On Lookaheads in Regular Expressions with Backreferences

Nariyoshi Chida
NTT Corporation, Tokyo, Japan
Waseda University, Tokyo, Japan

Tachio Terauchi
Waseda University, Tokyo, Japan

Abstract

Many modern regular expression engines employ various extensions to give more expressive support for real-world usages. Among the major extensions employed by many of the modern regular expression engines are *backreferences* and *lookaheads*. A question of interest about these extended regular expressions is their expressive power. Previous works have shown that (i) the extension by lookaheads does not enhance the expressive power, i.e., the expressive power of regular expressions with lookaheads is still regular, and that (ii) the extension by backreferences enhances the expressive power, i.e., the expressive power of regular expressions with backreferences (abbreviated as *rewb*) is no longer regular. This raises the following natural question: Does the extension of regular expressions with backreferences by lookaheads enhance the expressive power of regular expressions with backreferences? This paper answers the question positively by proving that adding either positive lookaheads or negative lookaheads increases the expressive power of rewb (the former abbreviated as *rewbl*$_p$ and the latter as *rewbl*$_n$). A consequence of our result is that neither the class of finite state automata nor that of memory automata (MFA) of Schmid [14] (which corresponds to regular expressions with backreferenes but without lookaheads) corresponds to rewbl$_p$ or rewbl$_n$. To fill the void, as a first step toward building such automata, we propose a new class of automata called *memory automata with positive lookaheads* (PLMFA) that corresponds to rewbl$_p$. The key idea of PLMFA is to extend MFA with a new kind of memories, called *positive-lookahead memory*, that is used to simulate the backtracking behavior of positive lookaheads. Interestingly, our positive-lookahead memories are almost perfectly *symmetric* to the capturing-group memories of MFA. Therefore, our PLMFA can be seen as a natural extension of MFA that can be obtained independently of its original intended purpose of simulating rewbl$_p$.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Regular expressions, Lookaheads, Backreferences, Memory automata

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.15

Funding This work was supported by JSPS KAKENHI Grant Numbers 17H01720, 18K19787, 20H04162, 20K20625, and 22H03570.

1 Introduction

Regular expressions, introduced by Kleene [9], and the extensions employed by many of the modern regular expression engines are widely studied in formal language theory. Among the major extensions are *backreferences* and *lookaheads*. Previous works on formal language theory have studied the two features mostly in isolation. Morihata [12] and Berglund et al. [3] showed that extending regular expressions by lookaheads does not enhance their expressive power. Their proofs are by a translation to boolean finite automata [4] whose expressive power is regular. The formal study of regular expressions with backreferences (rewb) dates back to the seminal work by Aho [1]. More recently, a formal semantics and a pumping lemma were given by Câmpeanu et al. [5], and Berglund and van der Merwe [2] showed that different variants of backreference semantics give rise to differences in expressive powers. Schmid [14] proposed *memory automata* (MFA) and showed that the expressive power of the automata is equivalent to that of rewb.

In this paper, we initiate a formal study of *regular expression with backreferences and lookaheads* (*rewbl* for short). We call the fragment containing only positive (resp. negative) lookaheads *rewbl$_p$* (resp. *rewbl$_n$*). We show that both rewbl$_p$ and rewbl$_n$ are more expressive than rewb, and also prove some language-theoretic properties of rewbl. One consequence of the results is the undecidability of a problem tackled in a recent work [11].

Another consequence of our results is that neither the class of finite state automata nor that of memory automata (MFA) of Schmid [14] (which corresponds to regular expressions with backreferences but without lookaheads) corresponds to rewbl$_p$ or rewbl$_n$. As remarked above, prior works [3, 12] have applied translation to boolean finite automata [4] (or alternating finite automata [7]) to build automata equivalent to regular expressions with lookaheads. They simulate lookaheads by executing multiple runs simultaneously without backtracking. Unfortunately, the interaction of lookaheads with backreferences prevents us from applying the approaches to rewbl. Namely, rewbl permits *cross-lookahead backreferences* whereby a string captured outside of a lookahead is referred from inside of the lookahead, or vice versa (only the former is allowed for negative lookaheads whereas both are allowed for positive lookaheads). Such cross-lookahead backreferences intrinsically require backtracking. In our work, as a first step toward building automata equivalent to rewbl, we introduce a new class of automata called *memory automata with positive lookaheads* (PLMFAs). We prove that PLMFAs are equivalent to rewbl$_p$ in expressive power. A key component of PLMFAs is a new kind of memories, called a *positive-lookahead memory*, that is used to simulate the backtracking behavior of positive lookaheads. Interestingly, our positive-lookahead memories are almost perfectly *symmetric* to the capturing-group memories of MFA. Therefore, our PLMFA can be seen as a natural extension of MFA that can be obtained independently of its original intended purpose of simulating rewbl$_p$.

In summary, this paper makes the following contributions:

- We show that the extension of rewb by either positive or negative lookaheads enhances the expressive power. Additionally, we prove some language-theoretic properties of rewbl. (Sec. 3)
- We introduce memory automata with positive lookaheads (PLMFAs), a new class of automata that we prove to be equivalent in expressive power to rewbl$_p$. A key component of PLMFAs is a new kind of memories called positive-lookahead memory, which is almost perfectly symmetric to capturing-group memory of MFA. (Sec. 4)

We believe that our work leads to interesting future developments in both theoretical and practical fronts: interesting practically because backreferences and lookaheads are practically motivated by real-world needs, and interesting theoretically because, as we shall show, rewbl does not appear to correspond to any known formal language classes.

2 Preliminaries

In this section, we introduce the preliminary notations (Sec. 2.1) and present the syntax and the semantics of rewbl (Sec. 2.2).

2.1 Notation

We write $\mathbb{N}$ for the set of natural numbers and $[i]$ for the set $\{1, 2, \ldots, i\}$ where $i \in \mathbb{N}$. For a sequence l, we write $|l|$ for its length, $l[i]$ (for $1 \leq i \leq |l|$) for its ith element, $l[i..j]$ for the sub-sequence from the ith element to the jth element (for $1 \leq i \leq j \leq |l|$). We write $l_1 :: l_2$ for the concatenation of l_1 and l_2. We abbreviate it as $l_1 l_2$ if clear from the context. We write $v \in l$ to denote that l contains v. We write Σ for a finite alphabet; $a, b \in \Sigma$ for a

r	$::=$	a	character	$\mid$	r^*	repetition
	$\mid$	$\emptyset$	empty set	$\mid$	$({}_i r)_i$	capturing group
	$\mid$	ϵ	empty string	$\mid$	$\backslash i$	backreference
	$\mid$	rr	concatenation	$\mid$	$(?{=}r)$	pos-lookahead
	$\mid$	$r\|r$	union	$\mid$	$(?!r)$	neg-lookahead

Figure 1 The syntax of rewbl expressions.

character; $x, y \in \Sigma^*$ for a sequence of characters (i.e., *string*); ϵ for the empty string; Σ_ϵ for $\Sigma \cup \{\epsilon\}$; In what follows, we fix a finite alphabet Σ. For $1 \leq i < j \leq |x|$, we define $x[i..j)$ to be $x[i...j-1]$. For $x, y \in \Sigma^*$, we define $x\backslash y$ to be the left quotient of x divided by y, i.e., v where $yv = x$. Dually, the right quotient of x divided y, x/y, is v where $vy = x$. $S \subset U$ denotes that S is a proper subset of U, i.e., $S \subseteq U \wedge S \neq U$. For a partial map f, we write $dom(f)$ for the domain of f. $\mathcal{P}(S)$ denotes that the power set of a set S. For f a (partial) function, $f[\alpha \mapsto \beta]$ denotes the (partial) function that maps α to β and behaves as f for all other arguments. We write $f(\alpha) = \bot$ if f is undefined at α.

2.2 Regular Expressions with Backreferences and Lookaheads

The syntax of *regular expressions with backreferences and lookaheads* (*rewbl*) is given by Fig. 1. The semantics of the pure regular expression constructs (i.e., the first six constructs of Fig. 1) is standard. We write $\cdot$ for the rewbl that matches any character (i.e., $a_1|\dots|a_n$ where $\Sigma = \{a_1, \dots, a_n\}$). The precedence order of the operators is as follows: Kleene-*, concatenation, and union. The left has a higher precedence. For example, the expression $a^*|bc^*$ means $((a^*)|(b(c^*)))$ due to the priority.

The remaining constructs, i.e., capturing groups, backreferences, and lookaheads, are the extensions considered in this paper. In conformance with the nomenclature from the literature [10], we call the fragment of rewbl without lookaheads *rewb*. We call the fragment of rewbl without negative (resp. positive) lookaheads *rewbl*$_p$ (resp. *rewbl*$_n$). In what follows, we explain the semantics of the extended features informally in terms of the standard backtracking-based matching algorithm which attempts to match the given regular expression with the given string and backtracks when the attempt fails. A *capturing group* $({}_i r)_i$ (or $(r)_i$ if no ambiguity arises) attempts to match r, and if successful, stores the matched substring in the storage identified by the index i. Otherwise, the match fails and the algorithm backtracks. A *backreference* $\backslash i$ refers to the substring matched to the corresponding capturing group $({}_i r)_i$, and attempts to match the same substring if the capture had succeeded. If the capture had not succeeded, i.e., is an *unassigned backreference*, or the matching against the captured substring fails, then the algorithm backtracks. Capturing groups in practice often do not have explicit indexes, but we write them here for readability. A *positive* (resp. *negative*) *lookahead* $(?{=}r)$ (resp. $(?!r)$) attempts to match r without any character consumption, proceeds if the match succeeds (resp. fails), and backtracks otherwise.

