Verified and Efficient Matching of Regular Expressions with Lookaround

Agnishom Chattopadhyay
Rice University
Houston, USA
agnishom@rice.edu

Angela W. Li
Rice University
Houston, USA
awl@rice.edu

Konstantinos Mamouras
Rice University
Houston, USA
mamouras@rice.edu

Abstract

Regular expressions can be extended with lookarounds for contextual matching. This paper discusses a Coq formalization of the theory of regular expressions with lookarounds. We provide an efficient and purely functional algorithm for matching expressions with lookarounds and verify its correctness. The algorithm runs in time linear in both the size of the regular expression as well as the input string. Our experimental results provide empirical support to our complexity analysis. To the best of our knowledge, this is the first formalization of a linear-time matching algorithm for regular expressions with lookarounds.

CCS Concepts: • **Theory of computation → Regular languages**; *Automata extensions*; *Pattern matching*; • **Software and its engineering → Software verification**.

Keywords: lookbehind, lookahead, NFA, regex

ACM Reference Format:
Agnishom Chattopadhyay, Angela W. Li, and Konstantinos Mamouras. 2025. Verified and Efficient Matching of Regular Expressions with Lookaround. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25), January 20–21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3703595.3705884

1 Introduction

While textbook regular expressions involve only concatenation $(r \cdot s)$, alternation $(r + s)$ and Kleene iteration r^*, modern regex engines (such as PCRE [43]) support a multitude of additional features. Some of these features are purely syntactic sugar (such as the use of character classes or the use of the ? operator), some of these features (like counting) add exponential succinctness and some features (like backreferences) make it possible to match certain non-regular languages. Certain constructs, such as capture groups or lookarounds, do not add to the expressive power in terms of the overall strings that can be matched, but are best described using a different semantics (i.e., different from the simple Boolean semantics of $w \in [\![r]\!]$ and $w \notin [\![r]\!]$).

The focus of this paper is on expressions with lookarounds. Lookahead assertions are used to assert that a certain pattern is satisfied in the future of the current position in the string. For example, the expression `\d+\.\d\d(?=USD)` could be used to match a price of some item formatted as `xxx.yy USD`. Since the pattern `USD` appears within a lookahead, this part of the string is not returned, so that the programmer can directly work with the numerical information. Similarly, if the pattern `USD` was expected to appear before the price, one could use the expression `(?<=USD)\d+\.\d\d`, where the pattern `(?<=USD)` is called a lookbehind.

The majority of popular regex engines bundled with the standard library of programming languages such as Java, Javascript and Python are based on backtracking, which may take exponential time in the worst-case scenario (called *catastrophic backtracking* [6]). This behavior could be used to conduct a denial-of-service attack [13]. Automata-based tools such as Google's RE2 [40] and Intel's Hyperscan [23] are guaranteed to finish execution in linear time. Despite this, backtracking engines remain popular due to their flexibility and support for non-classical features, including (but not limited to) lookarounds.

The **contributions** of this paper are as follows: (1) We formally verify in Coq an algorithm for matching regular expressions with lookarounds, that runs in linear time in terms of the size of the regular expression as well as the length of the input string; (2) We present a purely functional version of the Oracle NFA based algorithm in [31]; and (3) We empirically demonstrate the linear time-complexity of our algorithm.

Approaches for the linear-time matching of expressions with lookarounds have been recently published in [31], [21] and [5]. An approach that uses memoization and backtracking is described in [14]. In this paper, we follow the terminology and approach from [31]. The **key idea** in these algorithms can be summarized as a layered dynamic programming algorithm. As input, we are given expressions with lookarounds, which allow imposing conditions on the part of the string which is "ahead" or "behind" the current

CPP '25, January 20–21, 2025, Denver, CO, USA

ACM ISBN 979-8-4007-1347-7/25/01
https://doi.org/10.1145/3703595.3705884

position. We tackle these expressions by decomposing them into multiple layers, with each level of nested lookaround in a layer. At the bottom layer, we find ordinary regular expressions. When considering the subsequent upper layers, the information relating to whether or not the expression in the layer below was matched becomes necessary. Thus, by proceeding in a layerwise fashion, the recomputation of matching information in the lookarounds can be avoided. In the intermediate stages of the algorithm, we use the notion of *Oracle Regular Expressions* to capture the concept of expressions which rely on external oracle queries to determine their satisfaction. These expressions can be matched in a manner similar to standard regular expressions (such as by simulating an NFA, as seen in [31]) but with the added complication of handling the oracle queries.

In this paper, we show how algorithms based on Marked Regular Expressions [3, 19] can be extended to match Oracle Expressions. Marked expressions allow simulating NFAs in a manner that closely resembles the syntax of regular expressions, facilitating elegant verification without compromising performance. Our algorithm is purely functional, by which we mean that the execution of the algorithm does not require destructive updates. In contrast, direct efficient implementations of NFA simulations are usually done using mutable sets containing the "active states" of the NFA. Functional Alternatives to such set-like data structures would usually incur a logarithmic overhead in the time complexity.

Outline of the Paper. In Section 2, we discuss how the formal semantics of lookarounds are encoded in Coq, and how they can be used for equational reasoning. In Section 3, we define Oracle Regular Expressions and Oracle Strings, the core abstraction used in our algorithm. The matching algorithm for oracle expressions is discussed in Section 4. The main algorithm is presented in Section 5. Performance experiments using our extracted code are presented in Section 6. Finally, we discuss related work in Section 7.

2 Lookaround Semantics

This section introduces the formal definitions used in the paper.

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

$$r, r_1, r_2 ::= \varepsilon \mid \sigma \in \mathcal{P} \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r^* \mid (?{>}\, r) \mid (?{\not>}\, r) \mid (?{<}\, r) \mid (?{\not<}\, r)$$

The expressions of the form $(?{>}\, r)$, $(?{\not>}\, r)$, $(?{<}\, r)$ and $(?{\not<}\, r)$ are called *positive lookahead*, *negative lookahead*, *positive lookbehind* and *negative lookbehind* respectively. Collectively, they are called *lookaround assertions*. Expressions of the form $(?{>}\, r)$ and $(?{\not>}\, r)$ (respectively, $(?{<}\, r)$ and $(?{\not<}\, r)$) are called *lookahead assertions* (respectively, *lookbehind assertions*). Furthermore, $(?{>}\, r)$ and $(?{<}\, r)$ are called *positive lookarounds*, while $(?{\not>}\, r)$ and $(?{\not<}\, r)$ are called *negative lookarounds*.

$$\begin{aligned}
w, [i,j] \vDash \varepsilon &\iff i = j \\
w, [i,j] \vDash \sigma &\iff j = i+1 \wedge \sigma(w_i) = 1 \\
w, [i,j] \vDash r_1 + r_2 &\iff w, [i,j] \vDash r_1 \vee w, [i,j] \vDash r_2 \\
w, [i,j] \vDash r_1 \cdot r_2 &\iff \exists k \in [i,j] \text{ s.t. } \begin{pmatrix} w, [i,k] \vDash r_1 \\ \wedge w, [k,j] \vDash r_2 \end{pmatrix} \\
w, [i,j] \vDash r^* &\iff i = j \vee \exists k \in [i,j] \text{ s.t. } \begin{pmatrix} w, [i,k] \vDash r \\ \wedge w, [k,j] \vDash r^* \end{pmatrix} \\
w, [i,j] \vDash (?{>}\, r) &\iff i = j \wedge w, [i, |w|] \vDash r \\
w, [i,j] \vDash (?{\not>}\, r) &\iff i = j \wedge w, [i, |w|] \nvDash r \\
w, [i,j] \vDash (?{<}\, r) &\iff i = j \wedge w, [0, i] \vDash r \\
w, [i,j] \vDash (?{\not<}\, r) &\iff i = j \wedge w, [0, i] \nvDash r
\end{aligned}$$

Figure 1. The *satisfaction relation* $\vDash$ relating a string $w \in \Sigma^*$, a window $[i,j]$ with $0 \le i \le j \le |w|$, and a regular expression $r \in$ LReg, possibly with lookarounds.

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 a string $w \in \Sigma^*$, we will call a formal pair $[i,j]$ with $0 \le i \le j \le |w|$ a *window* in w. We use some additional notation throughout the paper. We write r^+ to denote $r \cdot r^*$, i.e., the repetition of r one or more times.

In Coq, LReg is encoded as an inductive data type `LRegex`. We use two inductively defined predicates `match_regex`, and `not_match_regex`. The Proposition `match_regex r w s d` and `not_match_regex r w s d` stand for the propositions $w, [s, s+d] \vDash r$ and its negation $w, [s, s+d] \nvDash r$, respectively. These two predicates are defined using mutual recursion.

We define `not_match_regex` directly, instead of encoding it as the negation of `match_regex`. This is because such an encoding cannot be used in the arguments of the constructor `NegLookAhead` or `NegLookBehind`. The encoding $\sigma \to \bot$ of the negation $\neg\sigma$ as an argument of a constructor would violate strict positivity. We prove the two following lemmas which show that the two predicates behave in the manner expected. Note that a different way of dealing with negation would be to encode the $\vDash$ relation as a fixpoint returning a `Prop` (as in [45], for example).

```
Lemma match_not_match : forall r w start delta,
    ~ (match_regex r w start delta)
    <-> (not_match_regex r w start delta).

Lemma match_lem : forall r w start delta,
    match_regex r w start delta
    \/ ~ (match_regex r w start delta).
```

In our Coq formalization, we choose to work with the representation $w, [s, s+d] \vDash r$, rather than $w, [i,j] \vDash r$. This

is so that we can avoid dealing with additional ordering constraints $i \leq j$ while dealing with proofs.

Example 2. Our use of predicates on the alphabet correspond to character classes in PCRE [43] terminology. For instance, one could use the character class `\d` or `[0-9]` to indicate the predicate that matches a digit. The slight difference in the semantics of lookarounds in our notation, in contrast with PCRE, is that our lookaheads (respectively, lookbehinds) match until the end (respectively, beginning) of the string. The positive PCRE lookahead `(?=r)` is equivalent to our $(?{>}\, r\Sigma^*)$ and our lookahead $(?{>}\, r)$ is equivalent to the PCRE `(?=r$)`.

