Efficient Matching of Regular Expressions with Lookaround Assertions

KONSTANTINOS MAMOURAS, Rice University, USA
AGNISHOM CHATTOPADHYAY, Rice University, USA

Regular expressions have been extended with lookaround assertions, which are subdivided into lookahead and lookbehind assertions. These constructs are used to refine when a match for a pattern occurs in the input text based on the surrounding context. Current implementation techniques for lookaround involve backtracking search, which can give rise to running time that is super-linear in the length of input text. In this paper, we first consider a formal mathematical semantics for lookaround, which complements the commonly used operational understanding of lookaround in terms of a backtracking implementation. Our formal semantics allows us to establish several equational properties for simplifying lookaround assertions. Additionally, we propose a new algorithm for matching regular expressions with lookaround that has time complexity $O(m \cdot n)$, where m is the size of the regular expression and n is the length of the input text. The algorithm works by evaluating lookaround assertions in a bottom-up manner. Our algorithm makes use of a new notion of nondeterministic finite automata (NFAs), which we call oracle-NFAs. These automata are augmented with epsilon-transitions that are guarded by oracle queries that provide the truth values of lookaround assertions at every position in the text. We provide an implementation of our algorithm that incorporates three performance optimizations for reducing the work performed and memory used. We present an experimental comparison against PCRE and Java's regex library, which are state-of-the-art regex engines that support lookaround assertions. Our experimental results show that, in contrast to PCRE and Java, our implementation does not suffer from super-linear running time and is several times faster.

CCS Concepts: • **Theory of computation** → *Regular languages*; • **Software and its engineering** → Semantics.

Additional Key Words and Phrases: regex, automata, lookahead, lookbehind, lookaround, Kleene algebra, regex matching, regex engine

ACM Reference Format:
Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proc. ACM Program. Lang.* 8, POPL, Article 92 (January 2024), 31 pages. https://doi.org/10.1145/3632934

1 INTRODUCTION

Since their introduction in the 1950s, regular expressions [Kleene 1956] and finite-state automata [Rabin and Scott 1959] have found applications in numerous domains to describe patterns over sequences. They have been used for the lexical analysis of programs [Johnson et al. 1968] during compilation, the search of words and patterns in text editors [Thompson 1968], and bibliographic search [Aho and Corasick 1975]. Regular patterns are also used in network security [Yu et al. 2006] to search for intrusion signatures in network traffic, in bioinformatics [Roy and Aluru 2016] for

Authors' addresses: Konstantinos Mamouras, Rice University, Houston, Texas, USA, mamouras@rice.edu; Agnishom Chattopadhyay, Rice University, Houston, Texas, USA, agnishom@rice.edu.

ACM 2475-1421/2024/1-ART92
https://doi.org/10.1145/3632934

describing protein, RNA, or DNA sequences, and in runtime verification [Bartocci et al. 2018] for specifying safety properties.

Classical regular expressions involve constructs for nondeterministic choice $r_1 + r_2$, concatenation $r_1 \cdot r_2$, and Kleene star r^* (repetition of r zero or more times). In practice, the syntax of regular expressions is often extended with more constructs that offer convenience, such as character classes for describing sets of letters/symbols (e.g., $[ab]$ and $[0{-}9]$), the construct $r?$ for indicating that the pattern r is optional, and Kleene plus r^+ (repetition of r at least once). The construct of bounded repetition, which is written as $r\{m, n\}$ and describes the repetition of r from m to n times, can be translated using concatenation and ? but makes regular expressions exponentially more succinct. Prior work explores the algorithmic challenges of bounded repetition [Kong et al. 2022; Le Glaunec et al. 2023]. In this paper, we focus on another commonly-used construct called *lookaround*, which goes beyond classical regular expressions by allowing one to describe not only a pattern to search for, but also the context in which the pattern should appear. There are two kinds of lookaround: *lookahead*, which we write as $(?{>}\, r)$, and *lookbehind*, which we write as $(?{<}\, r)$. The lookahead $(?{>}\, r)$ asserts that the text that lies ahead (i.e., in the "future" relative to the current position) matches the pattern r. Similarly, the lookbehind $(?{<}\, r)$ asserts that the text that lies behind (i.e., in the "past") matches the pattern r. For example, the regular expression $(?{<}\, [\text{␣}])[A{-}Z][a{-}z]^+(?{>}\, [,;])$ can be used to search for the occurrence of a word starting with a capital letter (the character classes $[A{-}Z]$ and $[a{-}z]$ recognize capital and lowercase letters respectively) that is preceded by a space (indicated with $[\text{␣}]$) and followed by a comma or a semicolon (indicated with $[,;]$). The use of lookaround allows us to focus only on the word described by the subpattern $[A{-}Z][a{-}z]^+$, without including in the match the context that is specified by the lookbehind $(?{<}\, [\text{␣}])$ and lookahead $(?{>}\, [,;])$. Lookaround assertions can also be used to express some forms of intersection. For example, a string matches the pattern $(?{>}\, .^*[0{-}9].^*)(?{>}\, .^*[a{-}z].^*)(?{>}\, .^*[A{-}Z].^*)(?{>}\, .^*[\#\&@].^*)$ (where . is notation for the character class that accepts every symbol) iff it contains at least one decimal digit, one lowercase letter, one uppercase letter, and one of the symbols of $\{\#, \&, @\}$.

The main computational problem for regular expressions is the *pattern matching* problem: given a regular expression r and input text w, find the locations in the input text that match the regular expression. A slightly more restricted variant is the *membership* problem, which asks whether the entire input text w matches the pattern r. Several different approaches have been used for implementing regular pattern matching. These implementations are often called *regex engines*. One common class of implementations relies on backtracking search. This approach is used in the PCRE library [The PCRE2 Developers 2023] and the regex engines of many popular programming languages, such as Java, Python, and JavaScript. Backtracking search suffers from exponential running time (in the length of the input text) for certain regular expressions that cause the exploration of a large number of paths. This exponential explosion of explored paths is often referred to as *catastrophic backtracking* [Berglund et al. 2014]. There are also regexes for which the running time is polynomial but super-linear (again, in the length of the input text). The super-linear (in some cases exponential) running time of regex engines based on backtracking is much worse than what can be achieved using standard automata-theoretic approaches. For example, Thompson's classical algorithm [Thompson 1968] is essentially an implementation based on nondeterministic finite automata (NFAs). It has time complexity $O(m \cdot n)$, where m is the size of the regular expression and n is the length of the input text.

Many modern regex engines are based on deterministic finite automata (DFAs), or NFAs, or a combination of both. This includes the widely used engines grep [2023], Google's RE2 [2023], and Intel's Hyperscan [2023]. In contrast to backtracking-based regex engines, automata-based engines offer strong guarantees of performance: the running time is *linear* in the length of the input text. Algorithms based on DFAs are very fast. They perform $O(1)$ work per input symbol, because they only need to perform a memory lookup in the DFA transition table for each input symbol. The problem with DFA-based implementations is that the size of the DFA can be exponential in the size of the regex in the worst case. NFAs, on the other hand, can be exponentially more succint than DFAs. Moreover, every regular expression that uses only the classical regular combinators can be translated into an NFA whose state space is linear in the size of the expression. The problem with NFA-based algorithms, compared to DFA-based algorithms, is that they may need to perform $\Theta(m)$ computation steps for each input symbol.

The current state of affairs regarding the support of lookaround in regex engines is rather disappointing. Existing automata-based regex engines (grep, RE2, Hyperscan) do not support lookaround at all. While it is known that the membership problem for regular expressions with lookaround can be solved using finite-state automata (see, for example, [Morihata 2012] and [Miyazaki and Minamide 2019]), these automata are very large due to the succinctness of lookaround. A DFA of doubly exponential size is needed in the worst case (and therefore an NFA of exponential size). These observations suggest that simple solutions based on automata that encode the entire pattern will suffer from high complexity (with respect to the size of the pattern). Berglund et al. [2021] consider the construction of alternating finite automata (AFA) from regular expressions with lookahead assertions. In this construction, the number of states of the AFA is linear in the size of the regular expression. A consequence of this, which is not explicitly discussed in [Berglund et al. 2021], is that membership can be decided in $O(m \cdot n)$ time with a right-to-left pass over the input string that simulates the AFA execution "in reverse". This approach does not handle lookbehind assertions. Moreover, while the simulation of AFA execution can decide membership, it is not applicable to match extraction. This is because the states for lookaheads are not distinguished from the states for the "main" part of the regex, and therefore a disambiguation policy cannot be expressed. Several backtracking-based regex engines support general lookahead, but only very restricted forms of lookbehind. For example, the widely used PCRE library only supports bounded lookbehind, which can only refer to a bounded amount of "past" text. As we will see in Section 6, the use of lookaround in existing backtracking-based engines can easily trigger catastrophic backtracking. This means that there is currently ***no efficient implementation of lookaround*** in the context of regular pattern matching. This is the main problem that we tackle in this paper. Our approach is fully general in that we allow the arbitrary nesting of unrestricted lookahead and lookbehind assertions.

The ***key idea*** of our approach is that it is possible to decompose the overall computation for pattern matching with lookaround. First, we simplify the problem by making the assumption that all lookaround assertions can be resolved by *oracles* that know the truth values of both lookaheads and lookbehinds at each position in the text. Under this assumption, we show that it is possible to augment NFA-based algorithms with a special kind of ε-transition that is guarded by a *query* to an oracle. The second step of our approach is to replace the oracles by algorithms that compute all necessary truth values. This is done in a bottom-up manner. Consider a sub-pattern r' of the overall pattern r whose top-level operator is a lookaround and contains no other occurrence of lookaround. We compute whether r' matches the input text at each position. After this computation is performed, then the sub-pattern r' can be replaced by an *oracle query* that uses the pre-computed truth values. By following this bottom-up "compute and replace" process we ensure that the answers for each oracle query have already been computed and are therefore available to the algorithm.

We note that our algorithm is not streaming in the general case because it performs right-to-left passes over the input text to deal with lookahead assertions efficiently. If the regular expression contains no lookahead assertions, then the algorithm can deal with lookbehind in a streaming manner (single left-to-right pass over the input text).

Main contributions. We make the following contributions in this paper:

(1) We present a formal semantics for lookaround using a *satisfaction relation* that relates a string w, a location (interval) $[i, j]$ within w, and a regular pattern r. We show that this is equivalent to an algebraic semantics that generalizes the classical language interpretation of regular expressions (without lookaround). This mathematical semantics complements existing definitions of the lookaround constructs that are operational, i.e., defined in terms of a backtracking matching algorithm.
(2) Using our formal semantics for lookaround, we prove that regular expressions with lookaround satisfy the equivalence properties of Kleene algebra [Kozen 1994]. Moreover, we establish a number of equivalences involving lookaround that can be used for simplifying patterns.
(3) We introduce the notion of regular expressions and ε-NFAs with *oracle queries* as a way to abstract away lookaround assertions. We call them *oracle-regexes* and *oracle-NFAs* respectively. We provide an algorithm for matching such oracle-regexes, which can be understood as a simulation of oracle-NFA semantics. The time complexity of this algorithm is $O(m \cdot n)$, where m is the size of the oracle-regex and n is the length of the input text.
(4) We propose a recursive algorithm for matching regular expressions with lookaround. This algorithm is based on the bottom-up decomposition approach described earlier and makes essential use of the algorithm for oracle-regex matching. Its time complexity is $O(m \cdot n)$, where m is the size of the regular expression and n is the length of the input text.
(5) We introduce three performance optimizations to the aforementioned algorithm: (i) common assertion elimination, (ii) one-pass unidirectional evaluation to reduce the memory footprint to $O(m)$, and (iii) approximation to avoid the computation of some lookaround assertions.
(6) We provide an experimental evaluation of a Rust implementation of our algorithm against PCRE and Java's regex library, state-of-the-art regex engines that support lookaround. Our experiments show (i) that our performance optimizations provide a significant performance benefit, and (ii) that, in contrast to PCRE and Java, our implementation does not suffer from super-linear (in the length of the input text) time complexity. PCRE and Java have worse performance than our implementation over the workloads that we consider.

The current work presents the first tool for matching regular expressions with lookaround that provides strong worst-case complexity guarantees and has competitive performance against state-of-the-art regex engines.

2 SEMANTICS OF LOOKAROUND

In this section, we present a formal mathematical semantics for regular expressions with lookaround. There is little prior work on the formal semantics of lookaround. The most common approach is to define it operationally: the meaning of lookaround is described by the backtracking algorithm that performs regex matching. This approach is, however, unsatisfactory because it conflates the specification (which should be simple and easily understandable) with the implementation (which can be very complex) and therefore does not allow one to formally prove the correctness of the matching algorithm. A notable exception to this is the semantic treatment of lookahead in [Miyazaki and Minamide 2019], where the authors use languages of string pairs to interpret regular expressions. These semantic objects are, however, insufficient for giving a semantics in the presence of both lookahead and lookbehind.

We provide two equivalent semantic perspectives. The first one is logical and employs a ternary *satisfaction relation*, which relates a string w, a location $[i, j]$ within the string w, and a regular expression r. The second one is algebraic and uses an algebra of "match-languages" to interpret the regular expressions. A match-language is a set of triples of the form (w, i, j), which specify a

$$
\begin{aligned}
w, [i, j] \models \varepsilon &\iff i = j \\
w, [i, j] \models p &\iff j = i + 1 \text{ and } p(w(i)) = 1 \\
w, [i, j] \models r_1 + r_2 &\iff w, [i, j] \models r_1 \text{ or } w, [i, j] \models r_2 \\
w, [i, j] \models r_1 \cdot r_2 &\iff \text{there is } k \text{ with } i \leq k \leq j \text{ such that } w, [i, k] \models r_1 \text{ and } w, [k, j] \models r_2 \\
w, [i, j] \models r^* &\iff i = j \text{ or there is } k \text{ with } i < k \leq j \text{ such that } w, [i, k] \models r \text{ and } w, [k, j] \models r^* \\
w, [i, j] \models (?\!> r) &\iff i = j \text{ and } w, [i, |w|] \models r \\
w, [i, j] \models (?\!\not> r) &\iff i = j \text{ and } w, [i, |w|] \not\models r \\
w, [i, j] \models (?\!< r) &\iff i = j \text{ and } w, [0, i] \models r \\
w, [i, j] \models (?\!\not< r) &\iff i = j \text{ and } w, [0, i] \not\models r
\end{aligned}
$$

Fig. 1. Formal semantics of regular expressions with lookaround (lookahead and lookbehind). The *satisfaction relation* $\models$ relates a string $w \in \Sigma$, a location $[i, j]$ with $0 \leq i \leq j \leq |w|$, and a regular expression r.

string w and a location $[i, j]$ within it. For each regular construct, we define an associated semantic operation for the algebra of match-languages.

We also consider a natural notion of equivalence between regular expressions, which essentially amounts to equality of their denotations, and establish several useful equivalences. These include the equivalences given by Kleene algebra [Kozen 1994], as well as several other equivalences for simplifying lookaround assertions.

Definition 1 (Regular Expressions with Lookaround). Let Σ be an alphabet, and $\mathcal{P}$ be a set of decidable predicates over Σ, i.e., functions of type $\Sigma \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$. The set $\mathsf{LReg}(\Sigma)$ of *regular expressions* (regexes) with lookaround is defined by the following grammar:

$$
\begin{aligned}
r, r_1, r_2 ::= {} & \varepsilon \mid && \text{[empty string]} \\
& p \in \mathcal{P} \mid && \text{[character class / predicate]} \\
& r_1 + r_2 \mid && \text{[nondeterministic choice / union]} \\
& r_1 \cdot r_2 \mid && \text{[concatenation]} \\
& r^* \mid && \text{[Kleene star / iteration]} \\
& (?\!> r) \mid && \text{[positive lookahead]} \\
& (?\!\not> r) \mid && \text{[negative lookahead]} \\
& (?\!< r) \mid && \text{[positive lookbehind]} \\
& (?\!\not< r) && \text{[negative lookbehind]}
\end{aligned}
$$

Expressions of the form $(?\!> r)$ and $(?\!\not> r)$ are called *lookahead assertions*. Similarly, expressions of the form $(?\!< r)$ and $(?\!\not< r)$ are called *lookbehind assertions*. Expressions of the form $(?\!> r)$, $(?\!\not> r)$, $(?\!< r)$ and $(?\!\not< r)$ are collectively referred to as *lookaround assertions*.

We write $|w|$ to denote the length of a string w. The empty string (i.e., the string of length 0) is denoted by ε. For $w \in \Sigma^*$, we will call a pair $[i, j]$ with $0 \leq i \leq j \leq |w|$ a *location* in w. A *position* in $|w|$ is an index in the range $0, 1, \ldots, |w|$. We write $w(i)$ for the letter at location $[i, i + 1]$ in w.

Definition 2 (Formal Semantics of Lookaround). Let $w \in \Sigma^*$ be a string, i and j be integers satisfying $0 \leq i \leq j \leq |w|$, and $r \in \mathsf{LReg}(\Sigma)$ be a regular expression. We write $w, [i, j] \models r$ to denote that the regular expression r *has a match* in w at the location $[i, j]$. The relation $\models$ is defined by induction as shown in Fig. 1. The characteristic function χ of $\models$ is defined as follows:

$$
\chi(w, [i, j], r) = \begin{cases} 1, & \text{if } w, [i, j] \models r \\ 0, & \text{if } w, [i, j] \not\models r. \end{cases}
$$

We also define $\llbracket r \rrbracket = \{w \in \Sigma^* \mid w, [0, |w|] \models r\}$. That is, $\llbracket r \rrbracket$ is the set of all strings that match r.