More formally, the semantics is defined by the *matching relation* $\rightsquigarrow$ that models the behavior of backtracking matching algorithms. The full rules for deriving the matching relation $\rightsquigarrow$ is shown in Fig. 2. The semantics is same as the one defined in our recent work [8] except for specializing the set-of-characters rules to the rules for a single character and the empty set. We define $ite(true, A, B) = A$ and $ite(false, A, B) = B$.

A matching relation is of the form $(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}$ where p is a position on the string w such that $1 \leq p \leq |w| + 1$, Λ, called an *environment*, is a function that maps each capturing group index to a string captured by the corresponding capturing group, and $\mathcal{N}$ is a set of

$$\frac{p \le |w| \qquad w[p] = a}{(a, w, p, \Lambda) \rightsquigarrow \{(p+1, \Lambda)\}}\ (\textsc{Character})$$

$$\frac{p > |w| \lor w[p] \neq a}{(a, w, p, \Lambda) \rightsquigarrow \emptyset}\ (\textsc{Character Failure})$$

$$\frac{}{(\emptyset, w, p, \Lambda) \rightsquigarrow \emptyset}\ (\textsc{Empty Set})$$

$$\frac{}{(\epsilon, w, p, \Lambda) \rightsquigarrow \{(p, \Lambda)\}}\ (\textsc{Empty String})$$

$$\frac{(r_1, w, p, \Lambda) \rightsquigarrow \mathcal{N} \qquad \forall (p_i, \Lambda_i) \in \mathcal{N},\ (r_2, w, p_i, \Lambda_i) \rightsquigarrow \mathcal{N}_i}{(r_1 r_2, w, p, \Lambda) \rightsquigarrow \bigcup_{0 \le i < |\mathcal{N}|} \mathcal{N}_i}\ (\textsc{Concatenation})$$

$$\frac{(r_1, w, p, \Lambda) \rightsquigarrow \mathcal{N} \qquad (r_2, w, p, \Lambda) \rightsquigarrow \mathcal{N}'}{(r_1 | r_2, w, p, \Lambda) \rightsquigarrow \mathcal{N} \cup \mathcal{N}'}\ (\textsc{Union})$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N} \quad \forall (p_i, \Lambda_i) \in (\mathcal{N} \backslash \{(p, \Lambda)\}),\ (r^*, w, p_i, \Lambda_i) \rightsquigarrow \mathcal{N}_i}{(r^*, w, p, \Lambda) \rightsquigarrow \{(p, \Lambda)\} \cup \bigcup_{0 \le i < |(\mathcal{N} \backslash \{(p, \Lambda)\})|} \mathcal{N}_i}\ (\textsc{Repetition})$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}}{((r)_j, w, p, \Lambda) \rightsquigarrow \{(p_i, \Lambda_i[j \mapsto w[p..p_i)]) \mid (p_i, \Lambda_i) \in \mathcal{N}\}}\ (\textsc{Capturing Group})$$

$$\frac{\Lambda(i) \neq \bot \qquad (\Lambda(i), w, p, \Lambda) \rightsquigarrow \mathcal{N}}{(\backslash i, w, p, \Lambda) \rightsquigarrow \mathcal{N}}\ (\textsc{Backreference})$$

$$\frac{\Lambda(i) = \bot}{(\backslash i, w, p, \Lambda) \rightsquigarrow \emptyset}\ (\textsc{Backreference Failure})$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}}{((?{=}r), w, p, \Lambda) \rightsquigarrow \{(p, \Lambda') \mid (\_, \Lambda') \in \mathcal{N}\}}\ (\textsc{Positive Lookahead})$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = \mathit{ite}(\mathcal{N} \neq \emptyset, \emptyset, \{(p, \Lambda)\})}{((?!r), w, p, \Lambda) \rightsquigarrow \mathcal{N}'}\ (\textsc{Negative Lookahead})$$

Figure 2 Rules of the matching relation $\rightsquigarrow$.

matching results. A *matching result* is a pair of a position and an environment. Roughly, $(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}$ is read: a rewbl expression r tries to match the string w from the position p, with the environment Λ and, if $(p', \Lambda') \in \mathcal{N}$, r consumed $p' - p$ characters and updated the environment to Λ'. Additionally, if $\mathcal{N} = \emptyset$, it means that the matching failed.

In the two rules for a character, the rewbl a tries to match the string w at the position p with the function capturing Λ. If the pth character $w[p]$ is a, then the matching succeeds returning the matching result $(p+1, \Lambda)$ (Character). Otherwise, the character $w[p]$ does not match or the position is at the end of the string, and $\emptyset$ is returned as the matching result indicating the match failure (Character Failure).

The rules (Empty Set), (Empty String), (Concatenation), (Union) and (Repetition) are self explanatory. Note that we avoid self looping in (Repetition) by not repeating the match from the same position.

In the rule (Capturing Group), we first get the matching result $\mathcal{N}$ from matching w against r at the current position p. And for each matching result $(p_i, \Lambda_i) \in \mathcal{N}$ (if any), we record the matched substring $w[p..p_i)$ in the corresponding environment Λ_i at the index i. The rule (Backreference) looks up the captured substring and tries to match it with the input at the current position. The match fails if the corresponding capture has failed as stipulated by the rule (Backreference Failure).

In the rule (Positive Lookahead), the expression r is matched against the given string w at the current position p to obtain the matching results $\mathcal{N}$. Then, for every match result $(p', \Lambda') \in \mathcal{N}$ (if any), we reset the position from p' to p. This models the behavior of lookaheads which does not consume the string. The rule (Negative Lookahead) is similar, except that we reset and proceed when there is no match. Note that captures made inside of a negative lookahead cannot be referred outside of the lookahead, which agrees with the behavior of regular expression engines in practice.

▶ **Definition 1** (Language). The *language* of a rewbl r is defined as $L(r) = \{w \mid (r, w, 1, \emptyset) \rightsquigarrow \mathcal{N} \land \exists \Lambda. (|w| + 1, \Lambda) \in \mathcal{N}\}$.

Recall that one subtle aspect of rewbl is that backreferences can cross lookahead boundaries (cf. Sec. 1). We next show some examples of cross-lookahead backreferences.

▶ **Example 2.** Consider the expression $(_1.^*\mathtt{z})_1(?{=}\backslash 1).^*$. Its language is $\{xzxzy \mid x, y \in \Sigma^*\}$. For example, when the input string is `azazbc`, the expression captures the prefix `az` and refers it from the inside of the positive lookahead `(?=\1)`.

▶ **Example 3.** Consider the expression $\texttt{(}_1\cdot\texttt{)}_1\texttt{(?!\textbackslash 1)}\cdot^*$. The language is $\{a \mid a \in \Sigma\} \cup \{abx \mid a, b \in \Sigma \wedge a \neq b \wedge x \in \Sigma^*\}$.

▶ **Example 4.** Consider the expression $\texttt{(?=(}_1\cdot^*\texttt{)}_1\texttt{z)}\cdot\texttt{\textbackslash 1}$. The language is $\{\texttt{z}x \mid x \in \{\texttt{z}\}^*\}$.

Example 2 (resp. 3) shows an example where a string captured outside of a positive (resp. negative) lookahead is backreferenced in the lookahead. Example 4 shows an example where a string captured inside of a positive lookahead is backreferenced from outside of the lookahead.

2.2.1 Conventions on Syntax and Semantics

We review the conventions regarding capturing groups and unassigned references. The conventions are proposed in prior works on rewb, and as shown by [2], they affect the expressive power of rewb. Here, we simply present the conventions and refer interested readers to [2] for the expressive power differences.

There are two convensions regarding capturing groups: *no label repetitions (NLR)* and *may repeat labels (MRL)*. NLR requires the indexes of capturing groups to be distinct, whereas MRL imposes no such restrictions. For example, $\texttt{(}\cdot^*\texttt{)}_1\texttt{\textbackslash 1}$ satisfies NLR, but $\texttt{((}\cdot^*\texttt{)}_1\texttt{|(}\cdot^*\texttt{)}_1\texttt{)\textbackslash 1}$ does not because the capturing group with index 1 appears twice. NLR is assumed in the prior works by Câmpeanu et al. [5] and Carle and Narendran [6] on the expressive power of rewb.

There are two conventions regarding unassigned references: the ϵ *semantics* and the $\emptyset$ *semantics*. The ϵ (resp. $\emptyset$) semantics defines that unassigned references are handled as an empty string ϵ (resp. a failure $\emptyset$). For example, for $r = \texttt{a\textbackslash 1}$, $L(r) = \{\texttt{a}\}$ with the ϵ semantics but $L(r) = \emptyset$ with the $\emptyset$ semantics. Additionally, prior works have proposed a condition called *no unassigned reference (NUR)*. The NUR condition does not allow unassigned references in expressions, i.e., all expressions with unassigned references are to be excluded (see below for the formal definition). For example, $r = \texttt{a\textbackslash 1}$ does not satisfy the NUR condition because $\texttt{\textbackslash 1}$ is an unassigned references. Note that the ϵ semantics and the $\emptyset$ semantics coincide under the NUR condition because there would be no unassigned references. The condition is also assumed in [5, 6] ([6] incorrectly remarks that [5] does not assume the condition).