Consider the expression

$$r = \texttt{(?<=c((?!\d\d).)+)(a+)b(?=.*d.*e)}$$

in PCRE notation. Writing δ for the character-class `\d`, we can translate this expression into our notation as

$$r = (?{<}\, \Sigma^* c \cdot ((?{\not}\, \delta\delta\Sigma^*)\Sigma)^+) \cdot a^+ b \cdot (?{>}\, \Sigma^* d \Sigma^* e \Sigma^*).$$

For the string w shown below, there are four potential matches for the subexpression a^+b, as indicated below by highlighting the substrings corresponding to their windows at $[2,4]$, $[9,12]$, $[10,12]$ and $[14,16]$:

$$c\ c\ \underline{a\ b}\ c\ 7\ 7\ d\ c\ \overline{a\ \underline{a\ b}}\ 7\ d\ \underline{a\ b}\ e$$

The window $[9,12]$ does not satisfy the lookbehind, since the Kleene $^+$ requires at least one character after the occurrence of c before the match window. The window $[2,4]$ does not satisfy the lookbehind either. This is because even though there is a c at position 0, the substring 77 appearing at $[5,7]$ in the future disqualifies it. The window $[14,16]$ does not satisfy the lookahead since there is no d (or e) in the future. It can be checked that only the window $[10,12]$ is a match.

Equational Reasoning. Using the semantics above, we can set up a framework for equational reasoning on regular expressions with lookarounds.

Definition 3 (Equivalence and Containment Relations on Regular Expressions)**.** Let $r, s \in \mathsf{LReg}$. We say that r and s are *equivalent*, and we write $r \equiv s$, if for every string $w \in \Sigma^*$ and every window $[i,j]$ of w, we have $w, [i,j] \vDash r$ if and only if $w, [i,j] \vDash s$. We also define the *containment* relation $r \sqsubseteq s$ as notational shorthand for $r + s \equiv s$.

We encode the $\equiv$ and $\sqsubseteq$ relations in Coq as follows:

```
Definition regex_eq (r1 r2 : LRegex) : Prop :=
    forall w s d,
        match_regex r1 w s d <-> match_regex r2 w s d.
Definition regex_leq (r s : Regex) : Prop :=
    regex_eq (Union r s) s.
```

We notice that this is an equivalence relation and a congruence with respect to the regular expression combinators such as concatenation, union, Kleene iteration and the lookarounds. In the terminology of Coq, these congruence relations can be expressed as *Proper morphisms* [41]. Declaring them in this manner allows standard rewrite tactics to use $\equiv$-based equations for rewriting inside proofs.

```
Instance regex_eq_equiv : Equivalence regex_eq.
Instance union_proper : Proper
  (regex_eq ==> regex_eq ==> regex_eq) Union.
Instance concat_proper : Proper
  (regex_eq ==> regex_eq ==> regex_eq) Concat.
Instance star_proper : Proper
  (regex_eq ==> regex_eq) Star.
...
```

An equivalent approach is to define $r \sqsubseteq s$ so that it holds when for every $w \in \Sigma^*$ and every window $[i,j]$ of w, we have $w, [i,j] \vDash r \implies w, [i,j] \vDash s$. We prove that these two definitions are equivalent.

```
Lemma subset_leq : forall (r s : LRegex),
  regex_leq r s <-> (forall (w : list A) (start d : nat),
    match_regex r w start d
    -> match_regex s w start d).
```

We have shown that $\sqsubseteq$ is a partial order and the operations of union and concatenation are monotone with respect to this order. These monotonicity theorems can also be expressed using proper morphisms.

```
Instance regex_leq_partialorder :
  PartialOrder regex_eq regex_leq.
Instance union_monotone : Proper
  (regex_leq ==> regex_leq ==> regex_leq) Union.
Instance concat_monotone : Proper
  (regex_leq ==> regex_leq ==> regex_leq) Concat.
```

The purpose of verifying these properties is to streamline reasoning about regular expressions with lookarounds into an equational manner when possible. In [31], the authors have proven several identities resembling the properties of Kleene Algebra [27] for the equivalence relation $\equiv$, along with several properties for lookarounds. We have verified them in Coq. In future work, this style of reasoning could serve as the basis for proving the correctness of simplification algorithms for expressions with lookarounds.

3 Oracles for Lookaround Assertions

In this section, we will assume that relevant information about lookaround assertions can be accessed using oracle queries. In the algorithm we describe later in Section 5 which is based on decomposing the given expression into layers, this information for the inner layers will be computed in a prior step. We provide an automata-theoretic framework describing how such regular expressions with oracle queries can be handled. To do this, we first make the notion of strings augmented with oracle valuations explicit. An extension of regular expressions which can query these oracles is presented. The idea is to replace lookaround expressions with these oracle queries, so that information involving

lookarounds can be answered using an oracle. These oracle-based expressions can be matched using a model that is an extension of NFAs. We show that these NFAs can be simulated using a purely functional algorithm. Similar to a usual NFA, this simulation is done in a single left-to-right pass, consuming the input word (including the oracle valuations) one character at a time, taking $O(m)$ time at each step, where m is the size of the regular expression.

3.1 Oracle Strings and Oracle Regular Expressions

Suppose V is a finite set of (Boolean) variables. A valuation of V is a truth assignment to the elements of V. We will denote the set of valuations of V as 2^V. An *o-string* is a pair $\langle w, \beta \rangle$ where $w \in \Sigma^*$ and $\beta \in (2^V)^*$ such that $|\beta| = |w| + 1$. The set of o-strings over the alphabet Σ and variables V is denoted $\mathcal{O}(\Sigma, V)$.

These represent strings together with additional information accessible via oracle queries. Suppose $w = a_0 \cdot a_1 \cdots a_{n-1}$ and $\beta = \beta_0 \cdot \beta_1 \cdot \beta_2 \cdots \beta_n$. Informally, we wish to associate each character a_i with the oracle valuation β_i to its left and β_{i+1} to its right. Thus, the o-string $\langle w, \beta \rangle$ could be more visually represented as

$$\beta_0\ a_0\ \beta_1\ a_1\ \beta_2\ a_2\ \beta_3 \cdots \beta_{n-1}\ a_{n-1}\ \beta_n.$$

This explains the constraints on the length of the components of o-strings. The length of the o-string $\langle w, \beta \rangle$ written $|\langle w, \beta \rangle|$ is defined to be $|w|$. Note that this means that there may be more than one o-string of length 0, since such an o-string consists of a single oracle valuation.

In Coq, we define valuations as list of Booleans, and o-strings simply as pairs of lists. Instead of working with dependent types, we enforce the constraints on the lengths (i.e, that $|\beta| = |w| + 1$, and that each valuation has the same number of Booleans) using a separate predicate.

```
Definition valuation : Type := list bool.
Definition ostring : Type := (list A) * (list valuation).

Definition outer_length_wf (s : ostring) : Prop :=
  length (fst s) + 1 = length (snd s).
Definition inner_length_wf (s : ostring) : Prop :=
  forall u v : valuation,
    In u (snd s) -> In v (snd s) -> length u = length v.
Definition ostring_wf (s : ostring) : Prop :=
  outer_length_wf s /\ inner_length_wf s.

Definition olength (s : ostring) : nat := length (fst s).
```

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

Note that the slice $[i, j]$ can be obtained by taking the slice $[0, j]$ to obtain $\langle w', \beta' \rangle$ and then selecting the slice $[i, |w'|]$. This is how we encode the notion of slicing in our Coq formalization. We define the functions oskipn and ofirstn corresponding to the slices $[i, |w|]$ and $[0, i]$ respectively. These mirror the definition of the functions skipn and firstn in the Coq standard library, and satisfy analogous lemmas. Note the slight asymmetry in the definitions below, resulting from the mismatch in the lengths of the components of o-strings.

```
Definition ofirstn (n : nat) (s : ostring) : ostring :=
  (firstn n (fst s), firstn (S n) (snd s)).
Definition oskipn (n : nat) (s : ostring) : ostring :=
  (skipn n (fst s), skipn (min n (length (fst s))) (snd s)).
```

The concatenation operation on $\mathcal{O}(\Sigma, V)$ needs to be defined carefully so that it matches the way we intend to use these strings. The concatenation of two elements of $\mathcal{O}(\Sigma, V)$ is only defined if the oracle-values agree. Formally, suppose $\langle w_1, \beta \cdot u \rangle, \langle w_2, v \cdot \gamma \rangle \in \mathcal{O}(\Sigma, V)$ with $u, v \in 2^V$, then $\langle w_1, \beta \cdot u \rangle \cdot \langle w_2, v \cdot \gamma \rangle$ is defined iff $u = v$. The concatenation $\langle w_1, \beta \cdot u \rangle \cdot \langle w_2, u \cdot \gamma \rangle$ is defined to be $\langle w_1 w_2, \beta \cdot u \cdot \gamma \rangle$. 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 4 (Oracle Regular Expressions). The set OReg of *oracle regular expressions* is defined with the following grammar:

$$R, S ::= \varepsilon \mid \sigma \in \mathcal{P} \mid \mathrm{Q}^+(v \in V) \mid \mathrm{Q}^-(v \in V) \mid R \cdot S \mid R + S \mid R^*.$$

Instead of lookaheads as in LReg, oracle regular expressions have positive queries of the form $\mathrm{Q}^+(v)$ or negative queries of the form $\mathrm{Q}^-(v)$, where $v \in V$. These queries express assertions on the accompanying oracle valuations. The semantics of a OReg expression R is given in terms of $[\![R]\!]$, a subset of $\mathcal{O}(\Sigma, V)$. This language is defined inductively as follows:

$$[\![\varepsilon]\!] = \{\langle \varepsilon, \beta_0 \rangle \mid \beta_0 \in 2^V\}$$
$$[\![\sigma]\!] = \{\langle a, \beta_0 \cdot \beta_1 \rangle \mid a \in \Sigma, \beta_0, \beta_1 \in 2^V, \sigma(a) = 1\}$$
$$[\![\mathrm{Q}^+(v)]\!] = \{\langle \varepsilon, \beta_0 \rangle \mid \beta_0 \in 2^V, \beta_0[v] = 1\}$$
$$[\![\mathrm{Q}^-(v)]\!] = \{\langle \varepsilon, \beta_0 \rangle \mid \beta_0 \in 2^V, \beta_0[v] = 0\}$$
$$[\![R + S]\!] = [\![R]\!] \cup [\![S]\!] \quad [\![R \cdot S]\!] = [\![R]\!] \cdot [\![S]\!] \quad [\![R^*]\!] = [\![R]\!]^*$$

Example 5. Consider the oregex $r_1 = \mathrm{Q}^+(v_0) \cdot a^+ b \cdot \mathrm{Q}^+(v_1)$. Here, the queries express the constraint that the valuation corresponding to the variable v_0 at the beginning of the string, and the valuation corresponding to the variable v_1 at the end of the string are both true. Concretely, we have $\langle w, \beta \rangle \in [\![r]\!]$ iff $w \in [\![a^+ b]\!]$ and $\beta_0[v_0] = \beta_{|w|}[v_1] = 1$.