Notice in Fig. 1 that lookaround assertions can only hold at locations of the form $[i, i]$, i.e., locations of length 0. Such locations are essentially positions in the string. For this reason, we can think of lookaround assertions as holding (or not) at positions within the string.

A *decomposition* of a location $[i, j]$ (where $i \leq j$) is a nonempty finite sequence of locations $[i_1, j_1], [i_2, j_2], \dots, [i_n, j_n]$ with $i_1 = i$, $j_n = j$, and $j_k = i_{k+1}$ for every $k = 1, 2, \dots, n-1$.

Claim 3. Let $r \in \mathsf{LReg}(\Sigma)$, $w \in \Sigma^*$, and $0 \leq i \leq j \leq |w|$. Then, $w, [i, j] \models r^*$ iff $i = j$ or there exists a decomposition $[i_1, j_1], [i_2, j_2], \dots, [i_n, j_n]$ of $[i, j]$ such that $w, [i_k, j_k] \models r$ for every $k = 1, \dots, n$.

Proof. The left-to-right direction is shown by induction on $j - i$. For the right-to-left direction, we argue by induction on the size $n \geq 1$ of the decomposition. There is one important observation in the proof for the step case: if the last pair $[i_{n+1}, j_{n+1}]$ of the decomposition is of length 0 (that is, $i_{n+1} = j_{n+1}$), then it can be removed. □

Claim 3 serves as a sanity check for the semantic definition of Fig. 1 for Kleene star. Notice that in the definition of $w, [i, j] \models r^*$ we consider r matching only over locations of length at least 1 (we require that $i < k$ in the location $[i, k]$). Claim 3 establishes that this restriction is without loss of generality.

Example 4. Consider the string $w = bbbcabbcbbdbbbc$ with $|w| = 15$. The regular expression $r_1 = bc$ has 3 matches in w at locations $[2, 4]$, $[6, 8]$, and $[13, 15]$. These matches are indicated below by underlining the substrings corresponding to their locations:

$$b\ b\ \underline{b\ c}\ a\ b\ \underline{b\ c}\ b\ b\ d\ b\ b\ \underline{b\ c}$$

The expression $r_1 = (?\!<\Sigma^* a \Sigma^*) bc$ has 2 matches in w at locations $[6, 8]$ and $[13, 15]$. Notice that an occurrence of bc within w matches r_1 only if it is preceded by an occurrence of the letter a. Similarly, the expression $r_2 = bc(?\!>\Sigma^* d \Sigma^*)$ has 2 matches in w at locations $[2, 4]$ and $[6, 8]$. An occurrence of bc within w matches r_2 only if it is followed by an occurrence of the letter d. Finally, the expression $r_3 = (?\!<\Sigma^* a \Sigma^*) bc (?\!>\Sigma^* d \Sigma^*)$ has 1 match in w at location $[6, 8]$.

Example 5. Suppose that $a_1, a_2, \dots, a_n$ are letters of the alphabet. The regular expression $r_i = (?\!>\Sigma^* a_i \Sigma^*)$ asserts that there is some occurrence of a_i from the current position to the end of the string. Let us consider now the expression

$$r = r_1 r_2 \cdots r_n = (?\!>\Sigma^* a_1 \Sigma^*) \cdot (?\!>\Sigma^* a_2 \Sigma^*) \cdots (?\!>\Sigma^* a_n \Sigma^*).$$

For a string w, we have that $w, [0, 0] \models r$ iff the string w contains at least one occurrence of each one of the letters $a_1, a_2, \dots, a_n$. This example shows that lookarounds can be used to encode certain kinds of intersection: $w, [0, 0] \models r$ iff w matches the regular expression $(\Sigma^* a_1 \Sigma^*) \cap (\Sigma^* a_2 \Sigma^*) \cap \cdots \cap (\Sigma^* a_n \Sigma^*)$, where $\cap$ is the intersection operation on regular expresssions (which is, of course, interpreted as intersection on languages).

For a regular expression r, the notation r^n is an abbreviation for the concatenation $r \cdot r \cdots r$ (n times). The notation $r\{n\}$ is also commonly used to describe the repetition of r exactly n times.

Example 6. Lookarounds are often used to extract substrings in a text by specifying constraints involving the context that the substring occurs in. For instance, one can use the regular expression $[0\text{–}9]\{3\}\text{–}[0\text{–}9]\{3\}\text{–}[0\text{–}9]\{4\}$ to find a telephone number. Usually, the first three digits of the phone number are the area code. To extract just the area code from a telephone number, one can use the regular expression $[0\text{–}9]\{3\}(?\!>\text{–}[0\text{–}9]\{3\}\text{–}[0\text{–}9]\{3\})$ that contains a lookahead assertion.

Another example that shows the usefulness of lookbehind expressions is the extraction of an email address domain. Suppose $\alpha = [0\text{–}9A\text{–}Za\text{–}z]$ is a predicate that contains the alphanumeric characters. One can use the regular expression $\alpha^*@\alpha^*.\alpha^*$ to match email addresses. To extract the domain of the email address, one can write the expression $(?\!<\alpha^*@)\alpha^*.\alpha^*$. Interestingly, this regex is not allowed by the PCRE standard, which disallows lookbehinds that could extend over a location of unbounded length. The algorithm we will present later in Section 4 does not have this limitation.

Definition 7 (Algebraic Semantics of Lookaround). If w is a string over Σ and $[i, j]$ is a location in w, then we call the triple (w, i, j) a *(string) slice* over Σ. We define

$$\mathrm{Slices}(\Sigma) = \{(w, i, j) \mid w \in \Sigma^* \text{ and } 0 \le i \le j \le |w|\},$$

which is the set of all string slices over Σ. A *match-language* over an alphabet Σ is a subset of $\mathrm{Slices}(\Sigma)$. For match-languages A, B we define the operations of concatenation $\cdot$, Kleene iteration * and lookaround as follows:

$$\begin{aligned}
A \cdot B &= \{(w, i, k) \mid (w, i, j) \in A \text{ and } (w, j, k) \in B \text{ for some } j\} \\
1_{\mathcal{M}} &= \{(w, i, i) \mid w \in \Sigma^* \text{ and } 0 \le i \le |w|\} \\
A^* &= \textstyle\bigcup_{n\ge 0} A^n, \text{ where } A^0 = 1_{\mathcal{M}} \text{ and } A^{n+1} = A^n \cdot A \text{ for every } n \ge 0 \\
(?\!> A) &= \{(w, i, i) \mid (w, i, |w|) \in A\} \\
(?\!\not> A) &= \{(w, i, i) \mid (w, i, |w|) \notin A\} \\
(?\!< A) &= \{(w, i, i) \mid (w, 0, i) \in A\} \\
(?\!\not< A) &= \{(w, i, i) \mid (w, 0, i) \notin A\}
\end{aligned}$$

For a regular expression r, we define its *match-language* $\mathcal{M}(r)$ as follows:

$$\mathcal{M}(r) = \{(w, i, j) \mid w, [i, j] \models r\}. \tag{1}$$

If $(w, i, j) \in \mathcal{M}(r)$, then we say that the slice (w, i, j) *matches* the expression r and also that r *recognizes* the slice (w, i, j). We say that the regular expressions r and r' are *equivalent*, and we write $r \equiv r'$, if they have equal match-languages, i.e., $\mathcal{M}(r) = \mathcal{M}(r')$.

Lemma 8 (Match-Language). The following hold for the match-languages of regular expressions with lookaround: $\mathcal{M}(\varepsilon) = 1_{\mathcal{M}}$, and

$$\begin{aligned}
\mathcal{M}(p) &= \{(w, i, i+1) \mid i < |w| \text{ and } p(w(i)) = 1\} & \mathcal{M}((?\!> r)) &= (?\!> \mathcal{M}(r)) \\
\mathcal{M}(r + s) &= \mathcal{M}(r) \cup \mathcal{M}(s) & \mathcal{M}((?\!\not> r)) &= (?\!\not> \mathcal{M}(r)) \\
\mathcal{M}(r \cdot s) &= \mathcal{M}(r) \cdot \mathcal{M}(s) & \mathcal{M}((?\!< r)) &= (?\!< \mathcal{M}(r)) \\
\mathcal{M}(r^*) &= \mathcal{M}(r)^* & \mathcal{M}((?\!\not< r)) &= (?\!\not< \mathcal{M}(r))
\end{aligned}$$

for every predicate p and all regular expressions r, s.

Proof. Let p be a predicate over the alphabet Σ. Let RHS be the right-hand side of the equality for $\mathcal{M}(p)$ shown above. For an arbitrary slice (w, i, j) the following equivalences hold: $(w, i, j) \in \mathcal{M}(p)$ iff $w, [i, j] \models p$ iff ($j = i + 1$ and $p(w(i)) = 1$) iff ($j = i + 1$ and $(w, i, i + 1) \in$ RHS) iff $(w, i, j) \in$ RHS. The rest of the cases are handled with similar arguments. We will just note that Claim 3 is useful for establishing that $\mathcal{M}(r^*) = \mathcal{M}(r)^*$. □

Informally, Lemma 8 says that the semantics of Fig. 1 is equivalent to an algebraic semantics in terms of match-languages and operations on them (namely, the operations defined earlier).

Observation 9 (Relationship to PCRE). The lookaround semantics presented in Fig. 1 is different from PCRE [The PCRE2 Developers 2023]. When the lookahead $(?\!> r)$ has a match at location $[i, i]$ in a string w, then this means that r matches at location $[i, |w|]$ which extends to the end of the string. For PCRE-style positive lookahead $(?{=}\,r)$, negative lookahead $(?!\,r)$, positive lookbehind $(?{<=}\,r)$ and negative lookbehind $(?{<!}\,r)$, the semantics can be expressed as follows:

$$\begin{aligned}
w,[i,j] \models (?{=}\,r) &\iff i=j \text{ and } w,[i,k] \models r \text{ for some } k \in [j,|w|] \\
w,[i,j] \models (?!\,r) &\iff i=j \text{ and } w,[i,k] \not\models r \text{ for every } k \in [j,|w|] \\
w,[i,j] \models (?{<=}\,r) &\iff i=j \text{ and } w,[k,j] \models r \text{ for some } k \in [0,i] \\
w,[i,j] \models (?{<!}\,r) &\iff i=j \text{ and } w,[k,j] \not\models r \text{ for every } k \in [0,i]
\end{aligned}$$

In particular, this means that the PCRE-style lookahead $(?{=}\,r)$ is equivalent to $(?{>}\,r \cdot \Sigma^*)$. Similarly, $(?{<=}\,r)$ is equivalent to $(?{<}\,\Sigma^* \cdot r)$.

From the definition of Fig. 1, we see that $w,[i,j] \models (?{>}\,\varepsilon)$ iff $i=j=|w|$. This means that $(?{>}\,\varepsilon)$ is equivalent to PCRE's *end-of-string anchor*, written as \$, which only matches at the end of the string. Similarly, it holds that $w,[i,j] \models (?{<}\,\varepsilon)$ iff $i=j=0$. So, $(?{<}\,\varepsilon)$ is equivalent to PCRE's *start-of-string anchor*, written as ^, which only matches at the beginning of the string.

The previous observations mean that our lookaround can express PCRE's lookaround and the other way round. Here is a summary of all equivalences:

$$\begin{array}{llll}
(?{=}\,r) \equiv (?{>}\,r\Sigma^*) & (?{<=}\,r) \equiv (?{<}\,\Sigma^* r) & (?{>}\,r) \equiv (?{=}\,r\$) & (?{<}\,r) \equiv (?{<=}\,\hat{}\,r) \\
(?!\,r) \equiv (?{\not>}\,r\Sigma^*) & (?{<!}\,r) \equiv (?{\not<}\,\Sigma^* r) & (?{\not>}\,r) \equiv (?!\,r\$) & (?{\not<}\,r) \equiv (?{<!}\,\hat{}\,r)
\end{array}$$

This discussion also establishes that our syntax is more economical than the syntax of PCRE, because the anchors ^ and \$ can be expressed using our lookaround constructs.

2.1 Equational Properties of Lookaround

Informally, the following lemma says that regular expressions with lookaround satisfy the properties of Kleene algebra [Kozen 1994] for the equivalence relation $\equiv$. We write $r_1 \sqsubseteq r_2$ as abbreviation for $r_1 + r_2 \equiv r_2$. It follows that $r_1 \sqsubseteq r_2$ iff $\mathcal{M}(r_1) \subseteq \mathcal{M}(r_2)$ iff ($w,[i,j] \models r_1$ implies $w,[i,j] \models r_2$ for all strings w and locations $[i,j]$ in w).

Lemma 10 (Equivalences for Regular Expressions). The following properties hold for all regular expressions $r, r_1, r_2, r_3 \in \mathsf{LReg}(\Sigma)$:

(1) $(r_1 \cdot r_2) \cdot r_3 \equiv r_1 \cdot (r_2 \cdot r_3)$ and $\varepsilon \cdot r \equiv r \cdot \varepsilon \equiv r$
(2) $(r_1 + r_2) + r_3 \equiv r_1 + (r_2 + r_3)$, $r_1 + r_2 \equiv r_2 + r_1$, and $r + \emptyset \equiv r$
(3) $r_1 \cdot (r_2 + r_3) \equiv r_1 \cdot r_2 + r_1 \cdot r_3$, and $(r_1 + r_2) \cdot r_3 \equiv r_1 \cdot r_3 + r_2 \cdot r_3$
(4) $\emptyset \cdot r \equiv r \cdot \emptyset \equiv \emptyset$
(5) $\varepsilon + rr^* \sqsubseteq r^*$
(6) $\varepsilon + r^* r \sqsubseteq r^*$
(7) $r_1 \cdot r_2 \sqsubseteq r_2 \implies r_1^* \cdot r_2 \sqsubseteq r_2$
(8) $r_2 \cdot r_1 \sqsubseteq r_2 \implies r_2 \cdot r_1^* \sqsubseteq r_2$

Proof. As a representative case, we consider property (5) and we leave the rest of the cases to the reader. It suffices to show that $\varepsilon \sqsubseteq r^*$ and $rr^* \sqsubseteq r^*$. We omit the proof of $\varepsilon \sqsubseteq r^*$. Let w be an arbitrary string and $[i,j]$ be a location in w. Suppose that $w,[i,j] \models rr^*$. It follows that there exists k such that $w,[i,k] \models r$ and $w,[k,j] \models r^*$. So, there is a decomposition S of $[k,j]$ such that $w,[i',j'] \models r$ for every location $[i',j']$ in S. Define $S' = [i,k] \cdot S$ and notice that S' is a decomposition of $[i,j]$ witnessing that $w,[i,j] \models r^*$. We have thus established that $rr^* \sqsubseteq r^*$. □

Alternatively, Lemma 10 can be established by directly considering the algebraic semantics of regular expressions from Definition 7 and using Lemma 8. Indeed, it suffices to show that the match-languages of Definition 7 together with the constants $\emptyset$, $1_{\mathcal{M}}$ and the operations $\cup$, $\cdot$ and *

form a Kleene algebra. This means that $\emptyset$, $1_{\mathcal{M}}$, $\cup$, $\cdot$ satisfy the axioms of idempotent semirings, and * additionally satisfies the four axioms of Kleene iteration from [Kozen 1994]. As an example, let us consider the equivalence $\varepsilon + r \cdot r^* \equiv r^*$. It can be immediately established by using the interpretation $\mathcal{M}$ of regular expressions in the Kleene algebra of match-languages:

$$\begin{aligned} \varepsilon + r \cdot r^* \equiv r^* &\iff \mathcal{M}(\varepsilon + r \cdot r^*) = \mathcal{M}(r^*) && \text{[def. of } \equiv\text{]} \\ &\iff 1_{\mathcal{M}} + \mathcal{M}(r) \cdot \mathcal{M}(r)^* = \mathcal{M}(r)^*, && \text{[Lemma 8]} \end{aligned}$$

which holds because match-languages form a Kleene algebra.

Lemma 11 (Algebraic Properties of Lookaround). The following properties for lookahead assertions hold (and completely symmetric properties hold for lookbehind assertions):

(1) Concatenation of lookarounds is commutative: $(?\!> r) \cdot (?\!> s) \equiv (?\!> s) \cdot (?\!> r)$.
(2) Idempotence: $(?\!> r) \cdot (?\!> r) \equiv (?\!> r)$.
(3) Kleene iteration over lookarounds is equivalent to ε: $(?\!> r)^* \equiv \varepsilon$.
(4) Union distributes over lookarounds: $(?\!> r + s) \equiv (?\!> r) + (?\!> s)$.
(5) Lookaheads can be flattened: $(?\!> (?\!> r) \cdot s) \equiv (?\!> r) \cdot (?\!> s)$.
(6) Lookaheads can be flattened in the presence of wildcards: $(?\!> r \cdot (?\!> s) \cdot \Sigma^*) \equiv (?\!> r \cdot s)$.
(7) The union of positive and negative lookaheads can be simplified: $(?\!> r) + (?\!\not> r) \equiv \varepsilon$.
(8) Positive and negative lookaheads cannot be matched together: $(?\!> r) \cdot (?\!\not> r) \equiv \emptyset$
(9) For predicates p_1 and p_2: $(?\!> p_1 r_1) p_2 r_2 \equiv (p_1 \cap p_2)(?\!> r_1) r_2$.

Proof. These properties can be proved in a straightforward manner using the formal semantics of lookaround expressions. To demonstrate, we prove the first one.