In the rest of this section, we give a formal definition of the NUR condition that we shall also use later in our proofs. We note that prior works that proposed the condition did not provide a formal definition of it [5, 6]. First, we define the function *Capture* from rewbls to the set of capturing group indexes that can be referred from their continuations:

$$
\mathit{Capture}(r) = \begin{cases}
\emptyset & (\text{if } r = a, \emptyset, \epsilon, r_1^*, \backslash i, \text{ or } (?!r_1)) \\
\mathit{Capture}(r_1) \cup \mathit{Capture}(r_2) & (\text{if } r = r_1 r_2) \\
\mathit{Capture}(r_1) \cap \mathit{Capture}(r_2) & (\text{if } r = r_1 | r_2) \\
\mathit{Capture}(r_1) \cup \{i\} & (\text{if } r = (_i r_1)_i) \\
\mathit{Capture}(r_1) & (\text{if } r = (?{=}r_1))
\end{cases}
$$

With this, we can define the predicate NUR(S,r) that says that r satisfies NUR condition if it occurs in a context where the capturing group indexes in S can be referred:

$$
\mathrm{NUR}(S, r) = \begin{cases}
\mathit{true} & (\text{if } r = a, \emptyset, \text{ or } \epsilon) \\
\mathrm{NUR}(S, r_1) \wedge \mathrm{NUR}(S \cup \mathit{Capture}(r_1), r_2) & (\text{if } r = r_1 r_2) \\
\mathrm{NUR}(S, r_1) \wedge \mathrm{NUR}(S, r_2) & (\text{if } r = r_1 | r_2) \\
\mathrm{NUR}(S, r_1) & (\text{if } r = r_1^*, (_i r_1)_i, (?{=}r_1), \text{ or } (?!r_1)) \\
i \in S & (\text{if } r = \backslash i)
\end{cases}
$$

Then, r can be said to satisfy the NUR condition iff NUR($\emptyset$, r) = *true*. For example, the expression r = `(`$_1$`a)`$_1$`\1` satisfies the NUR condition because NUR($\emptyset$, r) = NUR($\emptyset$, `(`$_1$`a)`$_1$) $\wedge$ NUR($\emptyset \cup$ *Capture*(`(`$_1$`a)`$_1$), `\1`) = NUR($\emptyset$, `a`) $\wedge$ NUR($\{1\}$, `\1`) = *true*. As another example, r = `(`$_1$`a\1)`$_1$ does not satisfy the NUR condition because NUR($\emptyset$, r) = NUR($\emptyset$, `a\1`) = NUR($\emptyset$, `a`) $\wedge$ NUR($\emptyset \cup$ *Capture*(`a`), `\1`) = *false*. In what follows, unless explicitly stated otherwise, we assume that a rewbl that satisfies NLR and NUR.

3 Language Properties of Rewbl

In this section, we prove some salient language properties of rewbl. Importantly, we show that both rewbl$_p$ and rewbl$_n$ is strictly more expressive than rewb, thus showing that the extension by either posistive or negative lookaheads changes the expressive power of rewb. In the following, we denote by $\mathbb{L}_B$, $\mathbb{L}_{BL}$, $\mathbb{L}_{BL_n}$, and $\mathbb{L}_{BL_p}$ the class of languages matched by rewb, rewbl, rewbl$_n$, and rewbl$_p$, respectively.

Our first result states that $\mathbb{L}_{BL_n}$ is closed under union, intersection, and complement.

▶ **Theorem 5.** $\mathbb{L}_{BL_n}$ *is closed under union, intersection, and complement.*

Proof. Suppose we have rewbl expressions r_1 and r_2. Then, rewbl expressions that accept the union of $L(r_1)$ and $L(r_2)$, the intersection of $L(r_1)$ and $L(r_2)$, and the complement of $L(r_1)$ can be constructed respectively as follows.

- **Union:** $L(r_1) \cup L(r_2) = L(r_1|r_2)$;
- **Intersection:** $L(r_1) \cap L(r_2) = L((?!(?!r_1(?!\cdot)))r_2)$; and
- **Complement:** $L(r_1)^c \triangleq \Sigma^* \backslash L(r_1) = L((?!r_1(?!\cdot))\cdot^*)$.

In **Intersection** and **Complement**, a subtle point is that a negative lookahead $(?!r)$ accepts a string even if the expression r rejects only a prefix of the string. For example, $L((?!(?!r_1))r_2)$ is the set of strings in $L(r_2)$ that have a prefix that belongs to $L(r_1)$, rather than the intersection of $L(r_1)$ and $L(r_2)$. To force whole matching, the negative lookahead $(?!\cdot)$ is appended. ◀

Of course, we could alternatively show **Intersection** from **Union** and **Complement** by applying De Morgan's laws: $L(r_1) \cap L(r_2) = (L(r_1)^c \cup L(r_2)^c)^c$. The above proof gives a direct construction which shows that the intersection can be obtained by a short rewbl$_n$ expression.

We next show that $\mathbb{L}_{BL}$ is also closed under union, intersection, and complement.

▶ **Theorem 6.** $\mathbb{L}_{BL}$ *is closed under union, intersection, and complement.*

Proof. The proof is the same as Theorem 5. Or, for **Intersection**, an even shorter proof is possible: $L(r_1) \cap L(r_2) = L((?{=}r_1(?!\cdot))r_2)$. ◀

A consequence of Theorem 5 is that rewbl and rewbl$_n$ are more expressive than rewb.

▶ **Corollary 7.** $\mathbb{L}_B \subset \mathbb{L}_{BL_n} \subseteq \mathbb{L}_{BL}$.

Proof. Immediate from Theorem 5 and Lemma 3 of [6] which showed that rewb is not closed under intersection. ◀

We show that adding just positive lookaheads also increases the expressive power of rewb.

▶ **Theorem 8.** $\mathbb{L}_B \subset \mathbb{L}_{BL_p}$.

Table 1 Summary of closure properties. The rows highlighted in gray present our new results. ? indicates that the problem is open.

	Closure under				
	$\cup$	$\cap$	Complement	Concatenation	Kleene-*
Regular	Yes	Yes	Yes	Yes	Yes
$\mathbb{L}_B$	Yes	No	No	Yes	Yes
$\mathbb{L}_{BL_p}$	Yes	?	?	Yes	Yes
$\mathbb{L}_{BL_n}$	Yes	Yes	Yes	Yes	Yes
$\mathbb{L}_{BL}$	Yes	Yes	Yes	Yes	Yes

Proof. From [6], the language $S = \{a^iba^{i+1}ba^k \mid k = i(i+1)k', k' > 0, \text{ and } i > 0\}$ is not in $\mathbb{L}_B$. Let $S' = \{a^iba^{i+1}ba^kc \mid k = i(i+1)k', k' > 0, \text{ and } i > 0\}$. We can prove that S' is also not in $\mathbb{L}_B$ in a manner similar to the proof that S is not in $\mathbb{L}_B$ [6].

Now, S' is the intersection of $L(r_1c)$ and $L(r_2c)$ where r_1 and r_2 are $(aa^*)_1b\backslash 1ab\backslash 1\backslash 1^*$ and $(aa^*)_1b(\backslash 1a)_2b\backslash 2\backslash 2^*$, respectively. Then, the intersection of the languages of r_1c and r_2c is $L((?{=}r_1c)r_2c) = S' \in \mathbb{L}_{BL_p}$. ◀

A summary of the results of closure properties can be found in Table 1. Additionally, the diagram below summarizes the results of the hierarchy of the language classes. Here, $\mathbb{L}_L$ denotes the class of languages matched by regular expressions with (both positive and negative) lookaheads, which is known to be equivalent in expressive power to the set of regular languages [3, 12].

$$\begin{tikzcd}
 & & & \mathbb{L}_{BL_p} \arrow[dr, phantom, "\subseteq" sloped] & \\
\text{Regular} \arrow[r, phantom, "="] & \mathbb{L}_L \arrow[r, phantom, "\subset"] & \mathbb{L}_B \arrow[ur, phantom, "\subset" sloped] \arrow[dr, phantom, "\subset" sloped] & & \mathbb{L}_{BL} \\
 & & & \mathbb{L}_{BL_n} \arrow[ur, phantom, "\subseteq" sloped] &
\end{tikzcd}$$

From Theorems 5 and 6, we also obtain the following result.

▶ **Theorem 9.** *The emptiness problems of rewbl and rewbl$_n$ are undecidable.*

Proof. Suppose for contradiction that the emptiness problem of rewbl is decidable. By Theorem 6, we know that the language of rewbl is closed under intersection. Therefore, the emptiness problem of intersection of two rewb expressions is also decidable. However, this contradicts a result from [6] which states that the latter problem is undecidable. The proof for rewbl$_n$ is similar by using Theorem 5. ◀

A recent work [11] has proposed a method for symbolically executing programs containing rewbl. Their method generates and tries to solve constraints of the form $x \in L(r)$ where r is a rewbl expression and x is a variable for which the method tries to find an assignment that satisfies the constraint. Theorem 9 implies that their constraint solving problem is undecidable.

▶ **Corollary 10.** *The constraint solving problem of [11] is undecidable.*

▶ Remark 11. The constructions used to show Cor. 7 and Theorem 8 do not contain backreferences that cross lookahead boundaries (recall the discussion from Sec. 1 and Sec. 2.2). Thus, our results show that lookaheads enhance the expressive power of rewb even without cross-lookahead backreferences. We leave for future work to investigate whether there are expressive power changes from allowing or disallowing cross-lookahead backreferences.