Consider the oregex $r_2 = c{\cdot}(\mathrm{Q}^-(v_2)\Sigma)^+$. Here, the negative query $\mathrm{Q}^-(v_2)$ asserts that the valuation corresponding to the variable v_2 be false before each character that matches the subexpression Σ. More concretely, suppose $\langle w, \beta\rangle \in [\![r_2]\!]$. Then, we have that $w_0 = c$ and $|w| > 1$ (enforced by the Kleene plus). For $1 \le i \le |w| - 1$, we must have $\beta_i[v_2] = 0$. Note that no constraints are placed on β_0 or $\beta_{|w|}$.

In our Coq formalization, we represent objects of type OReg as the inductive type `ORegular`. In order to represent membership in the set $[\![R]\!]$, we work with a satisfaction relation `match_oregex`. This is similar to our handling of LReg, except that the complication involving handling negation does not appear here. The case for concatenation and Kleene iteration, however, is interesting. This is because concatenation on o-strings is defined only when the oracle valuations agree. We resolve this issue by phrasing the concatenation in terms of the slicing operation. In particular, we say that $\langle w, \beta\rangle \in [\![R \cdot S]\!]$ iff there exists an i such that $\langle w, \beta\rangle[0, i] \in [\![R]\!]$ and $\langle w, \beta\rangle[i, |w|] \in [\![S]\!]$. The Kleene iteration is handled in a similar manner. These two cases are shown below.

```
...
| omatch_concat :
    forall (r1 r2 : ORegex) (os : ostring) (n : nat),
    match_oregex r1 (ofirstn n os)
    -> match_oregex r2 (oskipn n os)
    -> match_oregex (OConcat r1 r2) os
...
| omatch_star_cons :
    forall (r : ORegex) (os : ostring) (n : nat),
    match_oregex r (ofirstn n os)
    -> match_oregex (OStar r) (oskipn n os)
    -> match_oregex (OStar r) os
...
```

3.2 Choosing Appropriate Oracle Valuations

In this section, we will make concrete the connection between LReg and OReg. In particular, we will show that if the lookaround assertions are pre-evaluated, then they could be used as valuations for o-strings which could be supplied to an oracle regular expression obtained by replacing the lookarounds with queries.

Definition 6 (Maximal Lookarounds, Arity). Let $r \in$ LReg be an expression. The maximal lookarounds of r, written $\mathsf{maxLk}(r)$ is a list of tuples of type $\{\rhd, \lhd\} \times$ LReg. It is formally defined as follows:

$$\mathsf{maxLk}((?{>}\, r)) = [(\lhd, r)] \quad \mathsf{maxLk}((?{<}\, r)) = [(\rhd, r)]$$
$$\mathsf{maxLk}((?{\not>}\, r)) = [(\lhd, r)] \quad \mathsf{maxLk}((?{\not<}\, r)) = [(\rhd, r)]$$
$$\mathsf{maxLk}(r) = [], \text{ if } r \in \{\varepsilon, (\sigma \in \mathcal{P})\}$$
$$\mathsf{maxLk}(r_1 \circ r_2) = \mathsf{maxLk}(r_1) +\!\!+ \mathsf{maxLk}(r_2), \text{ if } \circ \in \{\cdot, +\}$$
$$\mathsf{maxLk}(r^*) = \mathsf{maxLk}(r)$$

where the notation ++ refers to list concatenation.

If $(\lhd, s)$ or $(\rhd, s)$ appears in $\mathsf{maxLk}(r)$, then we say that s is a *maximal lookaround* of r. The *arity* of an expression r is the number of its maximal lookarounds, i.e., $\mathsf{arity}(r) = |\mathsf{maxLk}(r)|$.

Note that the directions associated with the definitions above may be counter-intuitive. As we will see in subsection 5.1, lookbehinds need the scanning of the string from left to right, while lookaheads need the scanning from right to left.

Definition 7 (Abstraction). The abstraction $\mathsf{abstract}(r)$ of a regular expression r is obtained by replacing each maximal lookaround with $\mathrm{Q}^+(v)$ or $\mathrm{Q}^-(v)$. The variables here are chosen from the set $V = \{v_i \mid i < |\mathsf{arity}(r)|\}$. Formally, $\mathsf{abstract}(r)$ is defined to be $\mathsf{abstract}_0(r)$, and given $i \in \mathbb{N}$, we define $\mathsf{abstract}_i(r)$ as follows:

$$\mathsf{abstract}_i(r) = r, \text{ if } r \in \{\varepsilon, (\sigma \in \mathcal{P})\}$$
$$\mathsf{abstract}_i(r_1 \cdot r_2) = \mathsf{abstract}_i(r_1) \cdot \mathsf{abstract}_{i+\mathsf{arity}(r_1)}(r_2)$$
$$\mathsf{abstract}_i(r_1 + r_2) = \mathsf{abstract}_i(r_1) + \mathsf{abstract}_{i+\mathsf{arity}(r_1)}(r_2)$$
$$\mathsf{abstract}_i(r^*) = \mathsf{abstract}_i(r)^*$$
$$\mathsf{abstract}_i(r) = \mathrm{Q}^+(v_i), \text{ if } r \in \{(?{>}\, s), (?{<}\, s)\}$$
$$\mathsf{abstract}_i(r) = \mathrm{Q}^-(v_i), \text{ if } r \in \{(?{\not>}\, s), (?{\not<}\, s)\}$$

Example 8. Consider again the expression

$$r = (?{<}\, \Sigma^* c \cdot ((?{\not>}\, \delta\delta\Sigma^*)\Sigma)^+) \cdot a^+ b \cdot (?{>}\, \Sigma^* d \Sigma^* e \Sigma^*)$$

from Example 2. We have that $\mathsf{maxLk}(r) = [m_0, m_1]$ where $m_0 = (\rhd, \Sigma^* c \cdot ((?{\not>}\, \delta\delta\Sigma^*)\Sigma)^+)$ and $m_1 = (\lhd, \Sigma^* d \Sigma^* e \Sigma^*)$. Therefore, we have $\mathsf{arity}(r) = 2$. Note that even though $(?{\not>}\, \delta\delta\Sigma^*)$ is a lookahead that appears as a subexpression, it is not considered *maximal*.

The abstraction of r is given by

$$\mathsf{abstract}(r) = \mathsf{abstract}_0(r) = \mathrm{Q}^+(v_0) \cdot a^+ b \cdot \mathrm{Q}^+(v_1)$$

. If we consider the subexpression $s = \Sigma^* c \cdot ((?{\not>}\, \delta\delta\Sigma^*)\Sigma)^+$ of r, then we have $\mathsf{abstract}(s) = \Sigma^* c \cdot (\mathrm{Q}^-(v_0) \cdot \Sigma)^+$.

Next, we define the notion of *tapes* for expressions and lookarounds, which are sequences of truth values obtained by incrementally scanning the string.

Definition 9 (Tapes, Oracle Valuations). Let $r \in$ LReg be an expression and $w \in \Sigma^*$ be a string such that $|w| = n$. We define the left-to-right tape $\mathrm{tape}(\rhd, r, w)$ and the right-to-left tape $\mathrm{tape}(\lhd, r, w)$ as sequences of length $n + 1$ over $\{0, 1\}$ such that for each $0 \le i \le n$, the i-th entry of the tape satisfies the following:

$$\mathrm{tape}(\rhd, r, w)[i] = \begin{cases} 1 & \text{if } w, [0, i] \vDash r \\ 0 & \text{if } w, [0, i] \nvDash r \end{cases}$$

$$\mathrm{tape}(\lhd, r, w)[i] = \begin{cases} 1 & \text{if } w, [i, |w|] \vDash r \\ 0 & \text{if } w, [i, |w|] \nvDash r \end{cases}$$

When there is no confusion, we will write tape to mean $\mathrm{tape}(\rhd)$.

We also extend the notion of tapes to OReg and maximal lookarounds. For $s \in \mathsf{OReg}$ with $\langle w, \beta \rangle \in \mathcal{O}(\Sigma, V)$, we define $\mathsf{tape}(s, \langle w, \beta \rangle)$ in a similar manner as above, indicating the truth values of the slices $\langle w, \beta \rangle[0, i]$ for $\mathsf{tape}(\triangleright)$ and $\langle w, \beta \rangle[i, |w|]$ for $\mathsf{tape}(\triangleleft)$. Given a maximal lookaround of the form $m = (d, r)$, where $d \in \{\triangleright, \triangleleft\}$ and $r \in \mathsf{LReg}$, we define $\mathsf{tape}((d, r))$ in a natural way as $\mathsf{tape}((\triangleright, r), w) = \mathsf{tape}(\triangleright, r, w)$ and $\mathsf{tape}((\triangleleft, r), w) = \mathsf{tape}(\triangleleft, r, w)$.

Let $r \in \mathsf{LReg}$ be of arity k. Let T be a collection of k tapes s.t.for each $0 \le i < k$, tape $T[i] = \mathsf{tape}(\mathsf{maxLk}(r)[i], w)$. Let $V = v_0, v_1, \ldots v_{k-1}$. The oracle valuations for r on w, written $\mathsf{oval}(r, w)$ is the sequence of length $|w| + 1$ over 2^V obtained by transposing T. In other words,

$$\mathsf{oval}(r, w)[i][v_j] = T[j][i], \text{ for } 0 \le i \le |w|, 0 \le j < k$$

Example 10. Consider r and w from Example 2, whose abstraction and maximal lookarounds were shown in Example 8. The maximal lookaround m_0 searches for the character c in the past followed by a non-empty sequence of characters satisfying certain conditions, while the maximal lookaround m_1 searches for an occurrence of the character d followed by the character e in the future. The columns below show the word w and the tapes for $\mathsf{tape}(m_0, w)$ and $\mathsf{tape}(m_1, w)$, in that order. The oracle valuations $\mathsf{oval}(r, w)$ for r on w can be obtained by reading the table below in a column-wise manner.

	c	c	a	b	c	7	7	d	c	a	a	b	7	d	a	b	e
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

In our formalization, we define `is_lookaround_tape` to encode the notion of tape for maximal lookarounds, (i.e., pairs $\{\triangleright, \triangleleft\} \times \mathsf{LReg}$). The definition of oval is instrumented using an additional index s which corresponds to the subscript $\mathsf{abstract}_s$. This is necessary in order to formulate a stronger inductive hypothesis we can induct on. Note also that we phrase this definition as a relation rather than a function.

```
Definition oval_tapes_aux (e : @LRegex A) (w : list A)
  (s : nat) (ts : list tape) : Prop :=
  length ts >= s + arity e
  /\ (forall t, In t ts -> length t = length w + 1)
  /\ forall i t r, i < arity e
    -> nth_error ts (s + i) = Some t
    -> nth_error (maximal_lookarounds e) i = Some r
    -> is_lookaround_tape r w t.

Definition is_oval_aux (r : @LRegex A) (w : list A)
  (s : nat) (vs : list valuation) : Prop :=
  exists ts, oval_tapes_aux r w s ts
    /\ vs = transpose (length w + 1) ts.

Definition is_oval (r : @LRegex A) (w : list A)
  (vs : list valuation) : Prop :=
  is_oval_aux r w 0 vs.
```

The following lemma establishes the connection between LReg and OReg by choosing the appropriate oracle valuations.