Suppose $w, [i, j] \models (?\!> r) \cdot (?\!> s)$. Then, there exists $i \le k \le j$ such that $w, [i, k] \models (?\!> r)$ and $w, [k, j] \models (?\!> s)$. By the semantics of lookahead, it must be that $i = k$ and $k = j$. Thus, we have $w, [i, i] \models (?\!> r)$ and $w, [i, i] \models (?\!> s)$. From the semantics of $\cdot$ we obtain that $w, [i, i] \models (?\!> s) \cdot (?\!> r)$. Since $i = j$, this is the same as $w, [i, j] \models (?\!> s) \cdot (?\!> r)$. This shows that $\mathcal{M}((?\!> r) \cdot (?\!> s)) \subseteq \mathcal{M}((?\!> s) \cdot (?\!> r))$. By interchanging the roles of r and s, we can prove the other direction. □

The intuition for property (5) regarding the flattening of lookarounds is that both expressions describe the requirement that both r and s have a match at location $[i, |w|]$ (if we interpret them at position i).

We make some further observations about the simplification of lookaround assertions. We have previously stated that the PCRE expression $(?\!= r)$ is equivalent to $(?\!> r \cdot \Sigma^*)$ in our syntax. On the other hand, we can use equation (6) above to establish the equivalences

$$\begin{aligned} (?\!> r) &\equiv (?\!> r \cdot \varepsilon) && \text{[Lemma 10]} \\ &\equiv (?\!> r (?\!> \varepsilon) \Sigma^*). && \text{[Lemma 11]} \end{aligned}$$

The last expression is the same as the regex $(?\!= r\$)$ in PCRE notation. Thus, our syntax can be translated to PCRE notation by adding the \$ anchor. The main observation is that this fact, which we already knew from Observation 9, is established here purely by equational reasoning, using the properties from Lemma 10 and Lemma 11.

Note that $(?\!> r \cdot (?\!> s))$ cannot be simplified to $(?\!> r \cdot s)$. For example, $(?\!> ab \cdot (?\!> cd))$ cannot be true at any position i because ab has to extend to the end of the string where $(?\!> cd)$ cannot hold. So, this regex cannot be equivalent to $(?\!> abcd)$.

Another caveat is the following: Suppose $u \in [\![r]\!]$ and $v \in [\![s]\!]$. We cannot, in general, expect that $u \cdot v \in [\![r \cdot s]\!]$. This is because lookaround expressions are able to “view” the entire string. As a concrete example, consider $r = (?\!> (aa)^*)$, $s = a^*$, $u = \varepsilon$, $v = a$.

It may be possible to give a complete algebraic axiomatization of simple cases of lookaround (e.g., anchors and word boundaries) using the approach of Kleene algebra with extra equations [Grathwohl et al. 2014b; Kozen 1997; Kozen and Mamouras 2014; Mamouras 2015, 2017]. The axiomatization of general lookaround is more challenging, as it can encode some kinds of intersection.

3 ORACLES FOR LOOKAROUND ASSERTIONS

This section is a stepping stone towards the full algorithm for matching regular expressions with lookaround, which we will present in Section 4. Here, we will see how to match a regular expression over a string, assuming that the truth values of lookaround assertions at all positions can be obtained by querying oracles. To formalize this computation, we introduce a notion of regular expression that includes oracle queries instead of lookaround assertions. We call these "oracle-regexes" or "o-regexes" for brevity. Oracle-regexes are matched using a model of automata that is an extension of classical NFAs. We call these automata "oracle-NFAs" or ONFAs. ONFAs contain transitions that query the oracles. An oracle-transition is taken if and only if the oracle responds with "true", but it does not consume a character from the input string. So, we think of them as oracle-guarded ε-transitions. Matching for o-regexes and ONFAs is not defined w.r.t. plain strings over the input alphabet, because these strings lack information about the responses of the oracles. Instead, we define a semantics w.r.t. to "oracle-strings" or "o-strings", which are pairs of the form $\langle w, \bar{\beta} \rangle$, where w is a string over the alphabet and $\bar{\beta}$ is a sequence of all oracle responses for all positions $0, 1, \dots, |w|$. Similar to classical NFAs, the simulation of an ONFA is done in a single left-to-right pass over the input. At each step, a single character and an oracle valuation (for the position right after the character) are consumed. Each step needs $O(m)$ time, where m is the size of the ONFA.

3.1 Oracle Strings and Oracle Regular Expressions

Suppose V is a finite set of oracle names. In the later algorithmic development, we will be using natural numbers as oracles names, because they are convenient for indexing in arrays. A *V-valuation* is a function of type $V \to \mathbb{B}$, that is, a truth assignment for the oracle names V. We also use the notation $\mathbb{B}^V$ for the set of V-valuations.

Definition 12. An *oracle-string* or *o-string* is a pair $\langle w, \bar{\beta} \rangle$ where $w \in \Sigma^*$ and $\bar{\beta} \in (\mathbb{B}^V)^*$ such that $|\bar{\beta}| = |w| + 1$. The set of o-strings over the alphabet Σ and oracle names V is denoted $O(\Sigma, V)$.

Informally, an oracle-string is a string together with the responses for the oracle queries at every position. Suppose $w = a_0 a_1 \dots a_{n-1}$ and $\bar{\beta} = \beta_0 \beta_1 \beta_2 \dots \beta_n$. The character a_i has the oracle valuations β_{i-1} and β_i right before and right after it respectively.

When we slice an o-string, we must take into account the oracle valuations at the boundaries. Suppose we have an oracle-string $\langle w, \bar{\beta} \rangle \in O(\Sigma, V)$ with $w = a_0 a_1 \dots a_{n-1}$ and $\bar{\beta} = \beta_0 \beta_1 \dots \beta_n$. For $0 \leq i \leq j \leq n$, we define the *slice* of $\langle w, \bar{\beta} \rangle$ at location $[i, j]$, denoted by $\langle w, \bar{\beta} \rangle [i, j]$, as the o-string $\langle w', \bar{\beta}' \rangle$ where $\bar{\beta}' = \beta_i \beta_{i+1} \dots \beta_{j-1} \beta_j$ and $w' = a_i a_{i+1} \dots a_{j-1}$. In particular, when $i = j$, w' is the empty string ε and $\bar{\beta}'$ is the singleton sequence β_i.

The concatenation operation on $O(\Sigma, V)$ needs to be defined carefully so that it matches the way we intend to use o-strings. The concatenation of two elements of $O(\Sigma, V)$ is only defined if the oracle valuations agree. Formally, suppose $\langle w_1, \bar{\beta}_1 \cdot \beta \rangle, \langle w_2, \gamma \cdot \bar{\beta}_2 \rangle \in O(\Sigma, V)$ with $\beta, \gamma \in \mathbb{B}^V$, then $\langle w_1, \bar{\beta}_1 \cdot \beta \rangle \cdot \langle w_2, \gamma \cdot \bar{\beta}_2 \rangle$ is defined iff $\beta = \gamma$. The concatenation $\langle w_1, \bar{\beta}_1 \cdot \beta \rangle \cdot \langle w_2, \beta \cdot \bar{\beta}_2 \rangle$ is defined to be $\langle w_1 w_2, \bar{\beta}_1 \cdot \beta \cdot \bar{\beta}_2 \rangle$. We extend this definition in a natural way to concatenation of sets of o-strings. Kleene iteration of an o-string (or a set of o-strings) is also defined in an analogous manner, respecting the agreement of oracle valuations at concatenation boundaries.

Definition 13 (Oracle Regular Expressions). The set $\mathrm{OReg}(\Sigma, V)$ of *oracle regular expressions* (or o-regexes) over alphabet Σ and oracle names V is defined with the following grammar:

$$r, r_1, r_2 ::= \varepsilon \mid p \mid \mathrm{Q}^{\sigma}_{\varepsilon}(v) \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^*,$$

where $p \in \mathcal{P}$ is a predicate over Σ (character class), $\sigma \in \{+, -\}$ is the *sign* of an oracle query, and $v \in V$ is an oracle name.

Instead of lookaround assertions as in $\mathsf{LReg}(\Sigma)$, oracle regular expressions have queries of the form $\mathrm{Q}^{+}_{\varepsilon}(v)$ (positive queries) and $\mathrm{Q}^{-}_{\varepsilon}(v)$ (negative queries).

Example 14 (O-Regexes)**.** The o-regex $abcd \cdot \mathrm{Q}^{+}_{\varepsilon}(0)$ describes a pattern that includes the signature $abcd$ and additionally has to satisfy a positive lookaround assertion right after it. The o-regex does not specify the assertion itself, it only contains a reference to an assertion that should be provided separately. The o-regex $\mathrm{Q}^{-}_{\varepsilon}(0) \cdot hello \cdot \mathrm{Q}^{-}_{\varepsilon}(1)$ describes the pattern *hello*, which additionally has to satisfy two negative lookaround assertions right before h and right after o.

Let β be an oracle valuation and $\mathrm{Q}^{\sigma}_{\varepsilon}(v)$ be an oracle query. We say that β *satisfies* $\mathrm{Q}^{\sigma}_{\varepsilon}(v)$, and we write $\beta \models \mathrm{Q}^{\sigma}_{\varepsilon}(v)$, if ($\sigma = +$ and $\beta[v] = 1$) or ($\sigma = -$ and $\beta[v] = 0$).

Every expression $r \in \mathrm{OReg}(\Sigma, V)$ denotes a language of oracle strings, i.e., a subset of $O(\Sigma, V)$, written as $\llbracket r \rrbracket$. This is defined inductively as follows:

$$\llbracket \varepsilon \rrbracket = \{\langle \varepsilon, \beta \rangle \mid \beta \in \mathbb{B}^V\} \qquad \llbracket r_1 + r_2 \rrbracket = \llbracket r_1 \rrbracket \cup \llbracket r_2 \rrbracket$$

$$\llbracket p \rrbracket = \{\langle a, \beta\gamma \rangle \mid a \in \Sigma \text{ with } p(a) = 1 \text{ and } \beta, \gamma \in \mathbb{B}^V\} \qquad \llbracket r_1 \cdot r_2 \rrbracket = \llbracket r_1 \rrbracket \cdot \llbracket r_2 \rrbracket$$

$$\llbracket \mathrm{Q}^{\sigma}_{\varepsilon}(v) \rrbracket = \{\langle \varepsilon, \beta \rangle \mid \beta \in \mathbb{B}^V \text{ and } \beta \models \mathrm{Q}^{\sigma}_{\varepsilon}(v)\} \text{ for } \sigma \in \{+, -\} \qquad \llbracket r^* \rrbracket = \llbracket r \rrbracket^*$$

for all $p \in \mathcal{P}$, $v \in V$, and $r, r_1, r_2 \in \mathrm{OReg}(\Sigma, V)$. We also define the satisfaction relation as follows:

$$w, \bar{\beta}, [i, j] \models r \iff \langle w, \bar{\beta} \rangle [i, j] \in \llbracket r \rrbracket.$$

In the definition above, we are using the o-string slicing operation introduced earlier.

3.2 Choosing appropriate oracle valuations

Later in this section, we will prove Lemma 21, which formalizes the connection between OReg and LReg. This is crucial for the efficient matching algorithm described in the following section.

Suppose $r \in \mathsf{LReg}(\Sigma)$ and $w \in \Sigma^*$. We say that s is a *lookaround assertion* of r if (1) s is a subexpression of r, (2) s is a lookaround assertion. We say that s is a *maximal lookaround assertion* of r if (1) it is a lookaround assertion of r, and (2) it does not occur underneath a lookaround operator in r.

Example 15. Consider the regex r = `((?!  )(?!(?<![0-9])0)[0-9 ])+`, written using PCRE notation. The subexpressions `(?!  )` and `(?!(?<![0-9])0)` are the two maximal lookaround assertions of r. Notice that `(?<![0-9])` is a lookaround assertion of r, but it is not maximal.

For a type A, we write $\mathrm{Vect}(A)$ for the type of finite vectors (i.e., sequences) over A. If $x : \mathrm{Vect}(A)$, we write $x.\mathrm{len}()$ for its length and $x[i]$ for the i-th element of the vector, where $i = 0, 1, \ldots, x.\mathrm{len}() - 1$. The empty vector is written as $[\ ]$. If $x, y : \mathrm{Vect}(A)$, then $x \cdot y : \mathrm{Vect}(A)$ is their concatenation.

The finite type $\{\mathsf{L2R}, \mathsf{R2L}\}$ has two inhabitants that are used to indicate the direction of ONFA computation over an o-string. The element $\mathsf{L2R}$ (resp., $\mathsf{R2L}$) indicates a left-to-right (resp., right-to-left) pass, which is used for computing lookbehind (resp., lookahead) assertions.

The definition of the "shallow decomposition" of a regex (LReg) that follows (Definition 16) is meant to separate the "main" part of the regex from the maximal lookaround assertions that it contains. The main part is expressed as an oracle-regex (OReg) that contains references to the maximal lookaround assertions.

Definition 16 (Shallow Decomposition). For every index $i \in \mathbb{N}$, we define the *shallow decomposition* function $\mathsf{shallow}_i : \mathsf{LReg}(\Sigma) \to \mathsf{OReg}(\Sigma) \times \mathsf{Vect}(\{\mathsf{L2R}, \mathsf{R2L}\}) \times \mathsf{Vect}(\mathsf{LReg}(\Sigma))$ as follows:

$$\begin{aligned}
\mathsf{shallow}_i(r_1 \circ r_2) &= (s_1 \circ s_2, D_1 \cdot D_2, R_1 \cdot R_2), \text{ where} & \mathsf{shallow}_i(r_0) &= (r_0, [\,], [\,]) \text{ for } r_0 = \varepsilon, p \\
(s_1, D_1, R_1) &= \mathsf{shallow}_i(r_1) \text{ and} & \mathsf{shallow}_i((?> r)) &= (\mathsf{Q}^+_\varepsilon(i), [\mathsf{R2L}], [r]) \\
(s_2, D_2, R_2) &= \mathsf{shallow}_{i+|R_1|}(r_j) & \mathsf{shallow}_i((?\not> r)) &= (\mathsf{Q}^-_\varepsilon(i), [\mathsf{R2L}], [r]) \\
\mathsf{shallow}_i(r^*) &= (s^*, D, R), \text{ where} & \mathsf{shallow}_i((?< r)) &= (\mathsf{Q}^+_\varepsilon(i), [\mathsf{L2R}], [r]) \\
(s, D, R) &= \mathsf{shallow}_i(r) & \mathsf{shallow}_i((?\not< r)) &= (\mathsf{Q}^-_\varepsilon(i), [\mathsf{L2R}], [r])
\end{aligned}$$

where p is a character class and $\circ \in \{\cdot, +\}$. The parameter i in $\mathsf{shallow}_i$ is used to give unique names to the oracles used in queries. We also define $\mathsf{shallow}(r) = \mathsf{shallow}_0(r)$.

Example 17. Consider the regex r = `((?!  )(?!(?<![0-9])0)[0-9 ])+` (in PCRE notation). Using the notation of Section 2, the regex is

$$r = ((?\not> \text{␣␣}\Sigma^*)(?\not> (?\not< \Sigma^*[0\text{–}9])0\Sigma^*)[0\text{–}9\text{␣}])^+.$$

Then, $\mathsf{shallow}(r) = (s, [\mathsf{R2L}, \mathsf{R2L}], [r_0, r_1])$, where

$$r_0 = \text{␣␣}\Sigma^* \qquad r_1 = (?\not< \Sigma^*[0\text{–}9])0\Sigma^* \qquad s = (\mathsf{Q}^-_\varepsilon(0) \cdot \mathsf{Q}^-_\varepsilon(1) \cdot [0\text{–}9\text{␣}])^+.$$

Notice that the shallow decomposition of r extracts signs, directions, and expressions from the maximal lookaround assertions of r.

The oracle-arity $\mathsf{oarity}(r)$ of a regular expression r, defined below in Definition 18, is the number of subterms that are lookarounds that are not subterms of lookarounds (i.e., the number of maximal lookarounds).

Definition 18 (Oracle-Arity, and Oracle-Projection, Oracle-Matrix). The *oracle-arity* or *o-arity* of an expression $r \in \mathsf{LReg}(\Sigma)$ is defined inductively as follows:

$$\begin{aligned}
\mathsf{oarity}(\varepsilon) &= 0 & \mathsf{oarity}(r_1 \circ r_2) &= \mathsf{oarity}(r_1) + \mathsf{oarity}(r_2) \text{ for } \circ \in \{+, \cdot\} \\
\mathsf{oarity}(p) &= 0 & \mathsf{oarity}(r^*) &= \mathsf{oarity}(r)
\end{aligned}$$

and $\mathsf{oarity}(r) = 1$ when r is a lookaround assertion.

Let $r \in \mathsf{LReg}(\Sigma)$ and $i \in \mathbb{N}$. Define $\mathsf{oproj}_i(r) = s$, where $(s, D, R) = \mathsf{shallow}_i(r)$. The *oracle-projection* or *o-projection* of $r \in \mathsf{LReg}(\Sigma)$ is $\mathsf{oproj}(r) = \mathsf{oproj}_0(r)$.

Let $r \in \mathsf{LReg}(\Sigma)$ be a regular expression and $(s, D, R) = \mathsf{shallow}(r)$ be its shallow decomposition. Let $w \in \Sigma^*$. We define the *oracle matrix* for r and w, denoted by $\mathsf{Mat}(r, w) : \mathsf{Vect}(\mathsf{Vect}(\mathbb{B}))$, to be a vector of $k = \mathsf{oarity}(r)$ Boolean tapes of length $|w| + 1$ each. At the (v, i) entry of the matrix $\mathsf{Mat}(r, w)$, we note whether the v-th expression matches at position i in the string w. More formally,

$$\mathsf{Mat}(r, w)[v][i] = \begin{cases} \chi(w, [0, i], R[v]), & \text{if } D[v] = \mathsf{L2R} \\ \chi(w, [i, |w|], R[v]), & \text{if } D[v] = \mathsf{R2L}. \end{cases}$$

for $v = 0, \ldots, k-1$ and $i = 0, \ldots, |w|$. Each vector $\mathsf{Mat}(r, w)[v]$ (size $|w| + 1$) is called an *oracle tape.*

Let $r \in \mathsf{LReg}(\Sigma)$. Intuitively, $\mathsf{oarity}(r)$ is the number of occurrences of maximal lookaround assertions of r. For every $i \in \mathbb{N}$, it holds that $\mathsf{oarity}(r) = |D| = |R|$, where $(s, D, R) = \mathsf{shallow}_i(r)$.