3.1 Restricted Label Repetitions

As pointed out in [2], allowing rewbs to repeat labels of backreferences affects their expressive powers. In this sub-section, we introduce new conditions called *restricted may repeat labels* (RMRL) and *no self-capturing reference* (NSR). RMRL allows repeating the labels but only in a restricted way. NSR enforces that there is no reference that is nested by the capturing group of the same index. We use RMRL and NSR as an intermediary in the construction of automata equivalent in expressive power to rewbl$_p$ (with NUR and NLR) in the next section. But the new conditions may also be of independent interest.

Informally, RMRL requires the capturing group that is referred to by any reference is uniquely determined. For example, $({}_1a)_1({}_1b)_1\backslash 1$ satisfies RMRL because the reference $\backslash 1$ refers to the capturing group $({}_1b)_1$ while $(({}_1a)_1|({}_1b)_1)\backslash 1$ does not satisfy RMRL because the reference $\backslash 1$ can refer to the capturing groups $({}_1a)_1$ and $({}_1b)_1$. To represent the repetitions of indexes, we use a *multiset*. A multiset, denoted by $\{\!\{\cdots\}\!\}$, is a collection of elements with repetitions. For example, $\{\!\{a,a,b\}\!\}$ is a multiset that has two a's and one b. We use the notation for sets as that of multisets, e.g., we use $\cup$ for the union of multisets, e.g., $\{\!\{a\}\!\} \cup \{\!\{a,b\}\!\} = \{\!\{a,a,b\}\!\}$, and $|\cdot|$ for the number of elements, e.g., $|\{\!\{a,a,b\}\!\}| = 3$. Formally, we define RMRL by first defining *NumCaps* that takes a multiset $\mathbb{S}$ and a rewbl expression r and returns a multiset that represents the number of ways to capture for each index.

$$
\mathit{NumCaps}(\mathbb{S},r) = \begin{cases} \mathbb{S} & (\text{if } r = a, \emptyset, \epsilon, \backslash i, \text{ or } (?!r_1)) \\ \mathit{NumCaps}(\mathit{NumCaps}(\mathbb{S},r_1),r_2) & (\text{if } r = r_1 r_2) \\ \mathit{NumCaps}(\mathbb{S},r_1) \cup \mathit{NumCaps}(\mathbb{S},r_2) & (\text{if } r = r_1|r_2) \\ \{\!\{j \mid j \in \mathit{NumCaps}(\mathbb{S},r_1) \wedge j \neq i\}\!\} \cup \{\!\{i\}\!\} & (\text{if } r = ({}_i r_1)_i) \\ \mathit{NumCaps}(\mathbb{S},r_1) & (\text{if } r = (?{=}r_1) \text{ or } r_1^*) \end{cases}
$$

With this, we define RMRL($\mathbb{S}$,r) that says that r satisfies the RMRL condition if it occurs in a context where the capturing group indexes in $\mathbb{S}$ can be referred:

$$
\mathrm{RMRL}(\mathbb{S},r) = \begin{cases} \mathit{true} & (\text{if } r{=}a, \emptyset, \text{ or } \epsilon) \\ \mathrm{RMRL}(\mathbb{S},r_1) {\wedge} \mathrm{RMRL}(\mathit{NumCaps}(\mathbb{S},r_1),r_2) & (\text{if } r{=}r_1 r_2) \\ \mathrm{RMRL}(\mathbb{S},r_1) {\wedge} \mathrm{RMRL}(\mathbb{S},r_2) & (\text{if } r{=}r_1|r_2) \\ \mathrm{RMRL}(\mathbb{S},r_1) & (\text{if } r{=}r_1^*, ({}_i r_1)_i, (?{=}r_1), \text{ or } (?!r_1)) \\ |\{\!\{i \mid i \in \mathbb{S}\}\!\}| \leq 1 & (\text{if } r{=}\backslash i) \end{cases}
$$

We say that a rewbl expression r satisfies RMRL iff $\mathrm{RMRL}(\emptyset, r) = \mathit{true}$.

Next, we explain NSR. NSR requires that for every reference $\backslash i$, the reference is not nested by the capturing group whose index is i. For example, $({}_1a\backslash 1)_1({}_2\backslash 1)_2$ does not satisfy NSR because $\backslash 1$ appears in its capturing group. Formally, we define NSR as follows.

$$
\mathrm{NSR}(S,r) = \begin{cases} \mathit{true} & (\text{if } r = a, \emptyset, \text{ or } \epsilon) \\ \mathrm{NSR}(S,r_1) \wedge \mathrm{NSR}(S,r_2) & (\text{if } r = r_1 r_2 \text{ or } r_1|r_2) \\ \mathrm{NSR}(S,r_1) & (\text{if } r = r_1^*, (?{=}r_1), \text{ or } (?!r_1)) \\ \mathrm{NSR}(S \cup \{i\},r_1) & (\text{if } r = ({}_i r_1)_i) \\ i \notin S & (\text{if } r = \backslash i) \end{cases}
$$

We say that a rewbl expression r satisfies NSR iff $\mathrm{NSR}(\emptyset, r) = \mathit{true}$. We show that RMRL $\wedge$ NSR is equivalent to NLR in expressive powers.

▶ **Lemma 12.** *(1) For any NLR r there exists a NSR and RMRL r' such that $L(r) = L(r')$, and (2) for any NSR and RMRL r there exists a NLR rewbl r' such that $L(r) = L(r')$.*

Proof. (1) is immediate since NLR implies both RMRL and NSR (under the NUR assumption). To see (2), if r satisfies RMRL and NSR, then capturing groups that are referred are uniquely determined and are closed when they are referred. Thus, we can construct r' by replacing indexes of reference i in r and the capturing group referred to i with unique indexes and removing all unreferred capturing groups. ◀

4 Memory Automata with Positive Lookaheads

This section present PLMFA, a new class of automata that we prove to be equivalent to rewbl$_p$. PLMFA is obtained by extending MFA of Schmid [14] that is equivalent to rewb. The key extension is the addition of a new kind of memories called *positive-lookahead memories*. Roughly, a PLMFA is a non-deterministic finite state automata augmented with a list of capturing-group memories and a list of positive-lookahead memories. The former also exists in MFA and stores strings captured by capturing groups to simulate the behavior of backreferences. The latter stores strings matched by positive lookaheads and is used to simulate the behavior of positive lookaheads.

4.1 Formal Definition

A *memory* is a tuple $(x, \boldsymbol{s})$ of a string $x \in \Sigma^*$ and a *status* $\boldsymbol{s}$. A status is either *open* ($\boldsymbol{O}$) or *close* ($\boldsymbol{C}$). The statuses are changed by *memory instructions* (*instructions* for short) $\Theta = \{\mathtt{o}, \mathtt{c}, \diamond\}$ as follows: $\boldsymbol{s} \oplus \mathtt{o} = \boldsymbol{O}$, $\boldsymbol{s} \oplus \mathtt{c} = \boldsymbol{C}$, and $\boldsymbol{s} \oplus \diamond = \boldsymbol{s}$. Roughly, $\boldsymbol{O}$ means that the string in the memory is modified by appending consumed strings, while $\boldsymbol{C}$ means that the string in the memory is unmodified. Changing the status from $\boldsymbol{C}$ to $\boldsymbol{O}$ (resp. from $\boldsymbol{O}$ to $\boldsymbol{C}$) representing to an entering (resp. exiting) a capturing group if the memory is a capturing-group memory and otherwise (i.e., if positive-lookahead memory) a positive lookahead. At computation steps corresponding to backreferences, the strings in capturing group memories are used to left-divide the input string and appended to strings stored in any open memories. Symmetrically, when the strings in positively lookahead memories are used, they are prepended to the input string and used to right-divide strings stored in any open memories. A positive lookahead memory is used when it gets closed. For a memory $t = (x, \boldsymbol{s})$, we write $t.\mathit{word}$ for x and $t.\mathit{status}$ for $\boldsymbol{s}$. We define PLMFAs as follows.

▶ **Definition 13** (PLMFA). For $(k_c, k_p) \in \mathbb{N}^2$, a (k_c, k_p)*-memory automaton with positive lookaheads*, PLMFA(k_c, k_p), is a tuple (Q, δ, q_0, F) such that
1. Q is a finite set of *states*,
2. $\delta : Q \times (\Sigma_\epsilon \cup [k_c]) \to \mathcal{P}(Q \times \Theta^{k_c} \times \Theta^{k_p})$ is the *transition function*,
3. $q_0 \in Q$ is the *initial state*, and
4. $F \subseteq Q$ is the set of *accepting states*.

Here, k_c and k_p represent the number of capturing-group memories and positive-lookahead memories, respectively. Next, we define *configurations* of PLMFAs.

▶ **Definition 14** (Configuration). A *configuration* of a PLMFA M is a tuple $(q, w, \sigma_c, \sigma_p)$ where q is a state of M, w is an input string, and σ_c (resp. σ_p) is a list of memories that represents a list of capturing-group (resp. positive-lookahead) memories.

▶ **Definition 15** (Computation step). For a PLMFA(k_c, k_p) M and $\ell \in \Sigma_\epsilon \cup [k_c]$, a *step of computation* of M is a binary relation on configurations $\xrightarrow[M]{\ell}$ (or $\xrightarrow{\ell}$ or $\rightarrow$ if irrelevant), defined as follows: $(q, w, \sigma_c, \sigma_p) \xrightarrow[M]{\ell} (q', w', \sigma_c', \sigma_p')$ iff there is a transition $\delta(q, \ell) \ni (q', ir_c, ir_p)$ satisfying the following conditions.