Lemma 11. Let $r \in \mathsf{LReg}$ and $w \in \Sigma^*$. Let $\beta = \mathsf{oval}(r, w)$. Then, the following holds.

$$w, [i, j] \vDash r \iff \langle w, \beta \rangle[i, j] \in [\![\mathsf{abstract}(r)]\!]$$

Thus, in order to match an expression r with lookarounds on w, we can first evaluate the lookarounds to obtain the oracle valuations $\beta = \mathsf{oval}(r, w)$, and then match the oracle regular expression $\mathsf{abstract}(r)$ on $\langle w, \beta \rangle$. This formalizes the key idea behind our approach.

In Coq, this lemma is stated in terms of the `ofirstn` and `oskipn` functions. Recall that the index s is used to keep track of the subscript of $\mathsf{abstract}_s$.

```
Lemma oracle_compose_aux (r : @LRegex A) (w : list A)
  (s : nat) (vs : list valuation) :
  is_oval_aux r w s vs
  -> forall start delta, start + delta <= length w
    -> match_regex r w start delta
    <-> match_oregex (abstractAux s r)
      (ofirstn delta (oskipn start (w, vs))).
```

4 Purely Functional Matching of Oracle Expressions

In this section, we discuss how the oracle regular expressions can be matched on oracle strings, i.e., the computation of tapes (from Definition 9 for the case of OReg. This is very similar to the matching algorithm for ordinary regular expressions (without queries or lookarounds), except that the queries need to be checked against the oracle valuations.

The main difference between handling character-predicates and queries can be understood by analyzing the case of concatenation. For example, consider the regular expression $\sigma_1 \cdot \sigma_2$ obtained by concatenating two predicates σ_1 and σ_2. This matches a length 2 string where the first character satisfies σ_1 and the second character satisfies σ_2. In contrast, the oracle regular expression $\mathsf{Q}^+(v_1) \cdot \mathsf{Q}^+(v_2)$ matches an o-string $\langle \varepsilon, \beta_0 \rangle$, where $\beta_0[v_1] = \beta_1[v_2] = 1$. Thus, the concatenated queries match *the same oracle valuation*. In [31], this is done using the notion of Oracle NFAs in which the transitions are guarded with queries.

We use a purely functional approach based on marked regular expressions. The notion of marked regular expressions for classical regular expressions is discussed in [19] and [3]. In [36], the authors formalize this approach in Isabelle, and utilize them in order to decide equivalence. Here, we extend this approach to oracle regular expressions, and formalize it in Coq.

The marked regular expression approach directly deals with the syntax tree of regular expressions instead of building an explicit NFA. This is particularly pleasant in functional programming languages where we can represent the

expressions as inductive types and use pattern matching to access the subexpressions. The marks placed on the character classes of the regular expression are used to keep track of the active states of the corresponding NFA.

Definition 12 (Marked Oracle Regular Expressions). The set MReg of *marked oracle regular expressions* is defined with the following grammar:

$$M, N ::= \varepsilon \mid \sigma \mid \underline{\sigma} \mid \mathrm{Q}^+(v) \mid \mathrm{Q}^-(v) \mid M \cdot N \mid M + N \mid M^*$$

where $\sigma \in \mathcal{P}$ and $v \in V$. We call $\underline{\sigma}$ (respectively, σ) a *marked* (respectively, *unmarked*) predicate.

We extend the terminology above to say that an expression is *unmarked* if it contains no marked predicates. Given $r \in$ OReg, we can view it as an element of MReg in which none of the predicates are marked. This expression is written as $\mathsf{toMarked}(r)$. Conversely, given $m \in$ MReg, we can remove all the marks from it to obtain the expression $\mathsf{strip}(m) \in$ OReg. For each $m \in$ MReg, we define a subset $\langle\!\langle r \rangle\!\rangle$ of $\mathcal{O}(\Sigma, V)$ as follows:

$$\begin{aligned}
\langle\!\langle r \rangle\!\rangle &= \varnothing \text{ if } r \in \{\varepsilon, \sigma, \mathrm{Q}^+(v), \mathrm{Q}^-(v)\} \\
\langle\!\langle \underline{\sigma} \rangle\!\rangle &= [\![\sigma]\!] \\
\langle\!\langle e_1 + e_2 \rangle\!\rangle &= \langle\!\langle e_1 \rangle\!\rangle \cup \langle\!\langle e_2 \rangle\!\rangle \\
\langle\!\langle e_1 \cdot e_2 \rangle\!\rangle &= \langle\!\langle e_1 \rangle\!\rangle \cdot [\![\mathsf{strip}(e_2)]\!] \cup \langle\!\langle e_2 \rangle\!\rangle \\
\langle\!\langle e^* \rangle\!\rangle &= \langle\!\langle e \rangle\!\rangle \cdot [\![\mathsf{strip}(e)]\!]^*
\end{aligned}$$

Example 13. The intuition for the language of $\langle\!\langle r \rangle\!\rangle$ is that it represents the set of o-strings that could be matched by starting from the current mark. For example, consider the expressions $m_1 = \underline{a}bc$, $m_2 = a\underline{b}c$ and $m_3 = \underline{a}b\underline{c}$. Let $\langle w, \beta \rangle$ be an o-string. We have $\langle w, \beta \rangle \in \langle\!\langle m_1 \rangle\!\rangle$ iff $w = abc$, $\langle w, \beta \rangle \in \langle\!\langle m_2 \rangle\!\rangle$ iff $w = bc$ and $\langle w, \beta \rangle \in \langle\!\langle m_3 \rangle\!\rangle$ iff $w = abc$ or $w = c$. On the other hand, if $m_4 = abc$ with no marks, then $\langle\!\langle m_4 \rangle\!\rangle = \varnothing$. This can be understood by realizing that this represents a scenario where there are no active states in the simulated NFA. In general, we can observe that for unmarked m, the language $\langle\!\langle m \rangle\!\rangle$ is $\varnothing$.

The queries in expressions in MReg do not carry any marks, but they can still express constraints on valuations if they occur after a marked predicate. For instance, consider $m_5 = \underline{a}\mathrm{Q}^+(v)bc$ and $m_6 = \mathrm{Q}^+(v)\underline{a}bc$. Given $\langle w, \beta \rangle \in \langle\!\langle m_5 \rangle\!\rangle$, we must have $w = abc$ and $\beta_1[v] = 1$. However, for $\langle w, \beta \rangle \in \langle\!\langle m_6 \rangle\!\rangle$, we have no constraints on β.

In Coq, we define an inductive type `MRegex` to represent the set MReg. Similar to the case of LReg and OReg, we represent the notion of $\langle\!\langle \cdot \rangle\!\rangle$ using a satisfaction relation encoded as the inductive type `match_mregex`.

4.1 Operations on Marked Expressions

Our matching algorithm operates by transforming marked regular expressions. The functions nullable and final are used to extract information about the state machine that is being simulated. The functions follow and read presented in Figure 2 manipulate the marks without changing the underlying regex.

Let us start by characterizing the functions nullable and final. In the following, we call a marked predicate $\underline{\sigma}$ *spurious* if the predicate σ is unsatisfiable. If m has no such predicates, we say m has no spurious marks.

Lemma 14. Let $m \in$ MReg. Then, the following holds.

1. For $\beta_0 \in 2^V$, $\mathsf{nullable}(\beta_0, m) = 1$ iff $\langle \varepsilon, \beta_0 \rangle \in [\![\mathsf{strip}(m)]\!]$
2. Suppose m has no spurious marks and $\beta_1 \in 2^V$. If $\mathsf{final}(\beta_1, m) = 1$, then there exists $a \in \Sigma$ such that for all $\beta_0 \in 2^V$, $\langle a, \beta_0\beta_1 \rangle \in \langle\!\langle m \rangle\!\rangle$
3. Let $a \in \Sigma$ and $\beta_0, \beta_1 \in 2^V$. If $\langle a, \beta_0\beta_1 \rangle \in \langle\!\langle m \rangle\!\rangle$, then $\mathsf{final}(\beta_1, m) = 1$

Example 15. Consider the expressions $m_6 = (a + \varepsilon)\mathrm{Q}^+(v)$, and $m_7 = (\underline{a} + \varepsilon)\mathrm{Q}^+(v)$. Since $\mathsf{strip}(m_6) = \mathsf{strip}(m_7)$, we can see that $\mathsf{nullable}(\beta_0, m_6) = \mathsf{nullable}(\beta_0, m_7)$ for any choice of β_0. In particular, $\mathsf{nullable}(\beta_0, m_6) = 1$ iff $\beta_0[v] = 1$. Here, the function nullable tests whether the o-string $\langle \varepsilon, \beta_0 \rangle$ is in $[\![\mathsf{strip}(m_6)]\!]$.

Informally, the function final checks if there is a mark on a predicate that is in the final position, and whether the subsequent queries can be satisfied by the valuation provided. Thus, if $\beta_1[v] = 1$, we do have $\mathsf{final}(\beta_1, m_7) = 1$. For m_6, we have $\mathsf{final}(\beta_0, m_6) = 0$ for any β_0 since there are no marks in m_6. Similarly, $\mathsf{final}(\beta_0, (\underline{a}bc + \varepsilon)\mathrm{Q}^+(v)) = 0$, since the marked character a is not in a final position. Also, $\mathsf{final}(\beta_0, (\underline{a} + \varepsilon)) = 1$ for any β_0 since there are no queries imposing constraints on the valuations.

Next, we characterize the function read. The function $\mathsf{read}(a, \cdot)$ has the effect of removing the marks from marked predicates $\underline{\sigma}$ which do not satisfy $\sigma(a) = 1$. Roughly, this has the effect of removing from $\langle\!\langle m \rangle\!\rangle$ the o-strings which do not start with the character a.