Let $r \in \mathsf{LReg}(\Sigma)$ and $w \in \Sigma^*$. The oracle matrix $\mathsf{Mat}(r, w) : \mathsf{Vect}(\mathsf{Vect}(\mathbb{B}))$ is a $k \times (n+1)$ matrix, where $k = \mathsf{oarity}(r)$ and $n = |w|$. The following properties hold:

(i) $\mathsf{Mat}(r_1 \circ r_2, w) = \mathsf{Mat}(r_1, w) \cdot \mathsf{Mat}(r_2, w)$ for $\circ \in \{+, \cdot\}$
(ii) $\mathsf{Mat}(r^*, w) = \mathsf{Mat}(r, w)$
(iii) $\mathsf{Mat}((?> r), w) = [t]$, where $t[i] = \chi(w, [i, |w|], r)$ for $i = 0, 1, \ldots, |w|$.

(iv) $\mathsf{Mat}((?\!<r), w) = [t]$, where $t[i] = \chi(w, [0, i], r)$ for $i = 0, 1, \ldots, |w|$.

The above properties, together with $\mathsf{Mat}(\varepsilon, w) = [\ ]$ and $\mathsf{Mat}(p, w) = [\ ]$, can be used as an alternative definition for Mat.

Example 19. Consider the regex $r = (?\!< \Sigma^* a \Sigma^*) bc (?\!> \Sigma^* d \Sigma^*)$ from Example 4, which can be written as `(?<=a.*)bc(?=.*d)` in PCRE notation. The lookaround assertions of r are $r_1 = (?\!< \Sigma^* a \Sigma^*)$ and $r_2 = (?\!> \Sigma^* d \Sigma^*)$. Both of them are maximal. The shallow decomposition of r is

$$\mathsf{shallow}(r) = (\mathsf{Q}^+_\varepsilon(0) \cdot bc \cdot \mathsf{Q}^+_\varepsilon(1), [\mathsf{L2R}, \mathsf{R2L}], [\Sigma^* a \Sigma^*, \Sigma^* d \Sigma^*].)$$

Let us use the word $w = bbbcabbcbbdbbbc$ considered in a previous example (Example 4) to illustrate the oracle matrix $T = \mathsf{Mat}(r, w)$. As shown explicitly in the table below,

$$\begin{aligned} T[0][i] = 1 &\iff w, [0, i] \models \Sigma^* a \Sigma^* \text{ and} \\ T[1][i] = 1 &\iff w, [i, |w|] \models \Sigma^* d \Sigma^*. \end{aligned}$$

In this case, the first expression is testing for a presence of an a in the prefix, and the second expression is looking for the presence of a d in the suffix.

string w	b	b	b	c	a	b	b	c	b	b	d	b	b	b	c	
oracle tape $T[0]$	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
oracle tape $T[1]$	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0

Note 20 (Oracle Matrix & Valuation Sequences). Let $V_k = \{0, 1, \ldots, k-1\}$. We use V_k as a set of oracle names that can be used to index in arrays and vectors. Recall that the type of an oracle matrix T is $\mathsf{Vect}(\mathsf{Vect}(\mathbb{B}))$. More specifically, it is a $k \times (n+1)$ matrix, where k is the oracle arity and n is the string length. Informally, the matrix is laid out in rows. If we transpose it, then it would be laid out in columns. In this case, we can view T as a sequence $\bar{\beta} = \beta_0 \beta_1 \ldots \beta_n$ of n, where each $\beta_i : \mathsf{Vect}(\mathbb{B})$ has length k. So, each β_i is essentially a V_k-valuation, i.e., it has type $\mathbb{B}^{V_k}$.

Lemma 21 (Shallow Decomposition). Let $r \in \mathsf{LReg}(\Sigma)$ and $w \in \Sigma^*$. For all $0 \le i \le j \le |w|$,

$$w, [i, j] \models r \iff w, \mathsf{Mat}(r, w), [i, j] \models \mathsf{oproj}(r).$$

Equivalently, $\chi(w, [i, j], r) = \chi(w, \mathsf{Mat}(r, w), [i, j], \mathsf{oproj}(r))$.

Proof. The proof is by induction on the regular expression. Let us consider the case of a positive lookahead assertion $(?\!> r)$. We have that $\mathsf{oproj}((?\!> r)) = \mathsf{Q}^+_\varepsilon(0)$ and $\mathsf{Mat}((?\!> r), w) = [t]$, where $t[i] = \chi(w, [i, |w|], r)$ for every $i = 0, 1, \ldots, |w|$. Now, we have that

$$\begin{aligned} w, [i, j] \models (?\!> r) &\iff i = j \text{ and } w, [i, |w|] \models r \\ &\iff i = j \text{ and } \chi(w, [i, |w|], r) = 1 \\ &\iff i = j \text{ and } t[i] = 1 \\ &\iff i = j \text{ and } \mathsf{Mat}((?\!> r), w)[0][i] = 1 \\ &\iff w, \mathsf{Mat}((?\!> r), w), [i, j] \models \mathsf{Q}^+_\varepsilon(0) \end{aligned}$$

because $[\![\mathsf{Q}^+_\varepsilon(0)]\!] = \{\langle \varepsilon, \beta \rangle\}$ with $\beta(0) = 1$. We leave the rest of the cases to the reader. □

Lemma 21 says that the problem of matching a regular expression $r \in \mathsf{LReg}(\Sigma)$ over a string w can be reduced to matching its oracle-projection $\mathsf{oproj}(r) \in \mathsf{OReg}(\Sigma)$, assuming we have also computed the oracle matrix $\mathsf{Mat}(r, w)$. This assumption means that the truth values of all oracle queries are available.

3.3 NFAs with Oracles Queries

Now we define a class of acceptors for subsets of $O(\Sigma, V)$. These behave like standard nondeterministic finite automata on the part of the o-string that only involves letters from Σ, but additionally has ε-transitions which are guarded by oracle queries. We will see that these acceptors can recognize the o-string languages that are expressed by oracle-regexes.

Definition 22 (Oracle-NFAs). Let Σ be an alphabet and $\mathcal{P}$ a set of predicates over Σ. Let V be a set of oracle names. An *oracle-NFA* (or *ONFA*) $\mathcal{A}$ over the alphabet Σ and oracle names V is a tuple (Q, Δ, I, F), where Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\Delta \subseteq Q \times (\mathcal{P} \cup \mathrm{Q}^+_\varepsilon(V) \cup \mathrm{Q}^-_\varepsilon(V) \cup \{\varepsilon\}) \times Q$ is a transition relation, where $\mathrm{Q}^+_\varepsilon(V) = \{\mathrm{Q}^+_\varepsilon(v) \mid v \in V\}$ and $\mathrm{Q}^-_\varepsilon(V) = \{\mathrm{Q}^-_\varepsilon(v) \mid v \in V\}$. We write $\mathsf{ONFA}(\Sigma, V)$ for the set of all oracle-NFAs over Σ and V.

A transition $(q, \varepsilon, q') \in \Delta$ is an unguarded ε-transition. A transition $(q, \mathrm{Q}^+_\varepsilon(v), q')$ is an ε-transition guarded by a positive oracle query. Similarly, a transition $(q, \mathrm{Q}^-_\varepsilon(v), q')$ is an ε-transition guarded by a negative oracle query. A *path* π in an oracle-NFA $\mathcal{A}$ is a sequence

$$q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} q_2 \xrightarrow{x_2} \cdots \xrightarrow{x_{n-2}} q_{n-1} \xrightarrow{x_{n-1}} q_n$$

of transitions such that $(q_i, x_i, q_{i+1}) \in \Delta$ for every $i = 0, 1, \ldots, n-1$. Every transition (q, x, q') denotes a set of o-strings, denoted $[\![(q, x, q')]\!]$, which we define as follows:

$$[\![(q, \varepsilon, q')]\!] = \{\langle \varepsilon, \beta \rangle \mid \beta \in \mathbb{B}^V\} \qquad [\![(q, \mathrm{Q}^+_\varepsilon(v), q')]\!] = \{\langle \varepsilon, \beta \rangle \mid \beta(v) = 1\}$$
$$[\![(q, p, q')]\!] = \{\langle a, \beta\gamma \rangle \mid \beta, \gamma \in \mathbb{B}^V, a \in \Sigma \text{ and } p(a) = 1\} \qquad [\![(q, \mathrm{Q}^-_\varepsilon(v), q')]\!] = \{\langle \varepsilon, \beta \rangle \mid \beta(v) = 0\}$$

where q and q' are states, $p \in \mathcal{P}$ is a character class (predicate), and $v \in V$ is an oracle name. Similarly, a path $\pi = (q_0, x_0, q_1)(q_1, x_1, q_2) \ldots (q_{n-1}, x_{n-1}, q_n)$ in $\mathcal{A}$ denotes the set $[\![\pi]\!] = [\![(q_0, x_0, q_1)]\!] \cdot [\![(q_1, x_1, q_2)]\!] \cdots [\![(q_{n-1}, x_{n-1}, q_n)]\!]$ of o-strings.

A path in $\mathcal{A}$ is *accepting* if its first state is initial and its last state is final. The set of o-strings *accepted* by $\mathcal{A}$ is defined as $[\![\mathcal{A}]\!] = \bigcup \{[\![\pi]\!] \mid \pi \text{ is an accepting path in } \mathcal{A}\}$. That is, $[\![\mathcal{A}]\!]$ is the union of the denotations of all accepting paths in $\mathcal{A}$.

Lemma 23. Let $r \in \mathsf{OReg}(\Sigma, V)$. Then, there exists some $\mathcal{A} \in \mathsf{ONFA}(\Sigma, V)$ such that $[\![r]\!] = [\![\mathcal{A}]\!]$.

Proof. A variant of Thompson's construction [Thompson 1968] can be used to construct the desired ONFA. Predicates (character classes) in the regular expression would correspond to transitions in the ONFA that are labeled predicates, and oracle queries in the regular expression would correspond to oracle-guarded ε-transitions. Combinators like nondeterministic choice, concatenation and Kleene star can be handled in the usual manner. □

Let q, q' be states of an ONFA $\mathcal{A}$ and β be an oracle valuation. We say that q' is *β-reachable* from q if there exists a path $q_0 \to^{x_0} q_1 \to^{x_1} \cdots \to^{x_{n-1}} q_n$ in $\mathcal{A}$ such that (1) $q = q_0$ and $q' = q_n$, (2) every x_i is either ε or an oracle query, and (3) $\beta \models x_i$ for every x_i that is an oracle query.

Fig. 2 shows an algorithm for matching an oracle regular expression r by compiling it into an ONFA $\mathcal{A}$ and then simulating the execution of the ONFA. We consider both left-to-right and right-to-left matching, as this will be needed later in Section 4 for evaluating lookaround assertions. One important difference between ONFA execution and classical NFA execution is that ε-closure is not sufficient in the case of ONFAs. We have to consider ε-transitions that are either unguarded (similar to NFAs) or guarded by (positive or negative) oracle queries. In order to check which oracle-guarded ε-transitions are enabled, we have to use the oracle valuation for the current position. This is why both `Initial` and `Next` in Fig. 2 take an oracle valuation ($\beta : \mathsf{Vect}(\mathbb{B})$) as an additional argument.

```
// Precondition: $\beta$ is defined on $V$
Function Initial($\mathcal{A}$ : $ONFA(\Sigma, V)$, $\beta$ : $Vect(\mathbb{B})$):
    // Return the set of NFA states that are $\beta$-reachable from the initial states $I$.

// Precondition: $\beta$ is defined on $V$
Function Next($\mathcal{A}$ : $ONFA(\Sigma, V)$, $S \subseteq Q$, $a : \Sigma$, $\beta$ : $Vect(\mathbb{B})$):
    // Let $S'$ be the set of ONFA states that are obtained from $S$ by
    // transitioning on the character $a$.
    // Return the set of ONFA states that are $\beta$-reachable from $S'$.

// Precondition: the oracle matrix $T$ is defined on $V$
// Returns a Boolean tape $t : \mathrm{Vect}(\mathbb{B})$ that contains the matching output for $r$ over $\langle w, \bar{\beta} \rangle$.
// If dir = L2R, then $t[i] = 1$ iff $w, \bar{\beta}, [0, i] \models r$.
// If dir = R2L, then $t[i] = 1$ iff $w, \bar{\beta}, [i, n] \models r$, where $n = |w|$.
Function Match(dir : {L2R, R2L}, $r$ : $OReg(\Sigma, V)$, $w : \Sigma^*$, $T$ : $Vect(Vect(\mathbb{B}))$):
    $\mathbb{N}\ n \leftarrow |w|$   // length of string
    // The vector $T$ contains $k = \mathrm{oarity}(r)$ tapes and each tape $T[i] : \mathrm{Vect}(\mathbb{B})$ has length $n+1$.
    // So, $T$ can be turned into a sequence of oracle valuations.
    $\mathrm{Vect}(\mathrm{Vect}(\mathbb{B}))\ \bar{\beta} \leftarrow \mathrm{transpose}(T)$ // it is not actually necessary to explicitly transpose
    // $\bar{\beta}$ is a sequence of length $n+1$ containing $V_k$-valuations
    // $\langle w, \bar{\beta} \rangle$ is an oracle-string over $\Sigma$ and $V_k$
    assert $|\bar{\beta}| = n + 1$   // we need $n+1$ oracle valuations
    // if dir = R2L, then the regex has to be reversed!!!
    $\mathrm{ONFA}(\Sigma)\ \mathcal{A} \leftarrow$ Oracle-NFA for the oracle-regex $\mathbf{ite}(\mathrm{dir} = \mathrm{L2R}, r, \mathrm{rev}(r))$
    $\mathbb{N}\ i \leftarrow \mathbf{ite}(\mathrm{dir} = \mathrm{L2R}, 0, n)$   // position at which the matching starts
    $S \leftarrow$ Initial($\mathcal{A}, \bar{\beta}[i]$)   // initial powerstate
    $\mathrm{Vect}(\mathbb{B})\ \mathrm{out} \leftarrow [0;\ n+1]$   // vector of size $n+1$, initialized to 0
    if $S \cap F \neq \emptyset$ then $\mathrm{tape}[i] \leftarrow 1$ // $S$ contains a final state
    else $\mathrm{tape}[i] \leftarrow 0$
    $\mathbb{Z}\ d \leftarrow \mathbf{ite}(\mathrm{dir} = \mathrm{L2R}, 1, -1)$ // increment/decrement index based on direction
    for _ = $0, 1, \ldots, n-1$ do // process the input text
        $S \leftarrow$ Next($\mathcal{A}, S, w[i], \bar{\beta}[i+d]$)   // compute the next powerstate
        if $S \cap F \neq \emptyset$ then $\mathrm{tape}[i] \leftarrow 1$ // $S$ contains a final state
        else $\mathrm{tape}[i] \leftarrow 0$
        $i \leftarrow i + d$
    return tape
```

Fig. 2. Algorithm for matching oracle regular expressions using ONFA simulation.

We will continue now to prove the main correctness result for the Match algorithm of Fig. 2. Before we can prove this, we need to consider a semantic property of matching "in reverse". The *reverse* $\mathrm{rev}(r)$ of an oracle regular expression r is defined recursively as follows:

$$\mathrm{rev}(\varepsilon) = \varepsilon \qquad \mathrm{rev}(r_1 + r_2) = \mathrm{rev}(r_1) + \mathrm{rev}(r_2) \qquad \mathrm{rev}(r^*) = \mathrm{rev}(r)^*$$

$$\mathrm{rev}(p) = p \qquad \mathrm{rev}(r_1 \cdot r_2) = \mathrm{rev}(r_2) \cdot \mathrm{rev}(r_1) \qquad \mathrm{rev}(\mathrm{Q}^{\sigma}_{\varepsilon}(v)) = \mathrm{Q}^{\sigma}_{\varepsilon}(v)$$

where $\sigma \in \{+, -\}$. For example, $\mathrm{rev}(a \cdot \mathrm{Q}^{-}_{\varepsilon}(v_1) \cdot d \cdot \mathrm{Q}^{+}_{\varepsilon}(v_2)) = \mathrm{Q}^{+}_{\varepsilon}(v_2) \cdot d \cdot \mathrm{Q}^{-}_{\varepsilon}(v_1) \cdot a$.