(1) If $\ell \in \Sigma_\epsilon$, $v = \ell$ and otherwise (i.e., if $\ell \in [k_c]$) if $\sigma_c[\ell].status = \boldsymbol{C}$ then $v = \sigma_c[\ell].word$.

(2) $\forall \tau \in \{c, p\}.\forall i \in [k_\tau].\sigma'_\tau[i].status = \sigma_\tau[i].status \oplus ir_\tau[i]$.

(3) $\forall \tau \in \{c, p\}.\forall i \in [k_\tau]$.

$$\sigma'_\tau[i].word = \begin{cases} u :: v & (\text{if } \boldsymbol{O} \Rightarrow_{\tau,i} \boldsymbol{O}) \\ \sigma_\tau[i].word & (\text{if } \_ \Rightarrow_{\tau,i} \boldsymbol{C}) \\ v & (\text{if } \boldsymbol{C} \Rightarrow_{\tau,i} \boldsymbol{O}) \end{cases}$$

where $\sigma_\tau[i].status \Rightarrow_{\tau,i} \sigma'_\tau[i].status$, $bts = \sigma_p[j].word$ where $j = argmax_{j\in J}|\sigma_p[j].word|$ if $J = \{j \in [k_p] \mid \boldsymbol{O} \Rightarrow_{p,j} \boldsymbol{C}\} \neq \emptyset$ and otherwise $bts = \epsilon$, and $u = \sigma_\tau[i].word/bts$ if bts is a suffix of $\sigma_\tau[i].word$ and otherwise $u = \epsilon$.

(4) $w' = (bts :: w)\backslash v$.

We write $(q, w, \sigma_c, \sigma_p) \rightarrow^* (q', w', \sigma'_c, \sigma'_p)$ for $(q, w, \sigma_c, \sigma_p) \rightarrow \cdots \rightarrow (q', w', \sigma'_c, \sigma'_p)$. We give an intuitive reading of the definition. Roughly, v is the string to be consumed by the step, and (1) says that if $\ell \in \Sigma_\epsilon$ then ℓ is consumed, and otherwise ℓ is a capturing-group index (i.e., $\ell \in [k_c]$) and the string stored at the corresponding memory, i.e., $\sigma_c[\ell].word$, is consumed provided that the memory is closed. (2) and (3) stipulate how the statuses and strings of the memories are updated, respectively. Importantly, (3) defines the *backtrack string bts* to be used for backtracking caused by a closure of a positive lookahead memory (if any happens in the step). Namely, bts is set to be the longest string stored in positive lookahead memories closed by the step ($bts = \epsilon$ if no such closures happen), and is used in (3) to reset the content of the memories that are open and remain so (i.e., those satisfying $\boldsymbol{O} \Rightarrow_{\tau,i} \boldsymbol{O}$) by right-division (cf. the definition of u). The string contents remain unchanged for memories that are or remain closed by the step (i.e., those satisfying $\_ \Rightarrow_{\tau,i} \boldsymbol{C}$), and for the rest of the memories, the consumed string v is appended to their strings. (4) defines w', which is the input string in the post configuration (i.e., the string to be consumed in the continuation of the step), by prepending bts to the previous configuration's input string w to account for any backtracking that happens in the step, and consuming v by left-division. We remark that, for any configuration reachable from an initial configuration (see below), $\sigma_p[i].word$ is a prefix of $\sigma_p[j].word$ or vice versa for any $i, j \in [k_c]$, thus ensuring that bts is uniquely determined.

An *initial configuration* is $(q_0, w, \sigma_{c,0}, \sigma_{p,0})$, where $\sigma_{\tau,0}[i] = (\epsilon, \boldsymbol{C})$ for all $i \in [k_\tau]$ and $\tau \in \{c, p\}$. That is, every memory is initially closed and stores the empty string. A *run* of a PLMFA M is a sequence π such that $\pi[1]$ is an initial configuration and $\pi[i] \xrightarrow[M]{\ell} \pi[i+1]$ for all $1 \le i < |\pi|$. A run π is *accepting* if $\pi[|\pi|] = (q, \epsilon, \_, \_)$ for some $q \in F$. A string w is *accepted* by M if M has an accepting run. The language of M, denoted by $L(M)$, is the set of strings accepted by M. That is, $L(M) = \{w \in \Sigma^* \mid (q_0, w, \sigma_{c,0}, \sigma_{p,0}) \rightarrow^* (q, \epsilon, \sigma_c, \sigma_p) \wedge q \in F\}$. We note that when $k_p = 0$, a PLMFA(k_c, k_p) is a k_c*-memory automaton* (MFA(k_c) or simply MFA if k_c is irrelevant) introduced by Schmid [14].

▶ **Example 16.** As an example, consider a run of the PLMFA M shown in Fig. 3, which is equivalent to the rewbl$_p$ `(?=(`$_1$`.*)`$_1$`z)·\1` described in Example 4. In the figure, the (resp. double) circles represent (resp. accepting) states. The arrows represent transitions and the words on the labels represent labels on the transitions except for (?=, ($_1$,)$_1$, and). The arrow with the word (?= (resp.)) represents the transition $\delta(q_1, \epsilon) \ni (q_2, \diamond, \mathtt{o})$ (resp. $\delta(q_5, \epsilon) \ni (q_6, \diamond, \mathtt{c})$). Additionally, the arrow with the word ($_1$ (resp.)$_1$) represents the transition $\delta(q_2, \epsilon) \ni (q_3, \mathtt{o}, \diamond)$ (resp. $\delta(q_3, \epsilon) \ni (q_4, \mathtt{c}, \diamond)$). The rewbl$_p$ expression contains just one backreference and positive lookahead. For simplicity, we abbreviate the lists of memories $[(x, \boldsymbol{s})]$ as $(x, \boldsymbol{s})$.

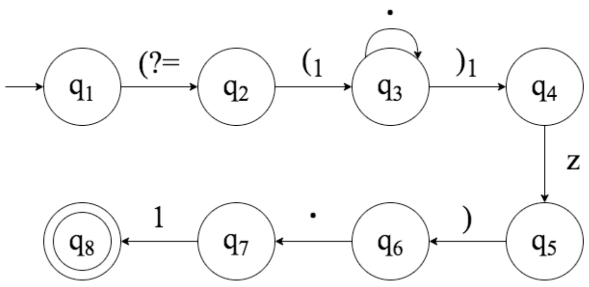

Figure 3 A PLMFA equivalent to the rewbl$_p$ expression `(?=(`$_1$`·*)`$_1$`z)·\1`.

Given an input string w=zz, the run of M is as follows: The run begins with the initial configuration $(q_0, \mathtt{zz}, (\epsilon, \boldsymbol{C}), (\epsilon, \boldsymbol{C}))$. First, the initial configuration changes to $(q_3, \mathtt{zz}, (\epsilon, \boldsymbol{O}), (\epsilon, \boldsymbol{O}))$ by applying the transitions $\delta(q_1, \epsilon) \ni (q_2, \diamond, \mathtt{o})$ and $\delta(q_2, \epsilon) \ni (q_3, \mathtt{o}, \diamond)$ in this order. The transition $\delta(q_1, \epsilon) \ni (q_2, \diamond, \mathtt{o})$ opens the positive-lookahead memory and the configuration changes to $(q_2, \mathtt{zz}, (\epsilon, \boldsymbol{C}), (\epsilon, \boldsymbol{O}))$. The transition $\delta(q_2, \epsilon) \ni (q_3, \mathtt{o}, \diamond)$ opens the capturing-group memory and the configuration changes to $(q_3, \mathtt{zz}, (\epsilon, \boldsymbol{O}), (\epsilon, \boldsymbol{O}))$. Next, the transitions $\delta(q_3, \cdot) \ni (q_3, \diamond, \diamond)$, $\delta(q_3, \epsilon) \ni (q_4, \mathtt{c}, \diamond)$, and $\delta(q_4, \mathtt{z}) \ni (q_5, \diamond, \diamond)$ are applied in this order. For the first transition, the configuration changes to the configuration $(q_3, \mathtt{z}, (\mathtt{z}, \boldsymbol{O}), (\mathtt{z}, \boldsymbol{O}))$ by consuming a character z. For the second transition, the configuration changes to the configuration $(q_4, \mathtt{z}, (\mathtt{z}, \boldsymbol{C}), (\mathtt{z}, \boldsymbol{O}))$ by closing the capturing-group memory. For the third transition, the configuration changes to the configuration $(q_5, \epsilon, (\mathtt{z}, \boldsymbol{C}), (\mathtt{zz}, \boldsymbol{O}))$ by consuming a character z.

Then, the configuration changes to $(q_6, \mathtt{zz}, (\mathtt{z}, \boldsymbol{C}), (\mathtt{zz}, \boldsymbol{C}))$ by applying the transition $(q_5, \epsilon) \ni (q_6, \diamond, \mathtt{c})$. The transition closes the positive-lookahead memory and therefore it simulates the backtracking behavior of the positive lookahead, i.e., it prepends the word of the positive-lookahead memory zz to the input string. Finally, the configuration changes to $(q_8, \epsilon, (\mathtt{z}, \boldsymbol{C}), (\mathtt{zz}, \boldsymbol{C}))$ by applying the transitions $\delta(q_6, \cdot) \ni (q_7, \diamond, \diamond)$ and $\delta(q_7, 1) \ni (q_8, \diamond, \diamond)$ in this order. The transition $\delta(q_6, \cdot) \ni (q_7, \diamond, \diamond)$ consumes a character z and the configuration changes to $(q_7, \mathtt{z}, (\mathtt{z}, \boldsymbol{C}), (\mathtt{zz}, \boldsymbol{C}))$. Next, the current state q_7 has the transition of the backreference \1, i.e., $\delta(q_7, 1) \ni (q_8, \diamond, \diamond)$. The transition tries to match the captured string, i.e., z, with the current input string. Since the match succeeds, the configuration changes to $(q_8, \epsilon, (\mathtt{z}, \boldsymbol{C}), (\mathtt{zz}, \boldsymbol{C}))$. Now, the current state q_8 is an accepting state and the current input string is ϵ, w is accepted by M.