Lemma 16. Let $m \in$ MReg and $a \in \Sigma$. Then, the following holds.

1. $\langle\!\langle \mathsf{read}(a, m) \rangle\!\rangle \subseteq \langle\!\langle m \rangle\!\rangle$
2. Suppose $\langle a \cdot w, \beta \rangle \in \langle\!\langle m \rangle\!\rangle$. Then, $\langle a \cdot w, \beta \rangle \in \langle\!\langle \mathsf{read}(a, m) \rangle\!\rangle$
3. Suppose $a' \in \Sigma$ and $\langle a' \cdot w, \beta \rangle \in \langle\!\langle \mathsf{read}(a, m) \rangle\!\rangle$. Then, $\langle a \cdot w, \beta \rangle \in \langle\!\langle m \rangle\!\rangle$
4. The expression $\mathsf{read}(a, m)$ has no spurious marks

Example 17. Let us use the predicate δ to denote the set of digits. Consider the expression $m = \underline{\delta}abc$. We observe that $\mathsf{read}(7, m) = m$ since the digit 7 satisfies the predicate δ. On the other hand, $\mathsf{read}(a, m) = \delta abc$ – the mark is lost since a does not satisfy the predicate.

In order to prove the properties of the function follow, it is helpful to define two other functions init and shift (also shown in Figure 2). These two functions are related in the

$$\mathsf{nullable} : 2^V \times \mathsf{MReg} \to \mathbb{B}$$
$$\mathsf{nullable}(\beta, \varepsilon) = 1 \qquad \mathsf{nullable}(\beta, m^*) = 1$$
$$\mathsf{nullable}(\beta, \sigma) = 0 \qquad \mathsf{nullable}(\beta, \underline{\sigma}) = 0$$
$$\mathsf{nullable}(\beta, \mathrm{Q}^+(v)) = \beta[v] \qquad \mathsf{nullable}(\beta, \mathrm{Q}^-(v)) = \neg\beta[v]$$
$$\mathsf{nullable}(\beta, m_1 + m_2) = \mathsf{nullable}(\beta, m_1) \vee \mathsf{nullable}(\beta, m_2)$$
$$\mathsf{nullable}(\beta, m_1 \cdot m_2) = \mathsf{nullable}(\beta, m_1) \wedge \mathsf{nullable}(\beta, m_2)$$

$$\mathsf{follow} : \mathbb{B} \times 2^V \times \mathsf{MReg} \to \mathsf{MReg}$$
$$\mathsf{follow}(\beta, \varepsilon) = \varepsilon$$
$$\mathsf{follow}(\beta, \mathrm{Q}^+(v)) = \mathrm{Q}^+(v) \qquad \mathsf{follow}(\beta, \mathrm{Q}^-(v)) = \mathrm{Q}^-(v)$$
$$\mathsf{follow}(b, \beta, \sigma) = \mathsf{follow}(b, \beta, \underline{\sigma}) = \begin{cases} \underline{\sigma} & \text{if } b = 1 \\ \sigma & \text{otherwise} \end{cases}$$
$$\mathsf{follow}(b, \beta, m_1 + m_2) = \mathsf{follow}(b, \beta, m_1) + \mathsf{follow}(b, \beta, m_2)$$
$$\mathsf{follow}(b, \beta, m_1 \cdot m_2) = \mathsf{follow}(b, \beta, m_1) \cdot \mathsf{follow}(b', \beta, m_2)$$
$$\text{where } b' = \mathsf{final}(\beta, m_1) \vee (b \wedge \mathsf{nullable}(\beta, m_1))$$
$$\mathsf{follow}(b, \beta, m^*) = (\mathsf{follow}(b \vee \mathsf{final}(\beta, m), \beta, m))^*$$

$$\mathsf{shift} : 2^V \times \mathsf{MReg} \to \mathsf{MReg}$$
$$\mathsf{shift}(\beta, r) = r, \text{ if } r \in \{\varepsilon, \mathrm{Q}^+(v), \mathrm{Q}^-(v)\}$$
$$\mathsf{shift}(\beta, \sigma) = \mathsf{shift}(\beta, \underline{\sigma}) = \sigma$$
$$\mathsf{shift}(\beta, m_1 + m_2) = \mathsf{shift}(\beta, m_1) + \mathsf{shift}(\beta, m_2)$$
$$\mathsf{shift}(\beta, m_1 \cdot m_2) = \mathsf{shift}(\beta, m_1) \cdot m_2'', \text{ where}$$
$$m_2'' = \begin{cases} \mathsf{init}(\beta, \mathsf{shift}(\beta, m_2)) & \text{if } \mathsf{final}(\beta, m_1) = 1 \\ \mathsf{shift}(\beta, m_2) & \text{otherwise} \end{cases}$$
$$\mathsf{shift}(\beta, m^*) = (m'')^*, \text{ where}$$
$$m'' = \begin{cases} \mathsf{init}(\beta, \mathsf{shift}(\beta, m)) & \text{if } \mathsf{final}(\beta, m) = 1 \\ \mathsf{shift}(\beta, m) & \text{otherwise} \end{cases}$$

$$\mathsf{final} : 2^V \times \mathsf{MReg} \to \mathbb{B}$$
$$\mathsf{final}(\beta, \varepsilon) = 0 \qquad \mathsf{final}(\beta, m^*) = \mathsf{final}(\beta, m)$$
$$\mathsf{final}(\beta, \sigma) = 0 \qquad \mathsf{final}(\beta, \underline{\sigma}) = 1$$
$$\mathsf{final}(\beta, \mathrm{Q}^+(v)) = 0 \qquad \mathsf{final}(\beta, \mathrm{Q}^-(v)) = 0$$
$$\mathsf{final}(\beta, m_1 + m_2) = \mathsf{final}(\beta, m_1) \vee \mathsf{final}(\beta, m_2)$$
$$\mathsf{final}(\beta, m_1 \cdot m_2) = (\mathsf{final}(\beta, m_1) \wedge \mathsf{nullable}(\beta, m_2)) \vee \mathsf{final}(\beta, m_2)$$

$$\mathsf{read} : \Sigma \times \mathsf{MReg} \to \mathsf{MReg}$$
$$\mathsf{read}(a, \varepsilon) = \varepsilon$$
$$\mathsf{read}(a, \mathrm{Q}^+(v)) = \mathrm{Q}^+(v) \qquad \mathsf{read}(a, \mathrm{Q}^-(v)) = \mathrm{Q}^-(v)$$
$$\mathsf{read}(a, \sigma) = \sigma \qquad \mathsf{read}(a, \underline{\sigma}) = \begin{cases} \underline{\sigma} & \text{if } \sigma(a) = 1 \\ \sigma & \text{otherwise} \end{cases}$$
$$\mathsf{read}(a, m_1 + m_2) = \mathsf{read}(a, m_1) + \mathsf{read}(a, m_2)$$
$$\mathsf{read}(a, m_1 \cdot m_2) = \mathsf{read}(a, m_1) \cdot \mathsf{read}(a, m_2)$$
$$\mathsf{read}(a, m^*) = \mathsf{read}(a, m)^*$$

$$\mathsf{init} : 2^V \times \mathsf{MReg} \to \mathsf{MReg}$$
$$\mathsf{init}(\beta, r) = r, \text{ if } r \in \{\varepsilon, \mathrm{Q}^+(v), \mathrm{Q}^-(v)\}$$
$$\mathsf{init}(\beta, \sigma) = \mathsf{init}(\beta, \underline{\sigma}) = \underline{\sigma}$$
$$\mathsf{init}(\beta, m_1 + m_2) = \mathsf{init}(\beta, m_1) + \mathsf{init}(\beta, m_2)$$
$$\mathsf{init}(\beta, m_1 \cdot m_2) = \mathsf{init}(\beta, m_1) \cdot m_2', \text{where}$$
$$m_2' = \begin{cases} \mathsf{init}(\beta, m_2) & \text{if } \mathsf{nullable}(\beta, m_1) = 1 \\ m_2 & \text{otherwise} \end{cases}$$
$$\mathsf{init}(\beta, m^*) = \mathsf{init}(\beta, m)^*$$

Figure 2. Operations on MReg expressions

following way:

$$\mathsf{follow}(0, \beta_0, m) = \mathsf{shift}(\beta_0, m)$$
$$\mathsf{follow}(1, \beta_0, m) = \mathsf{init}(\beta_0, \mathsf{shift}(\beta_0, m))$$

These equivalences can be proven using a straightforward induction once the idempotence of the function init is shown. Informally, the function init is used to place marks at initial positions in the expression, and the function shift is used to move the marks to the next positions. This intuition is formalized in the following lemmas.

Lemma 18. Let $m \in \mathsf{MReg}$ and $\beta_0, \beta_* \in 2^V$, $\beta \in (2^V)^*$ and $w \in \Sigma^*$. Then, the function init satisfies the following properties.

1. $\langle\!\langle m \rangle\!\rangle \subseteq \langle\!\langle \mathsf{init}(\beta_0, m) \rangle\!\rangle$
2. Suppose $w \neq \varepsilon$. Suppose $\langle w, \beta_0 \cdot \beta \rangle \in [\![\mathsf{strip}(m)]\!]$. Then, $\langle w, \beta_0 \cdot \beta \rangle \in \langle\!\langle \mathsf{init}(\beta_0, m) \rangle\!\rangle$.
3. Suppose $\langle w, \beta_* \cdot \beta \rangle \in \langle\!\langle \mathsf{init}(\beta_0, m) \rangle\!\rangle$. Then either $\langle w, \beta_* \cdot \beta \rangle \in \langle\!\langle m \rangle\!\rangle$ or $\langle w, \beta_0 \cdot \beta \rangle \in [\![\mathsf{strip}(m)]\!]$.

Next, let $\beta_1 \in 2^V$ and $a_0 \in \Sigma$. Then, the function shift satisfies the following properties.

1. Suppose $w \neq \varepsilon$ and $\langle a_0 \cdot w, \beta_0\beta_1 \cdot \beta \rangle \in \langle\!\langle m \rangle\!\rangle$. Then, $\langle w, \beta_1 \cdot \beta \rangle \in \langle\!\langle \mathsf{shift}(\beta_1, m) \rangle\!\rangle$.
2. Suppose $\langle w, \beta_* \cdot \beta \rangle \in \langle\!\langle \mathsf{shift}(\beta_1, m) \rangle\!\rangle$. Then, there exists some $a_0 \in \Sigma$ such that forall $\beta_0 \in 2^V$, $\langle a_0 \cdot w, \beta_0\beta_1 \cdot \beta \rangle \in \langle\!\langle m \rangle\!\rangle$.