Lemma 24 (Reversal). Let $s \in \mathrm{OReg}(\Sigma, V)$, $\langle w, \bar{\beta} \rangle \in \mathcal{O}(\Sigma, V)$, and $0 \leq i \leq j \leq |w|$. Then,

$$w, \bar{\beta}, [i, j] \models s \iff \mathrm{rev}(w), \mathrm{rev}(\bar{\beta}), [|w| - j, |w| - i] \models \mathrm{rev}(s).$$

Proof. The proof is by induction on r. For convenience, we use the following alternative characterization of the satisfaction relation:

$$w, \bar{\beta}, [i, j] \models \varepsilon \iff i = j$$

$$\begin{aligned}
w, \bar{\beta}, [i, j] \models p &\iff j = i + 1 \text{ and } p(w(i)) = 1 \\
w, \bar{\beta}, [i, j] \models r_1 + r_2 &\iff w, \bar{\beta}, [i, j] \models r_1 \text{ or } w, \bar{\beta}, [i, j] \models r_2 \\
w, \bar{\beta}, [i, j] \models r_1 \cdot r_2 &\iff \text{there is } k \text{ with } i \le k \le j \text{ such that } w, \bar{\beta}, [i, k] \models r_1 \text{ and } w, \bar{\beta}, [k, j] \models r_2 \\
w, \bar{\beta}, [i, j] \models r^* &\iff i = j \text{ or there is } k \text{ with } i < k \le j \text{ s.t. } w, \bar{\beta}, [i, k] \models r \text{ and } w, \bar{\beta}, [k, j] \models r^* \\
w, \bar{\beta}, [i, j] \models \mathrm{Q}^{\sigma}_{\varepsilon}(v) &\iff i = j \text{ and } \bar{\beta}[i] \models \mathrm{Q}^{\sigma}_{\varepsilon}(v)
\end{aligned}$$

We will show how to handle the representative case of oracle queries:

$$\begin{aligned}
w, \bar{\beta}, [i, j] \models \mathrm{Q}^{\sigma}_{\varepsilon}(v) &\iff i = j \text{ and } \bar{\beta}[i] \models \mathrm{Q}^{\sigma}_{\varepsilon}(v) \\
&\iff |w| - i = |w| - j \text{ and } \mathsf{rev}(\bar{\beta})[|w| - i] \models \mathrm{Q}^{\sigma}_{\varepsilon}(v) \\
&\iff \mathsf{rev}(w), \mathsf{rev}(\bar{\beta}), [|w| - j, |w| - i] \models \mathsf{rev}(\mathrm{Q}^{\sigma}_{\varepsilon}(v))
\end{aligned}$$

because $\mathsf{rev}(\bar{\beta})[|w| - i] = \bar{\beta}[i]$ and $\mathsf{rev}(\mathrm{Q}^{\sigma}_{\varepsilon}(v)) = \mathrm{Q}^{\sigma}_{\varepsilon}(v)$. The rest of the cases are handled with similar arguments and we therefore omit them. □

Proposition 25 (Correctness of Matching). Let $s \in \mathsf{OReg}(\Sigma, V)$, $w \in \Sigma^*$ with $n = |w|$, and $\bar{\beta}$ be an oracle matrix defined over V. The following hold:

(1) $\mathtt{Match}(\mathtt{L2R}, s, w, \bar{\beta})[i] = \chi(w, \bar{\beta}, [0, i], s)$ and
(2) $\mathtt{Match}(\mathtt{R2L}, s, w, \bar{\beta})[i] = \chi(w, \bar{\beta}, [i, n], s)$

for every $i = 0, 1, \dots, n$.

Proof. Part (1) can be proved with similar arguments as those that justify the simulation of classical NFAs. The only difference is that we need to consider the current oracle valuation in order to see whether an oracle-guarded ε-transition is enabled or not.

For Part (2), the main observation is that the execution of $\mathtt{Match}(\mathtt{R2L}, s, w, \bar{\beta})$ follows the same ONFA simulation steps as $\mathtt{Match}(\mathtt{L2R}, \mathsf{rev}(s), \mathsf{rev}(w), \mathsf{rev}(\bar{\beta}))$ and stores the output bits in reverse order. Let t be the output Boolean tape. From Part (1) we get that $t[i] = \chi(\mathsf{rev}(w), \mathsf{rev}(\bar{\beta}), [0, i], \mathsf{rev}(s))$ for every i. From Lemma 24, we get that $t[i] = \chi(w, \bar{\beta}, [|w| - i, |w|], s)$ for every i. Finally, the execution $\mathtt{Match}(\mathtt{R2L}, s, w, \bar{\beta})$ gives as output the reverse of t, which satisfies $\mathsf{rev}(t)[i] = t[|w| - i] = \chi(w, \bar{\beta}, [i, |w|], s)$ for every i. This concludes the proof. □

The matching algorithm of Fig. 2 proceeds in a single left-to-right (resp., right-to-left) pass over the input string w when dir = L2R (resp., dir = R2L). It performs $O(m)$ work per step, so the total running time is $O(m \cdot n)$, where m is the size of the o-regex and n is the length of the input text.

4 EFFICIENT MATCHING

Using the algorithm for oracle-regex matching from the previous section, we will now describe how regular expressions with lookaround can be efficiently matched. Our algorithm operates on the nested structure of LReg. If there are one or more levels of lookaround, our algorithm makes multiple forward or backward passes on the input string to extract the necessary information.

We have seen in Lemma 21 that by choosing appropriate oracle valuations, we can decide membership in expressions with lookaround, by converting them to oracle expressions. In the previous section, we saw that oracle regular expressions can be realized as ONFAs which behave similarly to standard NFAs but additionally have oracle-guarded ε-transitions. Our algorithm is expressed in terms of a recursive function (`EvalAux` in Fig. 3) which traverses the regular expression recursively. When a lookaround expression is found, the corresponding oracle tape is computed and the lookaround expression is replaced with an oracle query. Ultimately, the resulting o-regex and oracle tapes, which form an oracle matrix, are passed to an ONFA for matching.

```
// Returns a pair $(k, s)$, where
//   (1) $k$ is the oracle-arity of $r$, and
//   (2) $s \in \mathrm{OReg}(\Sigma, V)$ with $V = \{o, o+1, \ldots, o+k-1\}$.
// It must be that $s = \mathrm{oproj}_o(r)$ and $k = \mathrm{oarity}(r)$.
Function EvalAux($r : LReg(\Sigma)$, $w : \Sigma^*$, $o : \mathbb{N}$, $T$ : &mut $Vect(Vect(\mathbb{B}))$):
    switch $r$ do
        case $\varepsilon$ do return $(0, \varepsilon)$
        case $p$ do return $(0, p)$  // predicate (character class)
        case $r_1 \circ r_2$ where $\circ \in \{+, \cdot\}$ do
            $(k_1, s_1) \leftarrow$ EvalAux($r_1, w, o, T$)
            $(k_2, s_2) \leftarrow$ EvalAux($r_2, w, o + k_1, T$)
            return $(k_1 + k_2, s_1 \circ s_2)$
        case $r_1^*$ do
            $(k_1, s_1) \leftarrow$ EvalAux($r_1, w, o, T$)
            return $(k_1, s_1^*)$
        case $(?{>}\, r_1)$ do // positive lookahead
            $T.push$(Eval(R2L, $r_1, w$))
            return $(1, \mathrm{Q}^+_\varepsilon(o))$
        case $(?{\not>}\, r_1)$ do // negative lookahead
            $T.push$(Eval(R2L, $r_1, w$))
            return $(1, \mathrm{Q}^-_\varepsilon(o))$
        case $(?{<}\, r_1)$ do // positive lookbehind
            $T.push$(Eval(L2R, $r_1, w$))
            return $(1, \mathrm{Q}^+_\varepsilon(o))$
        case $(?{\not<}\, r_1)$ do // negative lookbehind
            $T.push$(Eval(L2R, $r_1, w$))
            return $(1, \mathrm{Q}^-_\varepsilon(o))$
// Returns a tape $t : \mathrm{Vect}(\mathbb{B})$ that contains the matching output for $r$ over $w$.
// If $\mathrm{dir} = \mathrm{L2R}$, then $t[i] = 1$ iff $w, [0, i] \models r$.
// If $\mathrm{dir} = \mathrm{R2L}$, then $t[i] = 1$ iff $w, [i, n] \models r$, where $n = |w|$.
Function Eval(dir : {L2R, R2L}, $r : LReg(\Sigma)$, $w : \Sigma^*$):
    Vect(Vect($\mathbb{B}$)) $T \leftarrow [\,]$ // empty vector of tapes
    $(k, s) \leftarrow$ EvalAux($r, w, 0$, &mut $T$)
    // The vector $T$ contains $k = \mathrm{oarity}(r)$ Boolean tapes.
    // Each tape $T[i] : \mathrm{Vect}(\mathbb{B})$ has length $n + 1$, where $n = |w|$.
    return Match(dir, $s, w, T$)
```

Fig. 3. Algorithm for matching regular expressions with lookaround assertions.

Since ONFAs are simulated by maintaining a set of active control states (similarly to NFAs), we are able to compute $\chi(w, [0, i], r) : \mathbb{B}$ for each position i, by running the ONFA in a single left-to-right pass. These truth values are useful as they form an oracle tape that could be used in evaluating an ONFA for a larger subexpression. For lookahead expressions $(?{>}\, r)$, the truth values $\chi(w, [i, |w|], r) : \mathbb{B}$ are required. To compute these values, the ONFA is executed in reverse.

Evaluation Algorithm. The overall algorithm for matching regular expressions with lookaround is shown in Fig. 3. The top-level function is `Eval` and it uses the auxiliary function `EvalAux` to recursively traverse the regular expression. `EvalAux` is similar to the shallow decomposition of Definition 16 and it computes both the oracle-projection and the oracle matrix. It takes four inputs:

(1) a regular expression $r \in \mathrm{LReg}(\Sigma)$ to evaluate,
(2) the input string $w \in \Sigma^*$,

(3) an index $o \in \mathbb{N}$ from which to start numbering the maximal lookaround assertions in r, and
(4) a mutable reference to a vector of Boolean tapes (which will form the Boolean matrix).

The function `EvalAux` returns a pair (k, s), where $k = \mathsf{oarity}(r)$ and $s = \mathsf{oproj}_o(r)$. Moreover, suppose T (resp., T') is the value of the last argument before (resp., after) the execution of `EvalAux`. Then, $T' = T \cdot T_1$, where $T_1 = [t'_0, \dots, t'_{k-1}] = \mathsf{Mat}(r, w)$. Notice that the lookaround cases in `EvalAux` call `Eval`, which in turn calls `EvalAux` to the subexpressions of the lookaround assertions. So, intuitively, `Eval` applies the shallow decomposition recursively to the top-level regex as well as all regexes of lookarounds. The overall computation can be understood as a "deep decomposition", which we will formalize later in Section 5.

Theorem 26 (Correctness of Matching). Let $r \in \mathsf{LReg}(\Sigma)$ and $w \in \Sigma^*$. The following hold:

(1) $\mathtt{Eval}(\mathsf{L2R}, r, w)[i] = 1$ iff $w, [0, i] \models r$ and
(2) $\mathtt{Eval}(\mathsf{R2L}, r, w)[i] = 1$ iff $w, [i, |w|] \models r$

for every position $i = 0, 1, \dots, |w|$.

Proof. For the sake of the proof, we can assume w.l.o.g. that the fourth argument of `EvalAux` is a matrix of type $\mathsf{Vect}(\mathsf{Vect}(\mathbb{B}))$ (i.e., it is passed by value) and `EvalAux` returns a triple (k, s, T), where T is the updated matrix. Properties (1) and (2) can be reformulated equivalently as follows:

(1) $\mathtt{Eval}(\mathsf{L2R}, r, w)[i] = \chi(w, [0, i], r)$ and
(2) $\mathtt{Eval}(\mathsf{R2L}, r, w)[i] = \chi(w, [i, |w|], r)$

for every $i = 0, 1, \dots, |w|$. We also claim the following property:

(3) Let $o \in \mathbb{N}$ and $T : \mathsf{Vect}(\mathsf{Vect}(\mathbb{B}))$. Suppose that $(k, s, T') = \mathtt{EvalAux}(r, w, o, T)$. Then, $k = \mathsf{oarity}(r)$, $s = \mathsf{oproj}_o(r)$ and $T' = T \cdot \mathsf{Mat}(r, w)$.

First, we will show that property (3) implies (1) and (2). Let $(k, s, T) = \mathtt{EvalAux}(r, w, 0, [\,])$. Property (3) says that $k = \mathsf{oarity}(r)$, $s = \mathsf{oproj}_0(r) = \mathsf{oproj}(r)$ and $T = \mathsf{Mat}(r, w)$. We have to examine two cases for the direction argument $\mathsf{dir} \in \{\mathsf{L2R}, \mathsf{R2L}\}$. For the case $\mathsf{dir} = \mathsf{L2R}$, we have that $\mathtt{Eval}(\mathsf{L2R}, r, w) = \mathtt{Match}(\mathsf{L2R}, s, w, T) = \mathtt{Match}(\mathsf{L2R}, \mathsf{oproj}(r), w, \mathsf{Mat}(r, w)) = \mathit{RHS}$. From Proposition 25, $\mathit{RHS}[i] = \chi(w, \mathsf{Mat}(r, w), [0, i], \mathsf{oproj}(r))$ for every $i = 0, 1, \dots, |w|$. So, from Lemma 21, we conclude that $\mathit{RHS}[i] = \chi(w, [0, i], r)$ for every $i = 0, 1, \dots, |w|$. Now, for $\mathsf{dir} = \mathsf{R2L}$ and every $i = 0, 1, \dots, |w|$, we similarly have that

$$\begin{aligned}
\mathtt{Eval}(\mathsf{R2L}, r, w)[i] &= \mathtt{Match}(\mathsf{R2L}, s, w, T)[i] && \text{[algorithm } \mathtt{Eval}\text{]} \\
&= \mathtt{Match}(\mathsf{R2L}, \mathsf{oproj}(r), w, \mathsf{Mat}(r, w))[i] && \text{[property (3)]} \\
&= \chi(w, \mathsf{Mat}(r, w), [i, |w|], \mathsf{oproj}(r)) && \text{[Proposition 25]} \\
&= \chi(w, [i, |w|], r). && \text{[Lemma 21]}
\end{aligned}$$

Since properties (1) and (2) follow from property (3), it suffices to establish property (3). The proof is by induction on r. We will only consider the representative cases of nondeterministic choice and positive lookbehind, since the arguments for the rest of the cases are similar.

For the case $r_1 + r_2$, let $(k_1, s_1, T_1) = \mathtt{EvalAux}(r_1, w, o, T)$ and $(k_2, s_2, T_2) = \mathtt{EvalAux}(r_2, w, o + k_1, T_1)$. Since $k_1 = \mathsf{oarity}(r_1)$ (I.H.) and $k_2 = \mathsf{oarity}(r_2)$ (I.H.), $k_1 + k_2 = \mathsf{oarity}(r_1 + r_2)$. From $s_1 = \mathsf{oproj}_o(r_1)$ (I.H.) and $s_2 = \mathsf{oproj}_{o+k_1}$ (I.H.), we get that $s_1 + s_2 = \mathsf{oproj}_o(r_1 + r_2)$. Finally, $T_1 = T \cdot \mathsf{Mat}(r_1, w)$ (I.H.) and $T_2 = T_1 \cdot \mathsf{Mat}(r_2, w)$ (I.H.) give us that $T_2 = T \cdot \mathsf{Mat}(r_1, w) \cdot \mathsf{Mat}(r_2, w) = T \cdot \mathsf{Mat}(r_1 + r_2, w)$.

For the positive lookbehind assertion $(?\!<r)$, let $(k, s, T') = \mathtt{EvalAux}((?\!<r), w, o, T)$. From the algorithm we see that $k = 1$, $s = \mathsf{Q}^+_\varepsilon(o)$ and $T' = T \cdot [\mathtt{Eval}(\mathsf{L2R}, r, w)]$. Notice that $k = \mathsf{oarity}((?\!<r)) = 1$ and $s = \mathsf{oproj}_o((?\!<r)) = \mathsf{Q}^+_\varepsilon(o)$. We also observe that $\mathsf{Mat}((?\!<r), w) = [t]$, where $t[i] = \chi(w, [0, i], r)$ for every $i = 0, 1, \dots, |w|$. From property (1) we know that $\mathtt{Eval}(\mathsf{L2R}, r, w)[i] = \chi(w, [0, i], r)$ for every i. So, $T' = T \cdot \mathsf{Mat}((?\!<r), w)$. □

Theorem 27 (Efficiency). The matching algorithm `Eval` of Fig. 3 needs $O(m \cdot n)$ time and $O(m \cdot n)$ space, where m is the size of the regular expression and n is the length of the input string.

Proof. For $r \in \mathsf{LReg}(\Sigma)$, define $L(r)$ to be the number of lookaround assertions that occur in r. For the memory bound, notice that at most $L(r)$ Boolean tapes are stored during the execution of the algorithm. Each Boolean tape requires n bits. So, the memory footprint is $O(m \cdot n)$.

For the time bound, let us start with the high-level idea of the proof. The time complexity is dominated by the cost of ONFA matching. `Eval`(r) performs $L(r) + 1$ calls to `Match` in total. The i-th call to `Match` has running time $O(m_i \cdot n)$, where m_i is the size of the i-th o-regex that is used for matching. But $\sum_i m_i = O(m)$, which means that the total running time is $O(m \cdot n)$. The algorithm constructs NFAs/ONFAs of total size $O(m)$, so these constructions do not affect the time complexity.

To make the argument more formal, we need some definitions. For $r \in \mathsf{LReg}(\Sigma)$, define $size_1(r) = |s|$ and $size_2(r) = |r_1| + \cdots + |r_k|$, where $(s, D, R) = \mathsf{shallow}(r)$ and $R = [r_1, \ldots, r_k]$. That is, $size_1(r)$ is the size of the oracle-projection of r, and $size_2(r)$ is the sum of the sizes of the expressions that are inside maximal lookaround assertions. Note that the quantities $size_1(r)$ and $size_2(r)$ would be same if we used $\mathsf{shallow}_i$ (for any i) instead of $\mathsf{shallow}$ in their definitions. It can be proved by induction on r that $|r| = size_1(r) + size_2(r)$. For example, we have that $\mathsf{shallow}((?\!> r)) = (\mathrm{Q}^+_\varepsilon(0), [\mathsf{R2L}], [r])$ and therefore $|(?\!> r)| = 1 + |r| = |\mathrm{Q}^+_\varepsilon(0)| + |r| = size_1((?\!> r)) + size_2((?\!> r))$.