▶ Remark 17. As seen above, capturing-group memories and positive-lookahead memories exhibit an interesting *symmetry*: at their use, the content of a capturing-group (resp. positive-lookahead) memory is left-divided from (resp. prepended to) the input string, and appended to (resp. right-divided from) the strings stored in memories. The symmetry is imperfect because positive-lookahead memories do not have "triggers" corresponding to backreferences of capturing-group memories and a use of a positive-lookahead memory is always synchronous with its closure. A perfect symmetry can be obtained by extending PLMFA with a new kind of transitions that trigger positive-lookahead memory uses, disassociating them from closures. The extension certainly does not decrease the expressive power of PLMFA and we conjecture that it will strictly increase the expressive power.

We define conditions on PLMFA that correspond to RMRL and NUR of rewbl (cf. Sec. 2). Note that we do not define conditions on PLMFA that correspond to NSR of rewbl because PLMFAs already satisfy such a condition, i.e., PLMFAs do not allow to refer to the memory whose status is open. For convenience, we simply call these conditions RMRL and NUR. Informally, a PLMFA satisfies RMRL if for all capturing-group memory (index) i and a state q from which the ith memory can be backreferenced, there exist a unique pair of transitions a and b that opened and closed the ith memory respectively so that the content of the ith memory when the computation reaches q is what was recorded between a and b. Intuitively, the pair of transitions correspond to the capturing group opening $(_i$ and closing $)_i$ of rewbl. Next, we formalize RMRL for PLMFA. For a configuration ϖ and $i \in [k_c]$, let us write $\mathit{status}(\varpi, i)$ for the status of the ith capturing-group memory of ϖ, i.e., $\sigma_c[i].\mathit{status}$ where $\varpi = (\_, \_, \sigma_c, \_)$.

▶ **Definition 18** (Opening and Closing Transitions Pair). For $q \in Q$ and $i \in [k_c]$, a pair of transitions $(\delta(p', \ell') \ni (q', ir'_c, ir'_p), \delta(p'', \ell'') \ni (q'', ir''_c, ir''_p)) = (a, b)$ is called an *opening-and-closing-transitions pair* of index i at state q if there exist a run π and $1 \leq j_1 < j_2 < |\pi|$ such that (1) the step from $\pi[j_1]$ to $\pi[j_1+1]$ takes the transition a and $\mathit{status}(\pi[j_1], i) = \boldsymbol{C}$, (2) the step from $\pi[j_2]$ to $\pi[j_2+1]$ takes the transition b and $\mathit{status}(\pi[j_2+1], i) = \boldsymbol{C}$, (3) for all $j_1 < l \leq j_2$, $\mathit{status}(\pi[l], i) = \boldsymbol{O}$, and for all $j_2 < l \leq |\pi|$, $\mathit{status}(\pi[l], i) = \boldsymbol{C}$, and (4) the state of $\pi[|\pi|]$ is q. We define $\mathit{RefSet}_{M,i}(q)$ (or $\mathit{RefSet}_i(q)$ if there is no danger of ambiguity) as the set of opening-and-closing-transitions pairs of i at q on M.

▶ **Definition 19** (RMRL-PLMFA). A PLMFA(k_c, k_p) $M = (Q, \delta, q_0, F)$ is called *restricted may repeat labels* (RMRL) if for all $(q, i) \in Q \times [k_c]$ such that $\delta(q, i) \neq \emptyset$, $|\mathit{RefSet}_i(q)| \leq 1$.

Next, we define NUR for PLMFA. Informally, a PLMFA satisfies NUR if no capturing-group memory can be backreferenced without capturing a word. Formally, for $(q, i) \in Q \times [k_c]$, we say that q is *assigned* with respect to index i on M, written $\mathit{Assigned}_M(q, i)$ (or $\mathit{Assigned}(q, i)$ if there is no danger of ambiguity), if for all runs π such that the state of $\pi[|\pi|]$ is q, there exists $1 \leq j < |\pi|$ such that $\mathit{status}(\pi[j], i) = \boldsymbol{O}$ and $\mathit{status}(\pi[j+1], i) = \boldsymbol{C}$.

▶ **Definition 20** (NUR-PLMFA). A PLMFA(k_c, k_p) $M = (Q, \delta, q_0, F)$ is *no unassigned reference (NUR)* if for all $(q, i) \in Q \times [k_c]$ such that $\delta(q, i) \neq \emptyset$, $\mathit{Assigned}(q, i) = \mathit{true}$.

In what follows, we assume that PLMFAs satisfy RMRL and NUR.

4.2 Normal Forms and Nested Forms

We show that a PLMFA can be converted into certain forms. The *normal form* enforces two restrictions: (1) only ϵ transitions can change memory statuses and at most one status of the memory at a time, and (2) no transitions open (resp. close) a memory that is already opened (resp. closed).

▶ **Definition 21** (Normal Form). A PLMFA is in *normal form* if the following properties are satisfied. For every transition $\delta(q, \ell) \ni (q', ir_c, ir_p)$, $\tau \in \{c, p\}$, and $j \in [k_\tau]$, (1) if $ir_\tau[j] \neq \diamond$, then $\ell = \epsilon$ and $ir_{\tau'}[l].\mathit{status} = \diamond$ for all $(\tau', l) \in \{c, p\} \times [k_{\tau'}]$ such that $(\tau', l) \neq (\tau, j)$, and (2) there is no run π such that $\pi[|\pi|] = (\_, \_, \sigma_c, \sigma_p)$ where $ir_\tau[j] = \mathsf{o}$ and $\sigma_\tau[j].\mathit{status} = \boldsymbol{O}$ or $ir_\tau[j] = \mathsf{c}$ and $\sigma_\tau[j].\mathit{status} = \boldsymbol{C}$.

▶ **Lemma 22.** *Any PLMFA M can be converted to a normal form PLMFA M' such that $L(M) = L(M')$.*

The proof is by adopting an analogous conversion of [14] and works by extending the states of M to record memory statuses and splitting simultaneous memory updates to multiple transitions. We remark that the conversion preserves RMRL and NUR.

Next, we define *nested form* which enforces that there are no *overlaps* in any runs. A run π is said to have an overlap if there exist $1 \leq j_1 < j_2 < j_3 < j_4 < |\pi|$ such that some memory is opened and closed respectively at the j_1th and the j_3th steps and some memory (possibly the same) is opened and closed respectively at the j_2th step and the j_4th step. There are four types of overlaps, *cc*, *cp*, *pc*, and *pp*, depending on the types of the first and the second memories (e.g., *cp*-overlap is when the first memory is capturing-group and the second is positive-lookahead). Intuitively, an overlap corresponds to an invalid expression that has an overlap of capturing groups or positive lookaheads. For example, *cc*-overlap corresponds to invalid expressions $(_i r_1 (_j r_2)_i r_3)_j$ and *pc*-overlap corresponds to invalid expressions $(?{=}r_1 (_i r_2) r_3)_i$.

▶ **Definition 23** (Nested Form). A PLMFA M is in *nested form* if there are no overlaps in any runs on M.

We show that we can transform a PLMFA to the nested form.

▶ **Lemma 24.** *Any PLMFA M can be converted to a normal and nested form PLMFA M' such that $L(M) = L(M')$.*

The proof is by adding new transitions that close the fragments of the memories which are open before opening the transitions that cause overlaps and open the next fragments of the memories after that. For cc-overlaps, the conversion coincides with the analogous one for MFAs [14]. In what follows, without loss of generality, we assume that PLMFAs are in normal and nested form.

4.3 From Rewbl$_p$ to PLMFA

We show that given a rewbl$_p$ expression r, we can construct a PLMFA M such that $L(r) = L(M)$. The construction of the PLMFA extends the standard Thompson construction from a pure regular expression to an NFA [15] with backreferences and lookaheads in a mostly straightforward manner. For space, the construction and the proof of correctness is omitted.

▶ **Theorem 25.** *Let a rewbl$_p$ r include k_c capturing groups and k_p positive lookaheads. Then, there exists a PLMFA(k_c, k_p) M such that $L(r) = L(M)$.*

4.4 From PLMFA to Rewbl$_p$

Now, we show that given a PLMFA M, we can construct a rewbl$_p$ expression r such that $L(M) = L(r)$. We first give the conversion and then show the correctness. For space, we only show the correctness of the language equivalence and omit the fact that the conditions NUR, NSR, and RMRL are satisfied by the resulting rewbl$_p$. However, they can be proved similarly to the language equivalence.