Example 19. We have $\mathsf{init}(\beta_0, abc + de) = \underline{a}bc + \underline{d}e$ for all β_0. Shifting would move each mark to the next position, i.e., $\mathsf{shift}(\beta_0, \underline{a}bc + \underline{d}e) = a\underline{b}c + d\underline{e}$. When there are queries guarding the expression, the supplied valuation must satisfy the query in order to place the mark. For example, $\mathsf{init}(\beta_0, \mathrm{Q}^+(v)abc) = \mathrm{Q}^+(v)\underline{a}bc$ only $\beta_0[v] = 1$. Similarly, $\mathsf{shift}(\beta_0, \underline{a}\mathrm{Q}^+(v)bc) = a\mathrm{Q}^+(v)\underline{b}c$ only if $\beta_0[v] = 1$.

4.2 Caching final and nullable for Marked Expressions

We wish to compute $\mathsf{follow}(m)$ in $O(|m|)$ time. However, translating the definitions from Figure 2 will result in an algorithm that would take $O(|m|^2)$ time in the worst case. This is because the functions final and nullable are called repeatedly on the same subexpressions. This can be avoided by augmenting the type of expressions to store these additional results. We call these expressions *marked expressions with caching*, denoted by CMReg. In Coq, we define the mutually inductive types `CMRegex` and `CMRe` as follows. Note that with

this definition, the functions cFinal and cNullable are simply field accessors which run in $O(1)$ time, which store the results of final and nullable respectively.

```
Inductive CMRegex : Type :=
  | MkCMRegex : bool -> bool -> CMRe -> CMRegex
with CMRe : Type :=
  | CMEpsilon : CMRe
  ...
  | CMStar : CMRegex -> CMRe.

Definition cNullable (r : CMRegex) : bool :=
  match r with | MkCMRegex b _ _ => b end.
Definition cFinal (r : CMRegex) : bool :=
  match r with | MkCMRegex _ b _ => b end.
Definition cRe (r : CMRegex) : CMRe :=
  match r with | MkCMRegex _ _ re => re end.
```

An interesting consideration here is the induction principle that allows us to prove properties involving CMRegex and CMRe which are mutually recursive. The auto-generated induction principle is insufficient since the subexpressions of CMRe (respectively, CMRegex) are nested inside an additional layer of CMRegex (respectively, CMRe). We write our custom induction principle in the term language, which is later used to prove properties.

Given $c \in \mathsf{CMReg}$, we write $\mathsf{unCache}(c) \in \mathsf{MReg}$ to denote the underlying marked regular expression. We say that c is *synced* to a valuation $\beta_0 \in V$ if $\mathsf{cNullable}(c)$ and $\mathsf{cFinal}(c)$ are $\mathsf{nullable}(\beta_0, \mathsf{unCache}(c))$ and $\mathsf{final}(\beta_0, \mathsf{unCache}(c))$ respectively. In order to streamline our functions and proofs, we define smart constructors mkEps, mkCharClass, etc which propagate the caching information. These have the property that if the supplied arguments are synced with respect to some valuation, then so is the resulting expression. The explicit definition and property for mkConcat are shown. Compare this with the cases of concatenation in the definition of nullable and final in Figure 2.

```
Definition mkConcat (r1 r2 : CMRegex) : CMRegex :=
  MkCMRegex
    (cNullable r1 && cNullable r2)
    ((cFinal r1 && cNullable r2) || cFinal r2)
    (CMConcat r1 r2).
```

We manipulate the cached expressions using the functions sync, cFollow and cRead as defined in Figure 3. The operations cFollow and cRead are the versions of follow and read on MReg expressions, respectively, that work on the cached version. The operation sync is defined in a manner that updates the cNullable and cFinal fields of the expression using a given oracle valuation. All of these functions are defined using the 'smart constructors' mkXXX. Their formal relationship with the corresponding functions for MReg is summarized in the following lemmas.

Lemma 20. Let $m \in \mathsf{MReg}$ and $\beta_0 \in 2^V$. Then, the following holds.

$$
\begin{aligned}
\mathsf{sync} &: 2^V \times \mathsf{CMReg} \to \mathsf{CMReg} \\
\mathsf{sync}(\beta, \varepsilon) &= \mathsf{mkEps} \\
\mathsf{sync}(\beta, \sigma) &= \mathsf{mkCharClass}(0, \sigma) \\
\mathsf{sync}(\beta, \underline{\sigma}) &= \mathsf{mkCharClass}(1, \sigma) \\
\mathsf{sync}(\beta, \mathrm{Q}^+(v)) &= \mathsf{mkQueryPos}(\beta[v], v) \\
\mathsf{sync}(\beta, \mathrm{Q}^-(v)) &= \mathsf{mkQueryNeg}(\beta[v], v) \\
\mathsf{sync}(\beta, m_1 + m_2) &= \mathsf{mkUnion}(\mathsf{sync}(\beta, m_1), \mathsf{sync}(\beta, m_2)) \\
\mathsf{sync}(\beta, m_1 \cdot m_2) &= \mathsf{mkConcat}(\mathsf{sync}(\beta, m_1), \mathsf{sync}(\beta, m_2)) \\
\mathsf{sync}(\beta, m^*) &= \mathsf{mkStar}(\mathsf{sync}(\beta, m))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{cFollow} &: \mathbb{B} \times \mathsf{CMReg} \to \mathsf{CMReg} \\
\mathsf{cFollow}(b, \varepsilon) &= \mathsf{mkEps} \\
\mathsf{cFollow}(b, \sigma) &= \mathsf{cFollow}(b, \underline{\sigma}) = \mathsf{mkCharClass}(b, \sigma) \\
\mathsf{cFollow}(b, \mathrm{Q}^+(v)) &= \mathsf{mkQueryPos}(\mathsf{cNullable}(\mathrm{Q}^+(v)), v) \\
\mathsf{cFollow}(b, \mathrm{Q}^-(v)) &= \mathsf{mkQueryNeg}(\mathsf{cNullable}(\mathrm{Q}^-(v)), v) \\
\mathsf{cFollow}(b, m_1 + m_2) &= \mathsf{mkUnion}(\mathsf{cFollow}(b, m_1), \mathsf{cFollow}(b, m_2)) \\
\mathsf{cFollow}(b, m_1 \cdot m_2) &= \mathsf{mkConcat}(\mathsf{cFollow}(b, m_1), \mathsf{cFollow}(b', m_2)) \\
&\quad \text{where } b' = \mathsf{cFinal}(m_1) \vee (b \wedge \mathsf{cNullable}(m_1)) \\
\mathsf{cFollow}(b, m^*) &= \mathsf{mkStar}(\mathsf{cFollow}(b \vee \mathsf{cFinal}(m), m))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{cRead} &: \Sigma \times \mathsf{CMReg} \to \mathsf{CMReg} \\
\mathsf{cRead}(a, \varepsilon) &= \mathsf{mkEps} \\
\mathsf{cRead}(a, \sigma) &= \mathsf{mkCharClass}(0, \sigma) \\
\mathsf{cRead}(a, \underline{\sigma}) &= \begin{cases} \mathsf{mkCharClass}(1, \sigma) & \text{if } \sigma(a) = 1 \\ \mathsf{mkCharClass}(0, \sigma) & \text{otherwise} \end{cases} \\
\mathsf{cRead}(a, \mathrm{Q}^+(v)) &= \mathsf{mkQueryPos}(\mathsf{cNullable}(\mathrm{Q}^+(v)), v) \\
\mathsf{cRead}(a, \mathrm{Q}^-(v)) &= \mathsf{mkQueryNeg}(\mathsf{cNullable}(\mathrm{Q}^-(v)), v) \\
\mathsf{cRead}(a, m_1 + m_2) &= \mathsf{mkUnion}(\mathsf{cRead}(a, m_1), \mathsf{cRead}(a, m_2)) \\
\mathsf{cRead}(a, m_1 \cdot m_2) &= \mathsf{mkConcat}(\mathsf{cRead}(a, m_1), \mathsf{cRead}(a, m_2)) \\
\mathsf{cRead}(a, m^*) &= \mathsf{mkStar}(\mathsf{cRead}(a, m))
\end{aligned}
$$

Figure 3. Operations on CMReg expressions

1. The expression $\mathsf{sync}(\beta_0, m)$ is synced with respect to β_0 and it preserves the underlying marked expression, i.e., $\mathsf{unCache}(\mathsf{sync}(\beta_0, m)) = m$.
2. If m is synced w.r.t. β_0, then so is $\mathsf{cRead}(\beta_0, m)$.
3. The function cRead simulates read on the underlying MReg expression, i.e., $\mathsf{unCache}(\mathsf{cRead}(a, m)) = \mathsf{read}(a, \mathsf{unCache}(m))$
4. When m is synced to β_0, $\mathsf{unCache}(\mathsf{cFollow}(b, m)) = \mathsf{follow}(b, \beta_0, \mathsf{unCache}(m))$. In other words, the function cFollow simulates follow on the underlying MReg expression.

These lemmas establish the correspondence between the operations on MReg and their cached version CMReg.

4.3 Consuming Oracle Strings

We define the function consume, whose effect is to simulate moving tokens in the corresponding NFA whose paths are labelled by the supplied string. We write toCached : MReg →

CMReg to denote the function which initializes the caching information to 0 for a given expression in MReg.

Definition 21 (Consuming OStrings). Let $r \in \mathsf{OReg}$, and $w = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1} \in \Sigma^*$, and $\beta = \beta_0 \cdot \beta_1 \cdot \ldots \cdot \beta_n \in (2^V)^*$. We define the series of expressions $m_0, m_1 \ldots m_n \in \mathsf{CMReg}$ and associated Boolean values $b_0, b_1 \ldots b_n$ as follows.

$$\begin{aligned} m_0 &= \mathsf{sync}(\beta_0, \mathsf{toCached}(\mathsf{toMarked}(r))) \\ m_1 &= \mathsf{sync}(\beta_1, \mathsf{cRead}(a_0, \mathsf{cFollow}(1, m_0))) \\ m_{i+1} &= \mathsf{sync}(\beta_{i+1}, \mathsf{cRead}(a_i, \mathsf{cFollow}(0, m_i))) \\ b_0 &= \mathsf{cNullable}(m_0) \\ b_{i+1} &= \mathsf{cNullable}(m_{i+1}) \end{aligned}$$

We define $\mathsf{oMatch}(r, \langle w, \beta \rangle) = [b_0, b_1, \ldots, b_n]$.