The matching algorithm of Fig. 2 needs time at most $f(r) = c|r|n$, where r is the input o-regex, n is the length of the input text, and c is a constant. We claim that (1) `EvalAux` needs time at most $g(r) = c \cdot size_2(r) \cdot n$ for matching, and (2) `Eval` needs time at most $f(r)$ for matching.

First, we show that property (1) implies property (2). `Eval`(r) calls `EvalAux`(r) and performs matching using the oracle-projection $s = \mathsf{oproj}(r)$. So, the total matching time for `Eval`(r) is bounded above by $g(r)+f(s) = c\cdot size_2(r)\cdot n+c\cdot size_1(r)\cdot n = c\cdot(size_1(r)+size_2(r))\cdot n = c|r|n = f(r)$. Now, we show property (1) by induction on r. For the case $r = r_1 + r_2$ of nondeterministic choice, we see that $size_2(r) = size_2(r_1) + size_2(r_2)$ and `EvalAux`(r) calls `EvalAux`(r_1) and `EvalAux`(r_2). So, the total matching time is bounded above by $g(r_1) + g(r_2) = c \cdot size_2(r_1) \cdot n + c \cdot size_2(r_2) \cdot n = c \cdot (size_2(r_1) + size_2(r_2)) \cdot n = c \cdot size_2(r) \cdot n = g(r)$. For the case $r = (?\!> r_1)$ of a positive lookahead, notice that $size_2(r) = |r_1|$ and `EvalAux`(r) calls `Eval`(r_1). It follows that the matching time is bounded above by $f(r_1) = c|r_1|n = c size_2(r)n = g(r)$. The rest of the cases use similar arguments and we therefore leave them to the reader. □

Intuitive Explanation for Complexity Bound. The main reason behind the $O(n \cdot m)$ bound on time and space is that our algorithm does not construct a single automaton for the whole regular expression, which would potentially cause an exponential blowup. Instead, we construct a collection of small automata (NFAs and ONFAs), each of which corresponds to part of the original regex. The sum of the sizes of all these NFAs/ONFAs is $O(m)$. This is a representation that is as succinct as the original regex, because we do not perform constructions that give rise to large automata. Our algorithm uses only these small automata and (potentially) additional memory for storing oracles tapes, where each oracle tape has size $\Theta(n)$.

For example, let us consider a regex of the form $r'(?\!> r_0)(?\!> r_1)\cdots(?\!> r_{k-1})$, where r' and $r_0, r_1, \ldots, r_{k-1}$ are lookaround-free. Our algorithm constructs (1) NFAs $\mathcal{A}_0, \ldots, \mathcal{A}_{k-1}$ for the regexes $r_0, \ldots, r_{k-1}$, and (2) an ONFA $\mathcal{A}'$ for the top-level oracle-regex $r' \cdot \mathrm{Q}^+_\varepsilon(0) \cdots \mathrm{Q}^+_\varepsilon(k-1)$. Let m_i be the size of $\mathcal{A}_i$ and m' be the size of $\mathcal{A}'$. The total size $m_0+\cdots+m_{k-1}+m'$ of the automata is proportional to the size of the original regex. We do *not* construct a product automaton, even though the lookarounds describe some kind of intersection, as the product automaton would be of exponential size. Our algorithm simulates these small automata. Let n be the length of the input text. First, each NFA $\mathcal{A}_i$ is reversed and simulated with a right-to-left pass over the input (work proportional to $m_i n$).

Then, the ONFA $\mathcal{A}'$ is simulated with a left-to-right pass over the input (work proportional to $m'n$). So, the total work performed is proportional to $m_0 n + \cdots + m_{k-1} n + m'n = (m_0 + \cdots + m_{k-1} + m')n$.

The only pre-processing performed by our algorithm happens in the `Match` procedure of Fig. 2 (see line 7). It involves Thompson-style constructions to obtain NFAs and ONFAs from oracle-regexes. These constructions can be performed in time $O(m)$, where m is the size of the regex.

Extracting Matches. The literature on automata and formal languages generally focuses on the membership problem for regular expressions: given $w \in \Sigma^*$ and $r \in \mathsf{LReg}(\Sigma)$, is it the case that $w, [0, |w|] \models r$? To answer this question, we can simply look at the last element of $\mathtt{Eval}(\mathtt{L2R}, w, r)$. However, regular expressions with lookaround are often used to specify additional constraints on the context in which a substring appears without capturing the context itself. For instance, telephone numbers have the form $xxx{-}yyy{-}zzzz$, where xxx is the area code. One might use the regular expression `[0-9]{3}(?=-[0-9]{3}-[0-9]{4})` to extract the area code. For such a task, the match extraction problem is of more interest than the membership problem. A match for r in w is a pair $[i, j]$ of indices with $0 \le i \le j \le |w|$ such that $w, [i, j] \models r$. The *leftmost longest* match is the longest out of the leftmost matches (it can be easily seen that it is unique). The computational problem of extracting matches (and sub-matches) has been considered before (see, e.g, the notes of Cox [2010]). The following two-step procedure uses the algorithm of Fig. 3 to efficiently extract the leftmost longest match for a given regular expression:

(1) Find the smallest index i such that $w, [i, |w|] \models r \cdot \Sigma^*$ using the output of $\mathtt{Eval}(\mathtt{R2L}, w, r \cdot \Sigma^*)$.
(2) Then, find the largest j with $w, [i, j] \models r$ using $\mathtt{Eval}(\mathtt{L2R})$ on the suffix of w starting at i.

We can also consider match extraction when a match other than the leftmost longest one is preferred.

Lookaround and Temporal Monitoring. The use of lookaround in regular expressions is reminiscent of the use of temporal connectives in temporal logic, which has found applications in runtime verification and online monitoring [Bartocci et al. 2018]. More specifically, lookahead (resp., lookbehind) is similar to future-time (resp., past-time) temporal connectives. The problem of (online or offline) temporal monitoring is analogous to the matching problem for regular expressions. It seems possible that the compositional regex matching algorithm of Fig. 3 can be combined with efficient and modular algorithms for temporal monitoring (see, e.g., [Chattopadhyay and Mamouras 2020; Dokhanchi et al. 2014; Maler et al. 2008; Mamouras et al. 2021a,b, 2023; Mamouras and Wang 2020; Thati and Roşu 2005]) in order to support more expressive temporal specification formalisms.

5 PERFORMANCE OPTIMIZATIONS

The algorithm of Fig. 3, presented in the previous section, provides strong worst-case performance guarantees. The upper bound $O(m \cdot n)$ for the running time is the same as the complexity of Thompson's algorithm [Thompson 1968], which only handles classical regular expressions (i.e., no lookaround). In order to provide a practical implementation, we will introduce in this section three performance optimizations that can reduce both the amount of work and memory needed for some regular expressions. We will see later in Section 6 through an experimental evaluation that these optimizations are significant in practice.

5.1 Common Assertion Elimination

In Definition 16, we introduced the concept of shallow decomposition of a regular expression r, which allows us to reduce the evaluation of r to the simulation of an ONFA, assuming that we have access to oracles that resolve the truth values of the lookaround assertions. Computationally, the algorithm of Fig. 3 performs a shallow decomposition with each invocation of `Eval`. In order to compute the oracle tapes, `Eval` is applied recursively whenever a lookaround assertion is

```
// Returns a tape $t : \mathsf{Vect}(\mathbb{B})$ that contains the matching output for $r$ over $w$.
// If dir = L2R, then $t[i] = 1$ iff $w, [0, i] \models r$.
// If dir = R2L, then $t[i] = 1$ iff $w, [i, n] \models r$, where $n = |w|$.
Function EvalDeep(dir : {L2R, R2L}, $r : LReg(\Sigma)$, $w : \Sigma^*$):
    // compute the deep decomposition of the regex $r$
    $\mathsf{OReg}(\Sigma) \times \mathsf{Vect}(\{\mathsf{L2R}, \mathsf{R2L}\}) \times \mathsf{Vect}(\mathsf{OReg}(\Sigma))\ (s, D, R) \leftarrow \mathsf{deep}(r)$
    $\mathbb{N}\ k \leftarrow R.\mathrm{len}()$ // number of lookaround assertions
    $\mathsf{Vect}(\mathsf{Vect}(\mathbb{B}))\ T \leftarrow [[\,];\ k]$ // vector of $k$ empty Boolean tapes
    for $i = 0, 1, \ldots, k-1$ do // process lookaround assertions in topological order
        $T[i] \leftarrow$ Match($D[i], R[i], w, T$)
    // Now, the vector $T$ contains $k$ tapes and each tape $T[i] : \mathsf{Vect}(\mathbb{B})$ has length $n + 1$.
    return Match(dir, $s, w, T$)
```

Fig. 4. Algorithm for matching regular expressions with lookaround assertions.

encountered. The overall effect is a decomposition that goes deeper than what the definition of $\mathsf{shallow}_i$ suggests. In order to illuminate this concept, we introduce here the concept of a "deep decomposition", which has a close correspondence to the algorithm of Fig. 3.

The deep decomposition separates all lookaround assertions, regardless of whether they are maximal or not. This decomposition does not cause an increase in size, because oracle queries are used to refer to lookaround assertions at all levels.

Definition 28 (Deep Decomposition). For every index $i \in \mathbb{N}$, we define the *deep decomposition* function $\mathsf{deep}_i : \mathsf{LReg}(\Sigma) \to \mathsf{OReg}(\Sigma) \times \mathsf{Vect}(\{\mathsf{L2R}, \mathsf{R2L}\}) \times \mathsf{Vect}(\mathsf{OReg}(\Sigma))$ as follows:

$$
\begin{aligned}
\mathsf{deep}_i(\varepsilon) &= (\varepsilon, [\,], [\,]) \text{ and } \mathsf{deep}_i(p) = (p, [\,], [\,]) \\
\mathsf{deep}_i(r_1 \circ r_2) &= (s_1 \circ s_2, D_1 \cdot D_2, R_1 \cdot R_2), \text{ where} \\
&\qquad (s_1, D_1, R_1) = \mathsf{deep}_i(r_1) \text{ and } (s_2, D_2, R_2) = \mathsf{deep}_{i+|R_1|}(r_2) \\
\mathsf{deep}_i(r^*) &= (s^*, D, R), \text{ where } (s, D, R) = \mathsf{deep}_i(r) \\
\mathsf{deep}_i((?\!>r)) &= (\mathsf{Q}^{+}_{\varepsilon}(i + |R|), D \cdot [\mathsf{R2L}], R \cdot [s]), \text{ where } (s, D, R) = \mathsf{deep}_i(r) \\
\mathsf{deep}_i((?\!\not>r)) &= (\mathsf{Q}^{-}_{\varepsilon}(i + |R|), D \cdot [\mathsf{R2L}], R \cdot [s]), \text{ where } (s, D, R) = \mathsf{deep}_i(r) \\
\mathsf{deep}_i((?\!<r)) &= (\mathsf{Q}^{+}_{\varepsilon}(i + |R|), D \cdot [\mathsf{L2R}], R \cdot [s]), \text{ where } (s, D, R) = \mathsf{deep}_i(r) \\
\mathsf{deep}_i((?\!\not<r)) &= (\mathsf{Q}^{-}_{\varepsilon}(i + |R|), D \cdot [\mathsf{L2R}], R \cdot [s]), \text{ where } (s, D, R) = \mathsf{deep}_i(r)
\end{aligned}
$$

where p is a character class and $\circ \in \{+, \cdot\}$. Finally, we define $\mathsf{deep}(r) = \mathsf{deep}_0(r)$.

Suppose that $\mathsf{deep}(r) = (s, D, R)$ with $R = [s_0, s_1, \ldots, s_{k-1}]$. Then, each o-regex s_i contains oracle queries that only refer to assertions s_j with $j < i$.

Example 29. Consider the regex r = `((?!  )(?!(?<![0-9])0)[0-9 ])+` (in PCRE notation), which is the same as $r = ((?\!\not>\textvisiblespace\textvisiblespace\Sigma^*)(?\!\not>(?\!\not<\Sigma^*[0{-}9])0\Sigma^*)[0{-}9\textvisiblespace])^+$. Its deep decomposition is $\mathsf{deep}(r) = (s, [\mathsf{R2L}, \mathsf{L2R}, \mathsf{R2L}], [s_0, s_1, s_2])$, where

$$
s_0 = \textvisiblespace\textvisiblespace\Sigma^* \qquad s_1 = \Sigma^*[0{-}9] \qquad s_2 = \mathsf{Q}^{-}_{\varepsilon}(1) \cdot 0\Sigma^* \qquad s = (\mathsf{Q}^{-}_{\varepsilon}(0) \cdot \mathsf{Q}^{-}_{\varepsilon}(2) \cdot [0{-}9\textvisiblespace])^+.
$$

The deep decomposition of a regex $r \in \mathsf{LReg}(\Sigma)$ gives us a sequence $[s_0, s_1, \ldots, s_{k-1}]$ of k o-regexes in topological order with respect to the evaluation dependencies that they have. The means that they can be evaluated in the given order. The output tapes of earlier o-regexes are used as oracle tapes for later o-regexes. This is essentially a reformulation of the algorithm `Eval` of Fig. 3. See the algorithm `EvalDeep` of Fig. 4 for the implementation of this idea. Compared to `Eval`, the version `EvalDeep` makes the order of evaluation of o-regexes more explicit.

```
// Returns a tape $t : \mathrm{Vect}(\mathbb{B})$ that contains the matching output for $r$ over $w$.
// It holds that $t[i] = 1$ iff $w, [0, i] \models r$.
Function EvalL2R($r : LReg(\Sigma)$, $w : \Sigma^*$):
    $\mathbb{N}\ n \leftarrow |w|$ // length of input string
    $\mathrm{OReg}(\Sigma) \times \mathrm{Vect}(\{\mathsf{L2R}, \mathsf{R2L}\}) \times \mathrm{Vect}(\mathrm{OReg}(\Sigma))\ (s, D, R) \leftarrow \mathrm{deep}(r)$
    $\mathbb{N}\ k \leftarrow R.\mathrm{len}()$ // number of lookaround assertions
    for $j = 0, 1, \ldots, k-1$ do // only lookbehind assertions
        assert $D[j] = \mathsf{L2R}$
    $\mathrm{Vect}(\mathbb{B})$ tape $\leftarrow [0;\ n+1]$ // output vector of size $n+1$, initialized to 0
    // Calculate oracle valuation for position 0.
    $\mathrm{Vect}(\mathbb{B})\ \beta \leftarrow [0;\ k]$ // oracle valuation initialized to 0
    for $j = 0, 1, \ldots, k-1$ do // process lookaround assertions in topological order
        $\mathrm{ONFA}(\Sigma)\ \mathcal{A}_j \leftarrow$ Oracle-NFA for the o-regex $R[j]$
        $S_j \leftarrow \mathtt{Initial}(\mathcal{A}_j, \beta)$ // initial powerstate
        $\beta[j] \leftarrow \mathbf{ite}(S_j \cap F_j \neq \emptyset, 1, 0)$ // $F_j$ is the set of final states of $\mathcal{A}_j$
    $\mathrm{ONFA}(\Sigma)\ \mathcal{A} \leftarrow$ Oracle-NFA for the top-level o-regex $s$
    $S \leftarrow \mathtt{Initial}(\mathcal{A}, \beta)$ // initial powerstate
    tape$[0] \leftarrow \mathbf{ite}(S \cap F \neq \emptyset, 1, 0)$ // $F$ is the set of final states of $\mathcal{A}$
    for $i = 0, 1, \ldots, n-2, n-1$ do // process the input text
        for $j = 0, 1, \ldots, k-1$ do // process lookaround assertions in topological order
            $S_j \leftarrow \mathtt{Next}(\mathcal{A}_j, w[i], \beta)$
            $\beta[j] \leftarrow \mathbf{ite}(S_j \cap F_j \neq \emptyset, 1, 0)$
        $S \leftarrow \mathtt{Next}(\mathcal{A}, S, w[i], \beta)$
        tape$[i] \leftarrow \mathbf{ite}(S \cap F \neq \emptyset, 1, 0)$
    return tape
```

Fig. 5. Algorithm for matching regular expressions with lookbehind-only assertions. A completely symmetric algorithm handles regular expresssions with lookahead-only assertions.

The advantage of the formulation of `EvalDeep` is that we can easily redefine deep in order to avoid the duplication of lookaround assertions. As an example, consider the regex

$$r = \texttt{(?=a(?<=c))(?=b(?<=c))} = (?{>}\, a(?{<}\, \Sigma^* c)\Sigma^*) \cdot (?{>}\, b(?{<}\, \Sigma^* c)\Sigma^*).$$

The algorithm of Fig. 3 computes `(?<=c)` twice. We can avoid this duplication of work in `EvalDeep` by modifying the deep decomposition to only create a new o-regex when it encounters a new lookaround assertion. For the example r above, we would then have $(s, D, R) = \mathrm{deep}(r)$, where $s = \mathrm{Q}^{+}_{\varepsilon}(1) \cdot \mathrm{Q}^{+}_{\varepsilon}(2)$, $D = [\mathsf{L2R}, \mathsf{R2L}, \mathsf{R2L}]$, and $R = [\Sigma^* c, a \cdot \mathrm{Q}^{+}_{\varepsilon}(0) \cdot \Sigma^*, b \cdot \mathrm{Q}^{+}_{\varepsilon}(0) \cdot \Sigma^*]$. We call this optimization *common assertion elimination* (similar to the common subexpression elimination used in compiler optimization).