The conversion, referred to as *PtoR*, is inspired by that of MFAs to rewbs [14]. The idea of the conversion is to use the nested relation of PLMFAs. Since PLMFAs are nested form, the transitions of the opening-and-closing-transitions pairs have a nested structure, i.e., they form a directed acyclic graph (DAG). Therefore, we can iteratively convert (sub)automaton corresponding to the part of the given PLMFA delimited by each such pairs to the rewbl$_p$ expression in a topological order starting from the pairs that nest nothing. Each step of the conversion makes an *extended PLMFA* (ePLMFA) whose labels are rewbl$_p$ expressions. That is, labels ℓ on transitions of PLMFAs are treated as rewbl$_p$ expressions ℓ of ePLMFAs if

$\ell \in \Sigma_\epsilon$. Additionally, if $\ell = i \in [k_c]$ on the transitions of PLMFAs, $\ell = \backslash i$ on the transitions of ePLMFAs. For an ePLMFA M and a rewbl$_p$ expression ℓ, a step of computation of M reading ℓ is defined as follows: $(q, w, \sigma_c, \sigma_p) \xrightarrow{\ell} (q', w', \sigma_c', \sigma_p')$ iff either $\ell = \epsilon$ and $(q, w, \sigma_c, \sigma_p) \xrightarrow{\epsilon} (q', w', \sigma_c', \sigma_p')$ according to Def. 15, or $\ell \neq \epsilon$ and there exist a transition $\delta(q, \ell) \ni (q', \diamond^{k_c+k_p})$ and steps of computations from $(q_0', w, \sigma_c, \sigma_p)$ to $(q_f', w', \sigma_c', \sigma_p')$ of $M' = (\_, \_, q_0', \{q_f'\})$ obtained from ℓ by the construction from a rewbl$_p$ expression to a PLMFA mentioned in Sec. 4.3.

We also extend the NUR and RMRL conditions for ePLMFAs as follows. For all transition, we reconstruct PLMFA from the rewbl$_p$ label by applying the construction mentioned in Sec. 4.3 and replace the transition with the PLMFA by replacing it with ϵ-labeled $\diamond$-only transitions to and from the initial and the final state of the PLMFA. We say that an ePLMFA satisfies NUR and RMRL if the reconstructed PLMFA satisfies NUR and RMRL, respectively.

We define the nesting relation. Firstly, for an ePLMFA and $\texttt{inst} \in \{\texttt{o}, \texttt{c}\}$, we define $Q_{\tau,i,\texttt{inst}}$ as the set of states q such that $\delta(q, \_) \ni (\_, ir_c, ir_p)$ where $ir_\tau[i] = \texttt{inst}$. Let $\Phi = \{(\tau, i, q, q') \mid \tau \in \{c, p\} \wedge i \in [k_\tau] \wedge q \in Q_{\tau,i,\texttt{o}} \wedge q' \in Q_{\tau,i,\texttt{c}}\}$. The *nesting relation* $\prec \subseteq \Phi \times \Phi$ is defined as follows: $(\tau_1, i_1, q_1, q_1') \prec (\tau_2, i_2, q_2, q_2')$ iff there exist a run π and $s < t < u < v < \pi[|\pi|]$ such that the step from $\pi[s]$ to $\pi[s+1]$ takes a transition $\delta(q_2, \_) \ni (\_, ir_c, ir_p)$ where $ir_{\tau_2}[i_2] = \texttt{o}$, the step from $\pi[t]$ to $\pi[t+1]$ takes a transition $\delta(q_1, \_) \ni (\_, ir_c, ir_p)$ where $ir_{\tau_1}[i_1] = \texttt{o}$, the step from $\pi[u]$ to $\pi[u+1]$ takes a transition $\delta(q_1', \_) \ni (\_, ir_c, ir_p)$ where $ir_{\tau_1}[i_1] = \texttt{c}$, and the step from $\pi[v]$ to $\pi[v+1]$ takes a transition $\delta(q_2', \_) \ni (\_, ir_c, ir_p)$ where $ir_{\tau_2}[i_2] = \texttt{c}$.

For an ePLMFA M, the procedure of *PtoR*(M) is defined as follows. Let us initialize $\Delta : \Phi \to \mathcal{P}(\Phi)$ as follows: $\Delta(\tau_1, i_1, q_1, q_1') = \{(\tau_2, i_2, q_2, q_2') \mid (\tau_2, i_2, q_2, q_2') \prec (\tau_1, i_1, q_1, q_1')\}$. We iteratively update M, Φ, and Δ by the following steps.

1. Find $(\tau, i, q, q') \in \Phi$ such that $\Delta(\tau, i, q, q') = \emptyset$. Let $\delta(q, \_) \ni (q_s, \_)$ (resp. $\delta(q', \_) \ni (q_e, \_)$) be the opening (resp. closing) transition of (τ, i, q, q'). Then, construct an ePLMFA M' from M by replacing the initial state and set of accepting states with q_s and $\{q'\}$, respectively, and deleting all transitions that open or close memories.
2. Convert M' to a rewbl$_p$ expression r using the standard state elimination method.
3. Delete (τ, i, q, q') from Φ and (the domain and the range of) Δ, and add the transition $\delta(q, ({}_i r)_i) \ni (q_e, \diamond^{k_c+k_p})$ if $\tau = c$ and otherwise $\delta(q, (?{=}r)) \ni (q_e, \diamond^{k_c+k_p})$ to M.
4. Repeat steps 1 to 3 until convergence.
5. Delete all transitions that open or close memories from M.
6. Convert M to a rewbl$_p$ expression r by the state elimination method and return r.

At step 1, we can find such a tuple (τ, i, q, q') since it is in nested form. The state elimination method used in steps 2 and 6 is a straightforward adoption of the standard state elimination method (see, e.g., [13]) that interprets the rewbl$_p$ expression that appear as labels as ordinary regular expressions.

We proceed to the proof of correctness.

▶ **Definition 26.** The *language* of an ePLMFA $M = (Q, \delta, q_0, F)$ parameterized by a string $w \in \Sigma^*$, a capturing-group memory σ_c, and a positive-lookahed memory σ_p, denoted by $L(M, w, \sigma_c, \sigma_p)$, is defined as

$$L(M, w, \sigma_c, \sigma_p) \quad = \quad \{(w', \sigma_c', \sigma_p') \mid \exists q' \in F.\ (q_0, w, \sigma_c, \sigma_p) \to^* (q', w', \sigma_c', \sigma_p')\}.$$

For an ePLMFA $M = (Q, \delta, q_0, F)$, we write $M(q', F')$ for the ePLMFA $M' = (Q, \delta, q', F')$ where $q' \in Q$ and $F' \subseteq Q$. We show that eliminating a state from an ePLMFA by applying one step of the state elimination method does not change the parameterized language of the ePLMFA. The state elimination method eliminates a state by deleting and adding transitions as shown in Fig. 4.

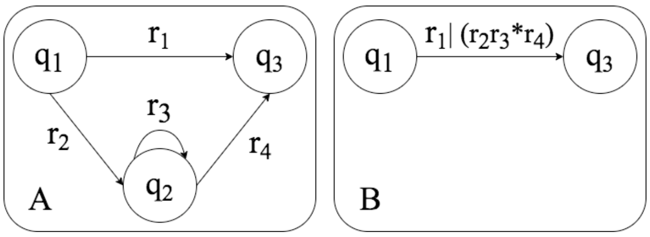

Figure 4 (A) Before eliminating the state q_2. (B) After eliminating the state q_2.

▶ **Lemma 27.** *Let $M = (Q, \delta, q_0, F)$ be an ePLMFA(k_c, k_p) such that for all transitions $\delta(q, \ell) \ni (q', ir_c, ir_p)$, $i \in [k_c]$, and $j \in [k_p]$, $ir_c[i] = \diamond$ and $ir_p[j] = \diamond$. Additionally, let $M' = (Q', \delta', q_0', F')$ be the ePLMFA obtained by eliminating a state in Q from M by the state elimination method. Then, for all $w \in \Sigma^*$, σ_c, and σ_p, $L(M, w, \sigma_c, \sigma_p) = L(M', w, \sigma_c, \sigma_p)$.*

Proof. Let us assume that q_1 and q_3 are in Q and Q', $q_2 \in Q$ is the state eliminated by the state elimination method as shown in Fig. 4. Since the only difference between M and M' comes from the state q_2, it suffices to show that $L(M(q_1, \{q_3\}), w, \sigma_c, \sigma_p) = L(M'(q_1, \{q_3\}), w, \sigma_c, \sigma_p)$. It is immediate from the construction. ◀

▶ **Theorem 28.** *For an ePLMFA M, let M_j be the ePLMFA obtained after the jth iteration of steps 1 to 3 of PtoR(M). We assume $M_j = M$ if $j = 0$. Then, for all $w \in \Sigma^*$, σ_c, and σ_p, $L(M, w, \sigma_c, \sigma_p) = L(M_j, w, \sigma_c, \sigma_p)$.*