Theorem 22 (Correctness of oMatch). Let $r \in \mathsf{OReg}$, and $\langle w, \beta \rangle \in \mathcal{O}(\Sigma, V)$. The list $\mathsf{oMatch}(\langle w, \beta \rangle)$ records whether $\langle w, \beta \rangle[0, i] \in [\![r]\!]$, i.e., $\mathsf{oMatch}(r, \langle w, \beta \rangle) = \mathsf{tape}(r, \langle w, \beta \rangle)$.

Proof. (Sketch) The key facts needed are the following: (1) $\langle \varepsilon, \beta_0 \rangle \in [\![r]\!]$ iff $\mathsf{cNullable}(m_0) = 1$, and (2) For $0 < i \leq |w|$, then $\langle w, \beta \rangle[0, i] \in [\![r]\!]$ iff $\mathsf{cFinal}(m_i) = 1$. These would follow from the properties of nullable and final (Lemma 14), and the fact that cFollow and cRead simulate the functions cFollow and cRead from subsection 4.1 (Lemma 20). □

Theorem 23 (Resource Usage of oMatch). Suppose $r \in \mathsf{OReg}$ with $|r| = m$ and $\langle w, \beta \rangle \in \mathcal{O}(\Sigma, V)$ with $|w| = n$. Then, one can compute $\mathsf{oMatch}(r, \langle w, \beta \rangle)$ in $O(m \cdot n)$ time in a streaming manner that requires an additional $O(m)$ state space.

Proof. With the caching trick explained in subsection 4.2, we know that reading off the cFinal and cNullable fields can be done in $O(1)$ time. Similarly, applications of the smart constructors mkEps, mkCharClass, etc also take $O(1)$ time. Using this information, we can show that the functions sync, cFollow and cRead in Figure 4 can be computed in $O(m)$ time, since they access each of their subexpressions exactly once. The function applies cRead using each character in w, sync using each valuation in β, and uses cFollow exactly $|w|$ times. Thus, the functions of complexity $O(m)$ are iterated $O(n)$ times, giving us a total time complexity of $O(m \cdot n)$.

Each bit of oMatch can be computed in a streaming manner: we only need to store the additional state represented as m_i as defined in Definition 21. This requires $O(m)$ space. □

5 Efficient Layerwise Matching

In Section 3, we have shown that oracle regular expressions can be used to evaluate regular expressions with lookahead if the truth values of the lookaheads are supplied as oracle valuations. In Section 4, we have demonstrated how oracle regular expressions can be matched on oracle strings. Now, we will show how to combine these two results to match regular expressions with lookahead in an efficient manner.

5.1 Computing Tapes

In Lemma 11, we see that the appropriate oracle valuations for the oracle regular expression $\mathsf{abstract}(r)$ is given in terms of $\mathsf{oval}(r, w)$, which is defined in terms of tapes of the maximal lookarounds. Here, we show how these can be computed using other existing concepts. One important detail here is that we need to scan the strings in reverse order in order to compute oracle valuations for lookahead.

When $r \in \mathsf{LReg}$ or $r \in \mathsf{OReg}$, we write $\mathsf{rev}(r)$ to denote its reversal, which is defined in the usual way. Notably, rev on LReg interchanges lookaheads and lookbehinds but rev on OReg fixes oracle queries. The reversals satisfy the following useful properties.

Lemma 24. Let $r \in \mathsf{LReg}$, $s \in \mathsf{OReg}$, $\hat{r} = \mathsf{abstract}(r)$, $w \in \Sigma^*$, $\beta = \mathsf{oval}(r, w)$ and $\beta' \in (2^V)^{|w|+1}$. Then, the following hold:

1. $w, [i, j] \vDash r \iff \mathsf{rev}(w), [|w| - j, |w| - i] \vDash \mathsf{rev}(r)$
2. $\langle w, \beta' \rangle, [i, j] \vDash s \iff \langle \mathsf{rev}(w), \mathsf{rev}(\beta') \rangle, [|w| - j, |w| - i] \vDash \mathsf{rev}(s)$
3. $\mathsf{tape}(\triangleleft, r, w) = \mathsf{rev}(\mathsf{tape}(\triangleright, \mathsf{rev}(r), \mathsf{rev}(w)))$
4. $\mathsf{tape}(\mathsf{rev}(r), \mathsf{rev}(w)) = \mathsf{tape}(\mathsf{rev}(\hat{r}), \langle \mathsf{rev}(w), \mathsf{rev}(\beta) \rangle)$

Part (1) and (2) of the lemma are characteristic properties of reversals of LReg and OReg expressions respectively. Part (3) gives us a way to express $\mathsf{tape}(\triangleleft)$ in terms of $\mathsf{tape}(\triangleright)$. It is a direct consequence of (1). Part (4) follows from Part (1) and Lemma 11. This tells us that we can work with the reversal of $\mathsf{abstract}(r)$ instead of r itself, and saves us from repeatedly reversing subexpressions, which is crucial for the linear time complexity of the algorithm.

5.2 Matching Algorithm

In this section, we will describe the function eval which given $r \in \mathsf{LReg}$ and $w \in \Sigma^*$ computes $\mathsf{abstract}(r)$ and $\mathsf{oval}(r, w)$ together using oMatch from Definition 21.

The definition of the function eval shown in Figure 4, is given in terms of the auxiliary function evalAux. The function evalAux is given five arguments: w, $\overline{w}$, r, i and T. The first two arguments are the string and its reversal. We compute the reversal in advance in the function eval so that we do not repeatedly reverse the string in recursive calls of evalAux. The third argument is the expression r that we want to abstract. The fourth argument is the index i used for computing indices of the queries. The fifth argument is a list of tapes, which have already been computed. The output has three components, s, n and T. The first component s is intended to be $\mathsf{abstract}(r)$. The second component n is $\mathsf{arity}(r)$. And the last component T consists of $\mathsf{tape}(m_i, w)$ for each maximal lookarounds m_i in r. However, these tapes appear in reverse order. This is because we add each new tape to the front of the list T for the sake of efficiency. The formal connection is stated in the following lemma.

```
// Computes the leftmost longest match
Definition llMatch (r : LReg) (w : list Σ) : Option(ℕ × ℕ) :=
  let (r̂, β) = eval(r, w) in
  let t1 = oMatch(Σ* · rev(r̂), ⟨rev(w), rev(β)⟩) in
  let i' be the largest index of t1 which is 1 in
  match i' with
    None ⟹ None
    Some(i') ⟹
      let i = |w| − i' in
      let t2 = oMatch(r̂, ⟨w, β⟩[i, |w|]) in
      let d be the largest index of t2 which is 1 in
      Some([i, i + d]) // d must exist
// Computes whether w, [i, i + k] ⊨ r for all 0 ≤ k ≤ j − i
Definition match (r : LReg) (w : list Σ) (i, j : ℕ) : list 𝔹 :=
  let (r̂, β) = eval(r, w[i, j]) in
  oMatch(r̂, ⟨w, β⟩[i, j])
// Computes ⟨abstract(r), oval(r, w)⟩
Definition eval (r : LReg) (w : list Σ) : OReg × list 𝔹 :=
  let (r̂, _, T) = evalAux(w, rev(w), r, 0, []) in
  let β = transpose(rev(T)) in
  (r̂, β)
// Ensure w̄ = rev(w)
// Appends the tapes for the maximal lookarounds of r to T
Fixpoint evalAux (w, w̄ : list Σ) (r : LReg)
  (i : ℕ) (T : list list 𝔹) : OReg × ℕ × list list 𝔹 :=
  match r with
    ε ⟹ (ε, 0, T)
    σ ⟹ (σ, 0, T)
    r1 ∘ r2 where ∘ ∈ {·, +} ⟹
      let (s1, n1, T') = evalAux(w, w̄, r1, i, T) in
      let (s2, n2, T'') = evalAux(w, w̄, r2, i + n1, T') in
      (s1 ∘ s2, n1 + n2, T'')
    r* ⟹
      let (s, n, T') = evalAux(w, w̄, r, i, T) in
      (s*, n, T')
    (?<r) ⟹
      let (s, β) = eval(r, w) in
      let tape = oMatch(s, ⟨w, β⟩) in
      let q = Q+(vi) in
      (q, 1, [tape] ++ T)
    (?≮r) ⟹
      let (s, β) = eval(r, w̄) in
      let tape = rev(oMatch(s, ⟨w̄, β⟩)) in
      let q = Q+(vi) in
      (q, 1, [tape] ++ T)
    (?>r) ⟹
      let (s, β) = eval(r, w̄) in
      let tape = rev(oMatch(rev(s), ⟨w̄, rev(β)⟩)) in
      let q = Q+(vi) in
      (q, 1, [tape] ++ T)
    (?≯r) ⟹
      let (s, β) = eval(r, w) in
      let tape = oMatch(s, ⟨w, β⟩) in
      let q = Q+(vi) in
      (q, 1, [tape] ++ T)
```

Figure 4. The Matching Algorithm

Lemma 25 (Behavior of evalAux and eval). Suppose $w \in \Sigma^*$, $\overline{w} = \mathsf{rev}(w)$, $r \in \mathsf{LReg}$, $i \in \mathbb{N}$, and T is a list of sequences each of length $|w|+1$. Define $(s, n, T') = \mathsf{evalAux}(w, \overline{w}, r, i, T)$. Then, the following holds:

1. $s = \mathsf{abstract}_i(r)$ (see Definition 7).
2. $n = \mathsf{arity}(r)$, the number of maximal lookarounds in r.
3. $T' = T'' +\!\!+ T$, where T'' is a list of $|w| + 1$ length sequences added to T. There are $n = \mathsf{arity}(r)$ sequences in T''. Furthermore, T'' consists of the tape of the maximal lookarounds stacked in reverse order, i.e., $T''[n - j + 1] = \mathsf{tape}(\mathsf{maxLk}(r)[j], w)$ for all $0 \le j < n$.

As a result, if $(\hat{r}, \beta) = \mathsf{eval}(r, w)$, then, $\hat{r} = \mathsf{abstract}(r)$ and $\beta = \mathsf{oval}(r, w)$.

The proof of this theorem makes use of the correctness of oMatch, (i.e., Theorem 22) along with Lemma 11 (about the connection of oval and abstract) and Lemma 24 (about the connection of tape($\triangleright$) and tape($\triangleleft$)).

A match for r on w is a window $[i, j]$ such that $w, [i, j] \vDash r$. Call a match $[i, j]$ *maximal* if it is not subsumed by any other match $[i', j']$ where $i' \le i \le j \le j'$ with either $i' < i$ or $j < j'$. The *leftmost longest* match is a maximal match $[i, j]$ where i has the least possible value. For a given regular expression r and a word w, this match is uniquely defined. The problem of extracting the leftmost-longest match has been discussed before (see, e.g., the notes by Russ Cox [12]) and is considered the POSIX standard.