5.2 Improving the Memory Footprint

An important memory-saving optimization is enabled when all assertions are lookaheads or all of them are lookbehinds. When this holds, we say that the regular expression is *unidirectional.* In this case, we see in Fig. 3 and Fig. 4 that all ONFA simulations are performed in the same direction. For this reason, we do not need to store oracle tapes with intermediate outputs. Instead, we can pipe the output from an ONFA to be used by other ONFAs that depend on it.

This idea is implemented in the function `EvalL2R` of Fig. 5 for the case where all lookaround assertions are lookbehinds. The case where all assertions are lookaheads is completely symmetric. For every o-regex s_j of the deep decomposition, the algorithm simulates the corresponding ONFA $\mathcal{A}_j$. At every step, the ONFAs are processed in topological order in order to ensure that each ONFA

```
// Returns a tape $t$ : Vect($\mathbb{T}$) that contains the matching output for $r$ over $w$.
// If $t[i] = 1$ then $w, [0, i] \models r$.
// If $t[i] = 0$ then $w, [0, i] \not\models r$.
Function PreEval($r$ : $LReg(\Sigma)$, $w : \Sigma^*$):
    $\mathbb{N}$ $n \leftarrow |w|$ // length of input string
    OReg($\Sigma$) $s \leftarrow$ oproj($r$) // top-level o-regex ("oracle-projection")
    ONFA($\Sigma$) $\mathcal{A} \leftarrow$ oracle-NFA for the top-level o-regex $s$
    NFA($\Sigma$) $\mathcal{A}^\top \leftarrow$ Replace oracle-guarded transitions in $\mathcal{A}$ by $\varepsilon$-transitions
    NFA($\Sigma$) $\mathcal{A}^\perp \leftarrow$ Remove oracle-guarded transitions of $\mathcal{A}$
    // $\mathcal{A}^\top$ (resp., $\mathcal{A}^\perp$) is an over-approximation (resp., under-approximation) of $\mathcal{A}$
    Vect($\mathbb{T}$) tape $\leftarrow [0; n+1]$ // output vector of size $n+1$, initialized to 0
    $S^\top \leftarrow$ Initial($\mathcal{A}^\top$)   // initial powerstate
    $S^\perp \leftarrow$ Initial($\mathcal{A}^\perp$)   // initial powerstate
    if $S^\perp \cap F \neq \emptyset$ then tape[$i$] $\leftarrow 1$
    else if $S^\top \cap F \neq \emptyset$ then tape[$i$] $\leftarrow$ ?
    else tape[$i$] $\leftarrow 0$
    for $i = 0, 1, \ldots, n-1$ do // process the input text
        $S^\top \leftarrow$ Next($\mathcal{A}^\top, S^\top, w[i]$)
        $S^\perp \leftarrow$ Next($\mathcal{A}^\top, S^\perp, w[i]$)
        if $S^\perp \cap F \neq \emptyset$ then tape[$i$] $\leftarrow 1$
        else if $S^\top \cap F \neq \emptyset$ then tape[$i$] $\leftarrow$ ?
        else tape[$i$] $\leftarrow 0$
    return tape
```

Fig. 6. An approximate algorithm for matching regular expressions with lookaround assertions. If this algorithm indicates that the output is uncertain, then one of the previous algorithms has to be used.

has the oracle valuation that it needs. The ONFA $\mathcal{A}$ for the top-level o-regex s is always processed last, as it may need the output values from all other ONFAs.

The intuition for the algorithm of Fig. 5, when compared to the algorithm of Fig. 4, is that the evaluation of the output matrix proceeds column-by-column instead of row-by-row. Since only the most recent column of the matrix is needed for the next steps, we do not need to store the entire matrix. So, we store only the last column and we update it at every step. This reduces the memory footprint from $O(m \cdot n)$ to $O(m)$.

5.3 Approximation for Saving Work

We also consider an optimization where the computation of lookaround assertions can be avoided altogether when they are not necessary for producing the output. For example, consider a regex of the form $r = \Sigma^* \cdot abcd \cdot r_1$, where r contains several lookaround assertions. If the input text contains no occurrence of the string $abcd$, then it cannot contain any match for r. In this case, we do not have to compute any of the lookaround assertions, because their values are not needed at all.

This idea is made more precise in the algorithm PreEval of Fig. 6. Given a regular expression $r \in \mathrm{LReg}(\Sigma)$, we first compute its oracle-projection $s \in \mathrm{OReg}(\Sigma, V_k)$ where $k = \mathrm{oarity}(r)$. We will attempt to compute the output without knowing the truth values of the oracle queries. In order to do this, we will approximate the ONFA $\mathcal{A}$ for s using two NFAs. The NFA $\mathcal{A}^\top$ is obtained from $\mathcal{A}$ by replacing each oracle-guarded transition of the form $q \to^{Q^\sigma_\varepsilon(v)} q'$ by an ε-transition $q \to^\varepsilon q'$. So, it over-approximates $\mathcal{A}$, that is, $[\![\mathcal{A}]\!] \subseteq [\![\mathcal{A}^\top]\!]$. The NFA $\mathcal{A}^\perp$ is derived from $\mathcal{A}$ by removing all oracle-guarded transitions. So, it under-approximates $\mathcal{A}$, that is, $[\![\mathcal{A}^\perp]\!] \subseteq [\![\mathcal{A}]\!]$. We examine cases:

(1) $\mathcal{A}^\perp$ accepts: It must also be the case that $\mathcal{A}$ and $\mathcal{A}^\top$ accept.

(2) $\mathcal{A}^\perp$ rejects and $\mathcal{A}^\top$ accepts: We do not know whether $\mathcal{A}$ accepts or rejects.
(3) $\mathcal{A}^\top$ rejects: It must also be the case that $\mathcal{A}$ and $\mathcal{A}^\perp$ reject.

In cases (1) and (3) the matching output is certain to be correct. Case (2) gives rise to uncertainty regarding the output. If case (2) never occurs during the evaluation, then we do not need to evaluate lookaround assertions at all. If case (2) occurs, then we have to use the previous algorithm in order to ensure the correctness of the output.

Example 30 (Over- and under-approximation). Consider the regular expression $r = \Sigma^* \cdot (r_1 + r_2)$, where $r_1 =$ `bbb(?=a)((?!aaa).)+(?<=a)bbb` and $r_2 =$ `bbbbb`. Using the expression r, we search for the occurrence of substrings that match r_1 or r_2. Intuitively, the over-approximation replaces r_1 by `bbb.+bbb`, and the under-approximation replaces r_1 by $\emptyset$. The table below shows the output values computed by the algorithm `PreEval` of Fig. 6 on the input `aabbbaabaaabbbbbaa`.

text	a	a	b	b	b	a	a	b	a	a	a	b	b	b	b	b	a	a
$\mathcal{A}^\top$	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	1	1	1
$\mathcal{A}^\perp$	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	1	1	1

The bold output values indicate that the output of `PreEval` is uncertain.

6 EXPERIMENTAL EVALUATION

We have implemented the pattern matching algorithms of Sections 3, 4 and 5 using the Rust programming language. We perform an experimental evaluation in order to answer the following two research questions:

(1) Do the optimizations of Section 5 provide a significant performance benefit?
(2) Does our implementation have competitive performance against state-of-the-art regex engines that support lookaround assertions?

There are many implementations of matching regexes with lookaround that are based on backtracking search instead of using automata-based algorithms. It is known that backtracking suffers from superlinear complexity (in the size of the input text). In fact, for some regexes the time complexity is exponential. This behavior is well-known and is often called "catastrophic backtracking". We will compare against the PCRE (Perl Compatible Regular Expressions) library (more specifically, the PCRE2 library) [The PCRE2 Developers 2023], which is implemented in C, and Java's regex library. We have chosen to compare with PCRE2 and Java's regex library because they are widely used regex engines that support lookaround assertions. Note that PCRE2 is implemented in a low-level programming language (and thus avoids the overheads of virtual machines such as the JVM).

Benchmarking Datasets. We have performed experiments using two widely used datasets of regular expressions that use PCRE syntax: (1) the *Snort* dataset [Snort 2023] and (2) the *Suricata* dataset [Suricata 2023]. Both these datasets specify signatures for network traffic that may indicate malicious network activity. We use real network traffic (obtained from https://archive.wrccdc.org/pcaps/) as input strings for the experimental evaluation.

The average length of the regexes in Snort (resp., Suricata) is 109 (resp., 102). This refers to the length of the textual representation using the PCRE syntax of the datasets.

Roughly 5.5% of regexes in Snort (7.6% in Suricata) contain lookaround. Out of the regexes with lookaround in Snort, 97% contain lookahead and 10% contain lookbehind, which means that 7% contain both lookahead and lookbehind. For Suricata, the corresponding percentages are 98% for lookahead, 8% for lookbehind, and 6% for both. The average number of lookaround assertions (resp., lookahead assertions and lookbehind assertions) in a regex that contains lookaround in Snort is 1.96 (resp., 1.85 and 0.11). For Suricata, the corresponding average numbers are 1.84, 1.74 and 0.10 respectively (for lookaround, lookahead and lookbehind assertions). Out of the regexes with

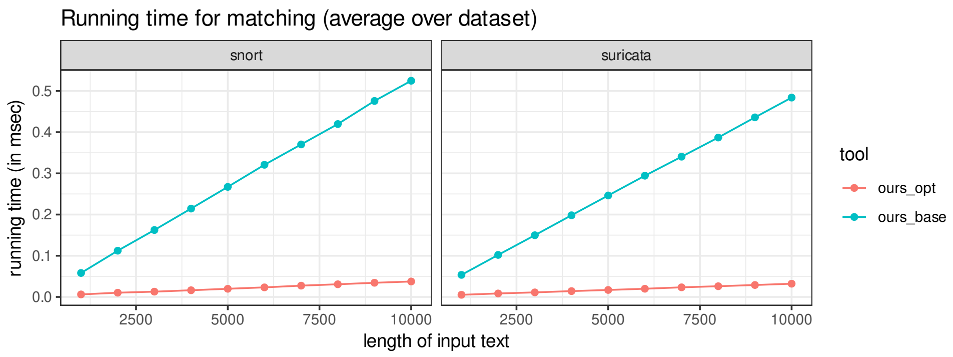


Fig. 7. Comparison between base algorithm and optimized algorithm.

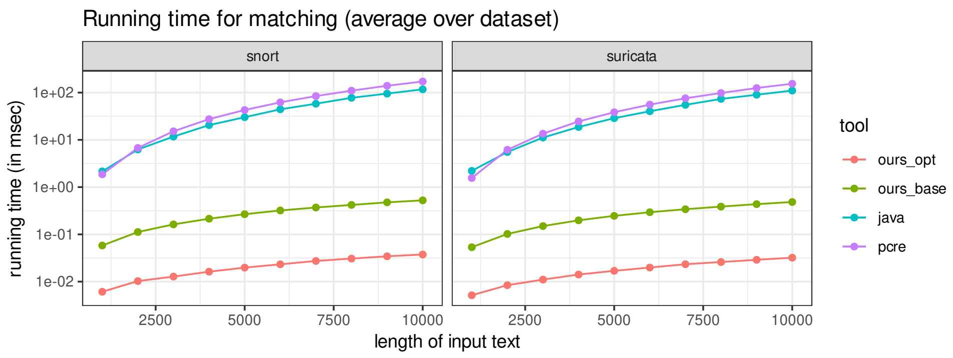


Fig. 8. Comparison of our algorithms with PCRE and Java's regex engine.

lookaround in Snort and Suricata, 96% and 97% respectively have lookaround depth 1 (which means that 4% and 3% respectively have nested lookaround assertions).

Effect of performance optimizations. Fig. 7 shows the performance of the basic version of our matching algorithm (called ours_base in the figure). This is the implementation of the algorithm of Fig. 3. The version that is called ours_opt in the figure also incorporates the optimizations described in Section 5: avoiding work duplication due to multiple occurrences of the same lookaround assertion, one-pass matching when the lookaround assertions are unidirectional, and the use of approximation to avoid the computation of some oracle truth values.

Fig. 7 contains two plots, one for each regex dataset, namely Snort and Suricata. The horizontal axis of each plot shows the length of the input string. The vertical axis shows the average running time of the regex matching algorithm in milliseconds. The average is taken over the entire dataset of regexes. Each point is annotated with error bars that show the standard deviation of the running time (the errors bars are too small to see). A crucial observation is that the running time of both versions of our algorithm is ***linear*** in the length of the input string. This behavior is consistent with the time complexity analysis of Theorem 27. The other observation is that the optimized version of our algorithm (ours_opt) is substantially faster than the basic version (ours_base). More specifically, the optimizations result in a ***speedup of at least*** **10×** across all experiments of Fig. 7.

Comparison with PCRE and Java's regex engine. In Fig. 8 we include the performance of PCRE2 and Java's regex engine. The plots are similar to the ones of Fig. 7. One difference is that the vertical

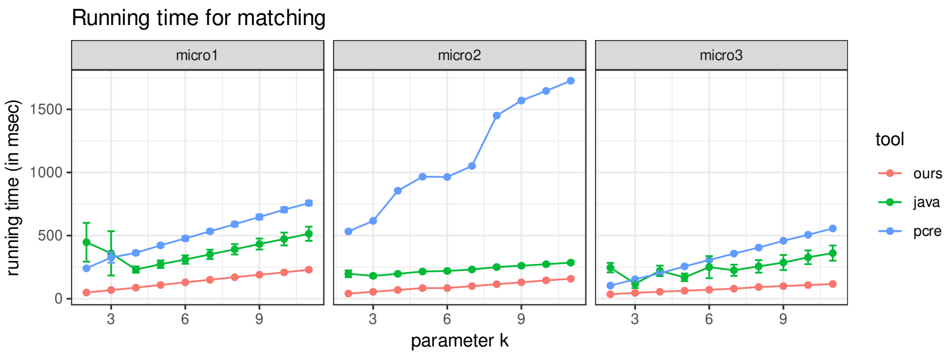


Fig. 9. Microbenchmarks: Comparison between our implementation and PCRE, Java.

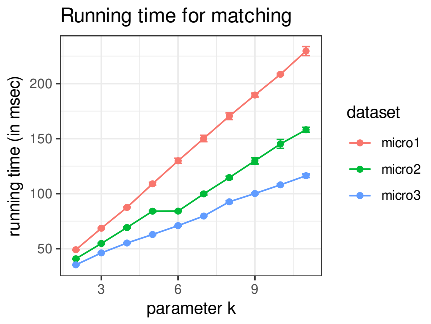
axis in the plots of Fig. 8 is log-scaled. Using logarithmic scale for the running time is necessary due to the big difference in running time between our implementation and the other tools. Our first observation is that the running time of both PCRE2 and Java is ***superlinear*** with respect to the length of the input string. This is witnessed by the widening gap between the green and red curves (ours_base and ours_opt respectively) and the blue and purple curves (java and pcre respectively) as the string length grows. The ratio between pcre and ours_opt is at least 250× for text length 1000. It grows to at least 4000× for text length 10000. Similar observations can be made for Java. So, our regex engine is ***several orders of magnitude faster*** than PCRE and Java across all experiments of Fig. 8.

Microbenchmarks. We also consider microbenchmarks that focus on cases that do not trigger super-linear behavior for backtracking engines (PCRE and Java). First, we consider the family $(\rho_k)_{k\geq 2}$ of regexes of the form $\rho_k = r(?{=}\,r_1)(?{=}\,r_2)\cdots(?{=}\,r_k)$, where $r, r_1, r_2, \ldots, r_k$ are lookaround-free signatures. We also consider the family $(\rho'_k)_{k\geq 2}$ of regexes of the form

$$\rho'_k = r'(?{=}\,(.\{2\})^+ r_\#)(?{=}\,(.\{3\})^+ r_\#)\cdots(?{=}\,(.\{k\})^+ r_\#),$$

where $r_\#$ is a signature that has the role of an "end-of-block" marker. The regex family $(\rho'_k)_k$ is inspired from the regex family T_n in section 3.6 of [Miyazaki and Minamide 2019]. The regexes $(T_n)_n$ witness the doubly exponential lower bound for DFAs that encode regexes with lookahead. Finally, we define the regex family $(\rho''_k)_{k\geq 2}$ by $\rho''_k = r''((?{=}\,r''_1) + (?{=}\,r''_2) + \cdots + (?{=}\,r''_k))$, where $r'', r''_1, r''_2, \ldots, r''_k$ are lookaround-free signatures.

The regex families ρ_k and ρ'_k use lookahead assertions in a way that encodes a form of *intersection*. They would pose a substantial challenge on algorithms that construct a single automaton through a product construction, as this would cause an exponential blowup in size. The regex family ρ''_k involves a nondeterministic choice over lookahead assertions. The regex families $\rho_k, \rho'_k, \rho''_k$ correspond to the microbenchmarks called micro1, micro2 and micro3 respectively. Fig. 9 shows experimental results for the performance of our implementation, PCRE and Java's regex engine over these 3 microbenchmarks. The experiments use input text of length 10^6. The horizontal axis corresponds to the parameter k. The vertical axis shows the matching running time in milliseconds. All regex engines seem to have running



time that is linear in parameter k. Note that the running time of our implementation does not blow up because we do not construct large automata, as explained earlier in Section 4. The plot shown above (on the right) focuses on the performance of our implementation over all 3 microbenchmarks. Observe that the running time of our implementation is linear in k.

Experimental Setup & Measurement Methodology. The experiments were executed in Ubuntu 20.04 on a desktop computer equipped with an Intel Xeon W-2295 CPU (18 cores) and 64 GB of RAM. We used version 1.71.0 of the Rust compiler. The PCRE2 library was installed using the `libpcre2-dev` package through usual repositories. At the time of executing the experiments, the `10.39-3ubuntu0.1` version of the package was used.