Proof. The proof is by induction on the number of the iteration j of step 1 to 3. The base step $j = 0$ is immediate since $M_j = M$. For the induction step, let $j > 0$. The inductive hypothesis is that $L(M, w, \sigma_c, \sigma_p) = L(M_j, w, \sigma_c, \sigma_p)$ holds for all j, w, σ_c, and σ_p. We then consider M_{j+1}. We show $L(M_j, w, \sigma_c, \sigma_p) = L(M_{j+1}, w, \sigma_c, \sigma_p)$. As described in the step 3, for the ePLMFA $M_j = (Q, \delta, q_0, F)$, $M_{j+1} = (Q, \delta \cup \{t\}, q_0, F)$ where $t = \delta(q, r') \ni (q_e, \diamond^{k_c+k_p})$. Recall that $r' = (_i r)_i$ or $(?{=}r)$ and r is the rewbl$_p$ expression obtained at step 2. For this, $L(M_j, w, \sigma_c, \sigma_p) \subseteq L(M_{j+1}, w, \sigma_c, \sigma_p)$ is immediate. For $L(M_{j+1}, w, \sigma_c, \sigma_p) \subseteq L(M_j, w, \sigma_c, \sigma_p)$, it suffices to show that for all w, σ_c, and σ_p, if there exists $(q, w, \sigma_c, \sigma_p) \xrightarrow{r'} (q_e, w', \sigma_c', \sigma_p')$ on M_{j+1}, then there exists $(q, w, \sigma_c, \sigma_p) \rightarrow^* (q_e, w', \sigma_c', \sigma_p')$ on M_j. We assume that there exists $\pi = (q, w, \sigma_c, \sigma_p) \xrightarrow{r'} (q_e, w', \sigma_c', \sigma_p')$ on M_{j+1}. The computation of the transition whose label is r' is defined by that of the ePLMFA $M_{r'}$ obtained from r' by the construction mentioned in Sec. 4.3. If $r' = (_i r)_i$, $M_{r'} = (Q_{r'}, \delta_{r'}, q, \{q_e\})$ where

- $Q_{r'} = \{q, q_e\} \cup Q_r$; and
- $\delta_{r'} = \delta_r \cup \{((q, \epsilon), \{(q_{0,r}, ir_c, \diamond^{k_p})\})\} \cup \{((q_{F,r}, \epsilon), \{(q_e, ir_c', \diamond^{k_p})\})\}$ where $ir_c[k] = \mathsf{o}$ and $ir_c'[k] = \mathsf{c}$ if $k = i$ and otherwise $ir_c[k] = \diamond$ and $ir_c'[k] = \diamond$ for $k \in [k_c]$

with the ePLMFA for r $M_r = (Q_r, \delta_r, q_{0,r}, \{q_{F,r}\})$. By the transitions in $\delta_{r'}$, $\pi = (q, w, \sigma_c, \sigma_p) \rightarrow (q_{0,r}, w, \sigma_c'', \sigma_p) \xrightarrow{r} (q_{F,r}, w', \sigma_c''', \sigma_p') \rightarrow (q_e, w', \sigma_c', \sigma_p')$ where $\sigma_c''[k] = (\epsilon, \boldsymbol{O})$ and $\sigma_c'''[k] = (\sigma_c'[k].word, \boldsymbol{O})$ if $k = i$ and otherwise $\sigma_c''[k] = \sigma_c[k]$ and $\sigma_c'''[k] = \sigma_c'[k]$. We focus on the computation $(q_{0,r}, w, \sigma_c'', \sigma_p) \xrightarrow{r} (q_{F,r}, w', \sigma_c''', \sigma_p')$. The label r is constructed from M' by the state elimination method at step 2. Additionally, by Lemma 27, the state elimination method preserves the parameterized language equivalence. For this, there exists $(q_s, w, \sigma_c'', \sigma_p) \rightarrow^* (q', w', \sigma_c''', \sigma_p')$ on M'. Since the transition function of M_j includes all transitions of M', there also exists $(q_s, w, \sigma_c'', \sigma_p) \rightarrow^* (q', w', \sigma_c''', \sigma_p')$ on M_j. As described at step 1, M_j has a transition from q to q_s that opens the ith capturing-group memory and a transition from q' to q_e that closes the ith capturing-group memory. Therefore, there exists $(q, w, \sigma_c, \sigma_p) \to (q_s, w, \sigma_c'', \sigma_p) \to^* (q', w', \sigma_c''', \sigma_p') \to (q_e, w', \sigma_c', \sigma_p')$. For this, $L(M_j, w, \sigma_c, \sigma_p) = L(M_{j+1}, w, \sigma_c, \sigma_p)$. By the inductive hypothesis, $L(M_j, w, \sigma_c, \sigma_p) = L(M, w, \sigma_c, \sigma_p)$. Thus, $L(M, w, \sigma_c, \sigma_p) = L(M_{j+1}, w, \sigma_c, \sigma_p)$ for the case $r' = ({}_i r)_i$. The case of $r' = (?{=}r)$ is analogous. ◀

Now, we obtain our main result.

▶ **Theorem 29.** *Let M be an ePLMFA. Then, $L(M) = L(r)$ where $r = PtoR(M)$.*

Proof. Let M' be the ePLMFA at the last step of the state elimination method at step 6 of *PtoR*. Then, $M' = (\{q, q'\}, \{((q, r), \{(q', \diamond^{k_c+k_p})\})\}, q, \{q'\})$. By Theorem 25, $L(M') = L(r)$. By Theorem 28, for all $w \in \Sigma^*$, $L(M', w, \sigma_{c,0}, \sigma_{p,0}) = L(M, w, \sigma_{c,0}, \sigma_{p,0})$, i.e., $L(M') = L(M)$. Thus, $L(M) = L(M') = L(r)$. ◀

As a corollary of Theorem 25 and Theorem 29, we obtain the following result.

▶ **Corollary 30.** *The expressive power of PLMFA is equivalent to that of $rewbl_p$.*

5 Related Work

Among the major extensions employed by modern real-world regular expression engines are backreferences and lookaheads. However, the previous works on formal language theory have studied the two features mostly in isolation, and to the best of our knowledge, our work is the first formal study of regular expressions extended with both features. Next, we discuss previous works that studied the features in isolation.

Prior works by Morihata and Berglund et al. [12, 3] showed that extending regular expressions by lookaheads does not enhance their expressive power. Their proofs are by a translation to boolean finite automata [4] whose expressive power is regular. However, adopting such an approach to defining an equivalent automata is difficult in the presence of backreferences because boolean automata express lookaheads by running several states simultaneously without backtracking, while the combination of lookaheads and backreferences intrinsically requires backtracking. For example, to match against `(?=`$\cdot$`(`$\cdot$`)`$_1$`)\1`$\cdot$, a boolean approach would run `(?=`$\cdot$`(`$\cdot$`)`$_1$`)` and `\1`$\cdot$ simultaneously, but then the automaton would get stuck while trying to process the backreference `\1` as it is unassigned at that point. By constrast, our PLMFA uses the novel positive-lookahead memories to store enough information to simulate the backtracking behavior of positive lookaheads.

A formal study of regular expressions with backreferences (rewb) dates back to the seminal work by Aho [1]. More recently, a formal semantics and a pumping lemma were given by Câmpeanu et al. [5]. Berglund and van der Merwe [2] showed that different variants of backreference semantics give rise to differences in expressive powers. Our works adopts and formalizes the no-label-repetitions (NLR) and no-unassigned-reference (NUR) semantics which is also used in [5, 6]. Schmid [14] proposed MFA, and showed that the expressive power of the automata is equivalent to that of rewb. Our PLMFA builds on MFA and extends it with positive-lookahead memories to handle positive lookaheads. As remarked before (cf. Remark 17), our positive-lookahead memory exhibits an interesting symmetry to the capturing-group memory of MFA.

6 Conclusion

We have studied the expressive powers of regular expressions with the popular backreferences and lookaheads extensions. We have shown that extending rewb by positive or negative lookaheads enhance their expressive power. Additionally, we have presented language-theoretic properties of rewb extended by the two forms of lookaheads, and have presented a new class of automata called PLMFA that is equivalent in expressive power to rewbl$_p$. We have introduced a new kind of memories called a positive-lookahead memory, which is almost perfectly symmetric to capturing-group memory of MFA, as a key component of PLMFA.

Despite the popularity of the backreference and lookaheads extensions in practice, to our knowledge, our work is the first formal study on regular expressions with both extensions. We hope that our results pave the way for more work on the topic.

References

1 Alfred V. Aho. Chapter 5 - algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 255–300. Elsevier, Amsterdam, 1990. doi:10.1016/B978-0-444-88071-0.50010-2.

2 Martin Berglund and Brink van der Merwe. Regular expressions with backreferences re-examined. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, pages 30–41. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017. URL: http://www.stringology.org/event/2017/p04.html.

3 Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. Regular expressions with lookahead. *J. Univers. Comput. Sci.*, 27(1):324–340, 2021. doi:10.3897/jucs.66330.

4 Janusz A. Brzozowski and Ernst L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10:19–35, 1980. URL: http://dblp.uni-trier.de/db/journals/tcs/tcs10.html#BrzozowskiL80.

5 Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018, 2003. doi:10.1142/S012905410300214X.

6 Benjamin Carle and Paliath Narendran. On extended regular expressions. In Adrian-Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, volume 5457 of *Lecture Notes in Computer Science*, pages 279–289. Springer, 2009. doi:10.1007/978-3-642-00982-2_24.

7 Ashok K. Chandra and Larry J. Stockmeyer. Alternation. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 98–108, 1976. doi:10.1109/SFCS.1976.4.

8 N. Chida and T. Terauchi. Repairing dos vulnerability of real-world regexes. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy, SP 2022*, pages 1049–1066, Los Alamitos, CA, USA, May 2022. IEEE Computer Society. doi:10.1109/SP46214.2022.00061.

9 S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*. 1956.

10 Vladimir Komendantsky. Matching problem for regular expressions with variables. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 2012. doi:10.1007/978-3-642-40447-4_10.

11 Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 425–438, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314645.

12 Akimasa Morihata. Translation of regular expression with lookahead into finite state automaton. *Computer Software*, 29(1):1_147–1_158, 2012. `doi:10.11309/jssst.29.1_147`.

13 Jacques Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, USA, 2009.

14 Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.*, 249:1–17, 2016. `doi:10.1016/j.ic.2016.02.003`.

15 Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. `doi:10.1145/363347.363387`.