorisidanto/symbolicautomata> Accessed on 2020-04-13.
- [Davis, 2019] Davis, J. C. (2019). Rethinking Regex engines to address ReDoS. In Dumas, M., Pfahl, D., Apel, S., and Russo, A., editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*, pages 1256–1258.
- [Davis et al., 2018] Davis, J. C., Coghlan, C. A., Servant, F., and Lee, D. (2018). The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, pages 246–256.
- [Davis et al., 2019] Davis, J. C., Michael IV, L. G., Coghlan, C. A., Servant, F., and Lee, D. (2019). Why Aren’t Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*, pages 443–454.

- [Ford, 2004] Ford, B. (2004). Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04, pages 111–122.
- [Hazel, 2015] Hazel, P. (2015). PCRE - Perl Compatible Regular Expressions. <https://www.pcre.org/> Accessed on 2019-11-03.
- [Keeler and Salomaa, 2020] Keeler, C. and Salomaa, K. (2020). Alternating finite automata with limited universal branching. In International Conference on Language and Automata Theory and Applications, pages 196–207. Springer.
- [Kleitman, 1969] Kleitman, D. J. (1969). On Dedekind's problem: The number of monotone boolean functions. *Proc. Amer. Math. Soc.*, 21:677–682.
- [Kozen, 1976] Kozen, D. (1976). On parallelism in turing machines. 17th Annual Symposium on Foundations of Computer Science (sfcs 1976), pages 89–97.
- [M. Hospodár and G. Jirásková, 2018] M. Hospodár and G. Jirásková (2018). The complexity of concatenation on deterministic and alternating finite automata. *RAIROTheor. Inf. Appl.*, 52(2-3-4):153–168.
- [Miyazaki and Minamide, 2019] Miyazaki, T. and Minamide, Y. (2019). Derivatives of Regular Expressions with Lookahead. *Journal of Information Processing*, 27:422–430.
- [Morihiya, 2012] Morihiya, A. (2012). Translation of Regular Expression with Lookahead into Finite State Automaton. *Computer Software*, 29(1):147–158.
- [Sloane, 1964] Sloane, N. (1964). The on-line encyclopedia of integer sequences. <https://oeis.org/A000372> Accessed on 2020-09-25.
- [Staicu and Pradel, 2018] Staicu, C.-A. and Pradel, M. (2018). Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In 27th USENIX Security Symposium (USENIX Security 18), pages 361–376.
- [Stallman, 2008] Stallman, R. (2008). IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004), pages 1–3874.
- [Stockmeyer and Meyer, 1973] Stockmeyer, L. J. and Meyer, A. R. (1973). Word Problems Requiring Exponential Time (Preliminary Report). In Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, STOC '73, pages 1–9.
- [Thompson, 1968] Thompson, K. (1968). Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422.