Each measurement of running time (for a matching algorithm that is given a regex and a string as input) is taken as the average of 10 trials. The uncertainty in the measurement is quantified using the standard deviation of the 10 trials.

7 RELATED WORK

Constructs similar to (positive and negative) lookahead assertions have been popular in the construction of parsers. Specifying a lookahead assertion in a parser can be used to reduce ambiguity (and thus limit backtracking, for backtracking implementations). For instance, parsers for context-free languages are often classified by the number of tokens the parser may need to peek ahead. We also see the use of lookahead in [Sakuma et al. 2012] where it is used to transform nondeterministic transducers into deterministic ones. Regular lookahead is used in the language *Bex* [Veanes 2015], which is used for specifying string transformations. The so-called "And-predicates" and "Not-predicates" in parsing expression grammars (PEGs) [Ford 2004] correspond to positive and negative lookahead assertions respectively. Miyazaki and Minamide [2021] have proposed extensions of context-free grammars with lookahead.

Lookaround assertions are often used to extract data that arise in specific contexts. The language CDuce [Benzaken et al. 2003] uses regular expression types to extract data from XML documents. The Kleenex language [Grathwohl et al. 2016] uses regular expressions as grammars (types) to describe string transductions that can extract data from streams. This involves a "greedy" disambiguation policy that generalizes greedy regex parsing [Frisch and Cardelli 2004; Grathwohl et al. 2013, 2014a; Nielsen and Henglein 2011].

The use of derivatives for matching regular languages is popular in functional and formally verified implementations. The simplest form are Brzozowski's derivatives [Brzozowski 1964] and they lend themselves to a natural functional implementation of an implicit DFA of the underlying regular expression. Coquand and Siles [2011] present a formally verified framework for deciding equivalence of regular expressions based on Brzozowski derivatives. Brzozowoski has shown that the number of derivatives are finite if they are simplified using associativity, idempotence and commutativity rules. Recent work [Egolf et al. 2022] shows how these optimizations could be incorporated in practice into a verified implementation. The size of a Brzozowoski derivative can be large. Antimirov [1996] suggested using sets of partial derivatives for a more efficient algorithm. This is also related to the technique of *prebases* discussed by Mirkin [1966] (see also [Brzozowski 1971] and [Champarnaud and Ziadi 2001]). Partial derivatives have been used for formally verified implementations in [Komendantsky 2012] and [Moreira et al. 2012].

Doczkal et al. [2013] have developed a comprehensive formalization of regular languages in Coq which encompasses regular expressions, NFAs, DFAs, and the Myhill-Nerode Theorem. We see another NFA based formalization in [Firsov and Uustalu 2013], where NFAs are simulated using their matrix representations in the Agda formalization.

Morihata [2012] studies the translation of regular expressions with lookahead into DFAs of doubly exponential size. A treatment of lookahead using derivatives can be found in [Miyazaki and Minamide 2019]. A regular expression with lookahead is interpreted as a set of pairs (s, t) of strings, where s is the matching string and t is the remaining string. A lookahead assertion is interpreted as a set of pairs of the form (ε, t) because it constrains the remaining string without consuming any string symbols. The finite state automata constructed using this derivative-based technique has a similar blow-up to the one considered by Morihata [2012]. The authors provide a lower bound argument showing that lookaround assertions can indeed cause a doubly exponential blow-up in some cases when converted to a DFA. Note that, while our algorithm runs in linear time, it is not a streaming algorithm, since it makes both forward and backward passes (in the case of lookbehind and lookahead assertions, respectively) on the input. Berglund et al. [2021] establish the semantics of lookarounds using alternating automata that can make forward or backward passes on the string. This definition is very close to the operational definition used by practitioners. However, alternating automata are a powerful model, and it is not easy to see how they can be simulated efficiently. [Trofimovich 2020] suggests an implementation of regular expression matching (tool RE2C) using automata and tagged transitions and lookahead. The tags are markers which help extract sub-matches. Moseley et al. [2023] consider a derivative-based approach for matching regular expressions with *anchors*, which are a very restricted form of lookaround assertions that only have a lookahead or lookbehind of at most one symbol.

Bando et al. [2012] consider regular expressions with lookahead and lookbehind in the context of deep packet inspection in networks. They propose an FPGA-based implementation and estimate that around 25,000 regexes can be accommodated and a throughput of 34 Gbps can be achieved.

Chida and Terauchi [2022] consider the expressiveness of regular expressions with lookaround and backreferences. They conclude that adding lookaround enhances the expressiveness of regular expressions with backreferences. This is in contrast to classical regular expressions (i.e., without backreferences), where adding lookaround assertions does not increase expressiveness.

8 CONCLUSION AND FUTURE WORK

We have proposed a formal semantics for regular expressions with lookaround. Many commonly used regex engines that support lookaround resort to backtracking search. Algorithms that are based on using one automaton for the entire pattern also seem to incur a non-trivial blow-up. Intuitively, this is because matching lookaround information requires additional contextual information about the remainder of the string. We have presented an algorithm that matches regexes with lookaround in time $O(m \cdot n)$, where m is the size of regex and n is the length of the input string. This time complexity is the same as that of Thompson's algorithm for classical (i.e., lookaround-free) regular expression. We see from our empirical evaluation that the implementation of our algorithm, which is augmented with some performance optimizations, has performance that is substantially better than the state-of-the-art PCRE and Java engines on the real workloads that we have considered.

A worthwhile direction for future work is the extension of our implementation with more advanced operators that are useful in practice. The incorporation of some optimizations for bounded repetition (see, for example, [Kong et al. 2022] and [Le Glaunec et al. 2023]) in our implementation seems to be feasible. Backreferences pose a challenge because they can give rise to non-regularity, but there are special cases (e.g., backreferences to bounded strings, as in the regex `(?P<q>[a-z]{3})(?P=q)`) that stay within the realm of regularity.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. This research was supported in part by the US National Science Foundation award CCF 2008096.

REFERENCES

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (1975), 333–340. https://doi.org/10.1145/360825.360855

Valentin Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theoretical Computer Science* 155, 2 (1996), 291–319. https://doi.org/10.1016/0304-3975(95)00182-4

Masanori Bando, N. Sertac Artan, and H. Jonathan Chao. 2012. Scalable Lookahead Regular Expression Detection System for Deep Packet Inspection. *IEEE/ACM Transactions on Networking* 20, 3 (2012), 699–714. https://doi.org/10.1109/TNET.2011.2181411

Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. 2018. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). LNCS, Vol. 10457. Springer, Cham, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, New York, NY, USA, 51–63. https://doi.org/10.1145/944705.944711

Martin Berglund, Frank Drewes, and Brink van der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. In *Automata and Formal Languages 2014 (AFL 2014) (Electronic Proceedings in Theoretical Computer Science (EPTCS), Vol. 151)*, Zoltán Ésik and Zoltán Fülöp (Eds.). Open Publishing Association, 109–123. https://doi.org/10.4204/eptcs.151.7

Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. 2021. Regular Expressions with Lookahead. *JUCS - Journal of Universal Computer Science* 27, 4 (2021), 324–340. https://doi.org/10.3897/jucs.66330

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

Janusz A. Brzozowski. 1971. Review of An Algorithm for Constructing a Base in a Language of Regular Expressions, by B. G. Mirkin. *The Journal of Symbolic Logic* 36, 4 (1971), 694–694. https://doi.org/10.2307/2272532

Jean-Marc Champarnaud and Djelloul Ziadi. 2001. From Mirkin's Prebases to Antimirov's Word Partial Derivatives. *Fundamenta Informaticae* 45, 3 (2001), 195–205. https://ip.ios.semcs.net/articles/fundamenta-informaticae/fi45-3-03

Agnishom Chattopadhyay and Konstantinos Mamouras. 2020. A Verified Online Monitor for Metric Temporal Logic with Quantitative Semantics. In *RV 2020 (LNCS, Vol. 12399)*, Jyotirmoy Deshmukh and Dejan Ničković (Eds.). Springer, Cham, 383–403. https://doi.org/10.1007/978-3-030-60508-7_21

Nariyoshi Chida and Tachio Terauchi. 2022. On Lookaheads in Regular Expressions with Backreferences. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:18. https://doi.org/10.4230/LIPIcs.FSCD.2022.15

Thierry Coquand and Vincent Siles. 2011. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *CPP 2011 (LNCS, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, Berlin, Heidelberg, 119–134. https://doi.org/10.1007/978-3-642-25379-9_11

Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html. [Online; accessed November 14, 2023].

Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. 2013. A Constructive Theory of Regular Languages in Coq. In *CPP 2013 (LNCS, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, Cham, 82–97. https://doi.org/10.1007/978-3-319-03545-1_6

Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. 2014. On-Line Monitoring for Temporal Logic Robustness. In *RV 2014 (LNCS, Vol. 8734)*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Springer, Cham, 231–246. https://doi.org/10.1007/978-3-319-11164-3_19

Derek Egolf, Sam Lasser, and Kathleen Fisher. 2022. Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022)*. ACM, New York, NY, USA, 27–39. https://doi.org/10.1145/3497775.3503694

Denis Firsov and Tarmo Uustalu. 2013. Certified Parsing of Regular Languages. In *CPP 2013 (LNCS, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, Cham, 98–113. https://doi.org/10.1007/978-3-319-03545-1_7

Bryan Ford. 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)*. ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *ICALP 2004 (LNCS, Vol. 3142)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, Berlin, Heidelberg, 618–629. https://doi.org/10.1007/978-3-540-27836-8_53

Bjørn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm, and Sebastian Paaske Tørholm. 2016. Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 284–297. https://doi.org/10.1145/2837614.2837647

Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen, and Ulrik Terp Rasmussen. 2013. Two-Pass Greedy Regular Expression Parsing. In *CIAA 2013 (LNCS, Vol. 7982)*, Stavros Konstantinidis (Ed.). Springer, Berlin, Heidelberg, 60–71. https://doi.org/10.1007/978-3-642-39274-0_7

Niels Bjørn Bugge Grathwohl, Fritz Henglein, and Ulrik Terp Rasmussen. 2014a. Optimally Streaming Greedy Regular Expression Parsing. In *Theoretical Aspects of Computing – ICTAC 2014 (LNCS, Vol. 8687)*, Gabriel Ciobanu and Dominique Méry (Eds.). Springer, Cham, 224–240. https://doi.org/10.1007/978-3-319-10882-7_14

Niels Bjørn Bugge Grathwohl, Dexter Kozen, and Konstantinos Mamouras. 2014b. KAT + B!. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14)*. ACM, New York, NY, USA, Article 44, 44:1–44:10 pages. https://doi.org/10.1145/2603088.2603095

grep 2023. GREP - Global Regular Expression Print. https://www.gnu.org/software/grep/.

Hyperscan 2023. Intel's Hyperscan: A high-performance multiple regex matching library. https://github.com/intel/hyperscan.

Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. 1968. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. *Commun. ACM* 11, 12 (1968), 805–813. https://doi.org/10.1145/364175.364185

Stephen Cole Kleene. 1956. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*, Claude E. Shannon and John McCarthy (Eds.). Number 34 in Annals of Mathematics Studies. Princeton University Press, 3–41.

Vladimir Komendantsky. 2012. Reflexive Toolbox for Regular Expression Matching: Verification of Functional Programs in Coq+Ssreflect. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (PLPV '12)*. ACM, New York, NY, USA, 61–70. https://doi.org/10.1145/2103776.2103784

Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-Hardware Codesign for Efficient In-Memory Regular Pattern Matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. ACM, New York, NY, USA, 733–748. https://doi.org/10.1145/3519939.3523456

Dexter Kozen. 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. *Information and Computation* 110, 2 (1994), 366–390. https://doi.org/10.1006/inco.1994.1037

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Transactions on Programming Languages and Systems* 19, 3 (1997), 427–443. https://doi.org/10.1145/256167.256195

Dexter Kozen and Konstantinos Mamouras. 2014. Kleene Algebra with Equations. In *ICALP 2014 (LNCS, Vol. 8573)*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer, Berlin, Heidelberg, 280–292. https://doi.org/10.1007/978-3-662-43951-7_24

Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching Using Bit Vector Automata. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 92 (2023), 30 pages. https://doi.org/10.1145/3586044

Oded Maler, Dejan Nickovic, and Amir Pnueli. 2008. Checking Temporal Properties of Discrete, Timed and Continuous Behaviors. In *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich (Eds.). LNCS, Vol. 4800. Springer, Berlin, Heidelberg, 475–505. https://doi.org/10.1007/978-3-540-78127-1_26

Konstantinos Mamouras. 2015. *Extensions of Kleene Algebra for Program Verification*. Ph. D. Dissertation. Cornell University, Ithaca, NY. http://hdl.handle.net/1813/40960

Konstantinos Mamouras. 2017. Equational Theories of Abnormal Termination Based on Kleene Algebra. In *FoSSaCS 2017 (LNCS, Vol. 10203)*, Javier Esparza and Andrzej S. Murawski (Eds.). Springer, Berlin, Heidelberg, 88–105. https://doi.org/10.1007/978-3-662-54458-7_6

Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. 2021a. Algebraic Quantitative Semantics for Efficient Online Temporal Monitoring. In *TACAS 2021 (LNCS, Vol. 12651)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, Cham, 330–348. https://doi.org/10.1007/978-3-030-72016-2_18

Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. 2021b. A Compositional Framework for Quantitative Online Monitoring over Continuous-Time Signals. In *RV 2021 (LNCS, Vol. 12974)*, Lu Feng and Dana Fisman (Eds.). Springer, Cham, 142–163. https://doi.org/10.1007/978-3-030-88494-9_8

Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. 2023. A Compositional Framework for Algebraic Quantitative Online Monitoring over Continuous-Time Signals. *International Journal on Software Tools for Technology Transfer* 25, 4 (2023), 557–573. https://doi.org/10.1007/s10009-023-00719-w

Konstantinos Mamouras and Zhifu Wang. 2020. Online Signal Monitoring with Bounded Lag. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3868–3880. https://doi.org/10.1109/TCAD.2020.3013053

B. G. Mirkin. 1966. An Algorithm for Constructing a Base in a Language of Regular Expression. *Engineering Cybernetics* 5 (1966), 110–116.

Takayuki Miyazaki and Yasuhiko Minamide. 2019. Derivatives of Regular Expressions with Lookahead. *Journal of Information Processing* 27 (2019), 422–430. https://doi.org/10.2197/ipsjjip.27.422

Takayuki Miyazaki and Yasuhiko Minamide. 2021. Context-Free Grammars with Lookahead. In *LATA 2021 (LNCS, Vol. 12638)*, Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron (Eds.). Springer, Cham, 213–225. https://doi.org/10.1007/978-3-030-68195-1_16

Nelma Moreira, David Pereira, and Simão Melo de Sousa. 2012. Deciding Regular Expressions (In-)Equivalence in Coq. In *RAMiCS 2012 (LNCS, Vol. 7560)*, Wolfram Kahl and Timothy G. Griffin (Eds.). Springer, Berlin, Heidelberg, 98–113. https://doi.org/10.1007/978-3-642-33314-9_7

Akimasa Morihata. 2012. Translation of Regular Expression with Lookahead into Finite State Automaton. *Computer Software* 29, 1 (2012), 147–158. https://doi.org/10.11309/jssst.29.1_147

Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 148 (2023), 24 pages. https://doi.org/10.1145/3591262

Lasse Nielsen and Fritz Henglein. 2011. Bit-coded Regular Expression Parsing. In *LATA 2011 (LNCS, Vol. 6638)*, Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide (Eds.). Springer, Berlin, Heidelberg, 402–413. https://doi.org/10.1007/978-3-642-21254-3_32

Michael O. Rabin and Dana Scott. 1959. Finite Automata and their Decision Problems. *IBM Journal of Research and Development* 3, 2 (1959), 114–125. https://doi.org/10.1147/rd.32.0114

RE2 2023. RE2: Google's regular expression library. https://github.com/google/re2.

Indranil Roy and Srinivas Aluru. 2016. Discovering Motifs in Biological Sequences Using the Micron Automata Processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 13, 1 (2016), 99–111. https://doi.org/10.1109/TCBB.2015.2430313

Yuto Sakuma, Yasuhiko Minamide, and Andrei Voronkov. 2012. Translating Regular Expression Matching into Transducers. *Journal of Applied Logic* 10, 1 (2012), 32–51. https://doi.org/10.1016/j.jal.2011.11.003 Special issue on Automated Specification and Verification of Web Systems.

Snort 2023. Snort Intrusion Detection System. https://www.snort.org/.

Suricata 2023. Suricata Threat Detection Engine. https://suricata.io/.

Prasanna Thati and Grigore Roşu. 2005. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electronic Notes in Theoretical Computer Science* 113 (2005), 145–162. https://doi.org/10.1016/j.entcs.2004.01.029 Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

The PCRE2 Developers. 2023. Perl-compatible Regular Expressions (revised API: PCRE2). https://pcre2project.github.io/pcre2/doc/html/index.html.

Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

Ulya Trofimovich. 2020. RE2C: A Lexer Generator Based on Lookahead-TDFA. *Software Impacts* 6 (2020), 100027. https://doi.org/10.1016/j.simpa.2020.100027

Margus Veanes. 2015. Symbolic String Transformations with Regular Lookahead and Rollback. In *PSI 2014 (LNCS, Vol. 8974)*, Andrei Voronkov and Irina Virbitskaite (Eds.). Springer, Berlin, Heidelberg, 335–350. https://doi.org/10.1007/978-3-662-46823-4_27

Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 2006. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '06)*. ACM, New York, NY, USA, 93–102. https://doi.org/10.1145/1185347.1185360

Received 2023-07-11; accepted 2023-11-07