The function match passes the results of $\mathsf{eval}(r, w)$ to oMatch to check if we have a match. The computation of the function llMatch also uses the results of $\mathsf{eval}(r, w)$, and it makes two passes to identify the two end points. The following theorem follows from Lemma 25 (establishing the correctness of eval), Theorem 22 (establishing the correctness of oMatch) and Lemma 11 (establishing the connection between $\mathsf{abstract}(r)$ and $\mathsf{oval}(r, w)$).

Theorem 26 (Correctness of match, llMatch). Let $r \in \mathsf{LReg}$, $w \in \Sigma^*$ and $[i, j]$ a window in w. Then,

$$\mathsf{match}(r, w, i, j)[k] = \begin{cases} 1 & \text{if } w, [i, i + k] \vDash r \\ 0 & \text{if } w, [i, i + k] \nvDash r \end{cases}$$

for all $0 \le k \le j - i$.

If, $\mathsf{llMatch}(r, w) = \mathsf{Some}([i, j])$, then $w, [i, j] \vDash r$ and $[i, j]$ is the leftmost maximal match for r on w. Otherwise if $\mathsf{llMatch}(r, w) = \mathsf{None}$, for any window $[i, j]$ in w, $w, [i, j] \nvDash r$.

In [31], the authors show that the function match can be computed in **linear time** and requires a **linear amount of space**. This continues to hold for our implementation as well, where the main difference is that we have implemented oNFA execution using marked regular expressions. However, the resource consumption of matching using marked regular expressions is exactly the same as shown in Theorem 23.

We note some important implementation details for the function eval. Consider a regular expression r of size m, which has $O(m)$ layers of lookaheads. In the computation of eval, we would need to reverse the given string $O(m)$ times. However, we have avoided this by reversing the string in advance before passing it to evalAux. Similarly, we may have

to reverse the r itself $O(m)$ times. However, after processing each lookaround r_i of r, we work instead with an abstracted version of r in which r_i has been replaced by a query. If $|r_i| = m_i$ then, the abstracted version of r has size $m - m_i$, lowering the cost of subsequent reversals. These implementation details keep the time complexity of eval linear, instead of quadratic.

6 Experiments

We extracted our Coq scripts into Haskell code and conducted experiments on to empirically confirm the claim that our verified matching algorithm takes linear time both in terms of the size of the string and that of the expresion.

We have chosen three families of synthetic regexes (shown in PCRE notation, in Table 1) which we have called `DNLA`, `NX` and `ND`. The family `DNLA` consists of increasingly many disjuncts of negative lookaheads, while `NX` and `ND` consist of increasingly nested lookaheads. Observe that the size of the regular expressions in each family increases linearly with its parameter. For the families `NX` and `ND`, we use input strings of the form a^n; and for the family `DNLA`, we use input strings of the form $e^n \cdot abcd$. These expressions and inputs are deliberately chosen so that they would have been computationally challenging for a naive implementation (such as backtracking) of lookahead matching.

In Figure 5, the running time of the algorithm is plotted against the length of the input string. In Figure 6, the running time of the algorithm is plotted against the size of the regular expression, while keeping the input string fixed. The points shown in the figures are the average of 10 trials. In both cases, we can observe linear trends, validating our claim of linear time complexity.

Our experiments in Figure 7 demonstrate that regular expressions chosen in Table 1 were chosen so that they would be computationally challenging for naive implementations of lookaround matching. They also serve as a preliminary comparison between the performance of our extracted Haskell code, and other available tools for matching expressions with lookarounds. We have measured the performances of the expressions in Table 1 on PCRE [43], `java.util.regex` of the Java standard library [37] and the tool developed and verified in Lean presented in [45]. The Lean tool is based on Brzozowski derivatives [9], and the two industrial tools are backtracking-based. The results are shown in Figure 7. We can observe that the performance of PCRE, Java and the Lean tool are not linear in the size of the input string, in general.

Experimental Setup. The experiments were conducted on a MacBook Air M2 (8 cores) running Ventura 13.6.1 with 8GB of RAM. We used PCRE version 10.42 and Apple clang version 14.0.3 for compilation. The Lean code and proof-scripts are taken from [45] and built with Lean 4 (version 4.5.0-rc1). The runtime environment used for Java is Java SE version 20.0.2. Our extracted Haskell code is compiled with ghc version 9.4.8.

7 Related Work

The widely used regex engines which do support lookarounds are all based on backtracking, and can thus take exponential time to match in the worst-case scenario (even in the absence of lookarounds). Some of them, including PCRE and Python, do not allow unbounded lookbehinds. A notable exception is the integration of the RegElk matcher [5] with the Javascript engine V8, which is part of the Chromium browser engine. The linear time algorithm for matching regular expressions with lookarounds we have presented in this paper is based on the techniques from [31]. The key difference in our implementation is the purely functional approach, in particular the use of Marked Regular Expressions instead of explicit NFAs. The approach of using marked regular expressions to simulate NFAs was discussed in [4, 19]. Formalized versions of Marked Expressions in Isabelle/HOL are seen in [36] and [3]. A discussion on two variants for the marking of regular expressions can be found in [36]. While marked regexes have been discussed and formalized before, our paper introduces their generalization to expressions with oracle queries. This addition presents a substantial challenge in the formalization, since the change of the language $\langle\!\langle r \rangle\!\rangle$ produced when operations are applied on the marked regex r (i.e, the lemmas in Section 4.1) are much less elegant.

What we have described as o-strings in this paper (following the terminology of [31]) are widely known in the literature as *guarded strings*. They were introduced by [25] as an abstraction for program semantics. Note that such strings also arise as traces over labelled transition systems. A closely notion is that of Kleene Algebra with Tests (KAT) [28]. KAT Expressions form a Kleene Algebra with a Boolean subalgebra known as the algebra of tests. In particular, the concatenation operator behaves as conjunction on the tests, and additionally they come equipped with the notion of negation. Other than languages over guarded strings, KAT expressions can be interpreted over binary relations or semantics of programs. In fact, KAT expressions subsume propositional Hoare logic. Our Oracle Regular Expressions are a special case of KAT expressions, where the oracle queries are the tests, and the proposed syntax expresses them in negation normal form. Automata on Guarded strings, similar to the notion of Oracle-NFAs in [31], have been studied in [29]. These can be viewed as a generalization of ε-NFAs: the ε transition on NFAs are akin to a trivial test, while transitions with tests may perform multiple tests in succession without consuming additional characters. Pous [39] has formalized the theory of KAT expressions in Coq (including their completeness theorem, decision procedures for equivalence and models including guarded strings) for the purpose of reasoning about while-programs. Similar work for Schematic

Table 1. Families of Regular Expressions in Experiments

Family	$N = 1$	$N = 2$	$N = 3$	⋯
DNLA	`(?!.*a.*).*`	`((?!.*a.*)|(?!.*b.*)).*`	`(((?!.*a.*)|(?!.*b.*))|(?!.*c.*)).*`	⋯
NX	`a(?=.*c)`	`a(?=.*(?=.*c))`	`a(?=.*(?=.*(?=.*c)))`	⋯
ND	`a(?=.*b)`	`a((?=.*c)|.*a(?=.*b))`	`a((?=.*d)|(.*a((?=.*c)|.*a(?=.*b))))`	⋯

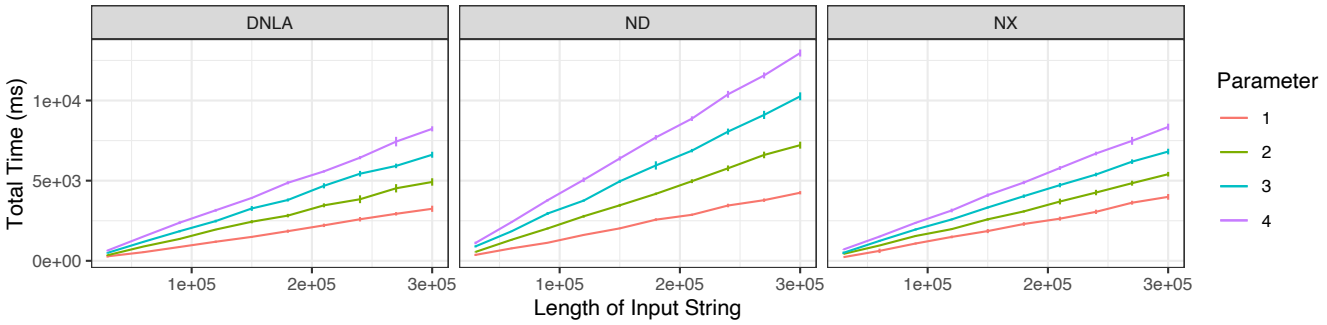

Figure 5. Performance of the matching algorithm on strings of increasing length

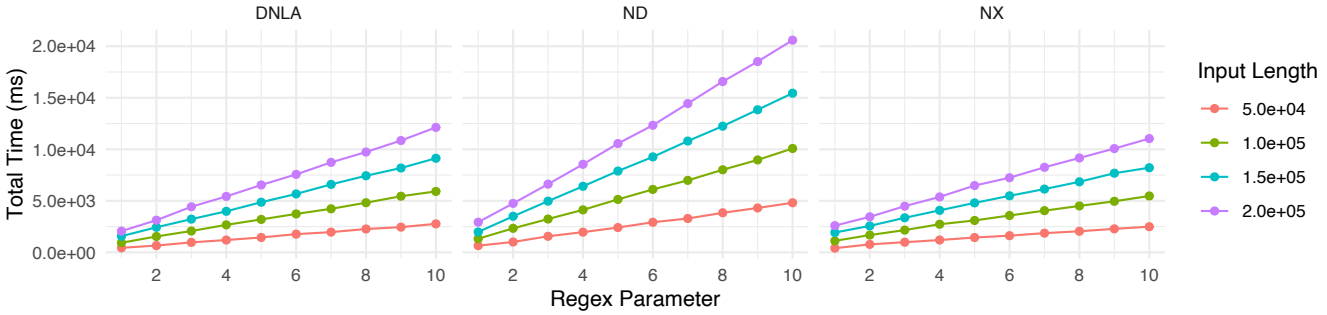

Figure 6. Performance of the matching algorithm on regular expressions of increasing size

KAT, a specific extension of KAT has been formalized in Isabelle/HOL [2].

Techniques for the linear time matching of lookaround expressions have also been published in [5] and [21]. Barriere et al [5] represent the NFAs using Virtual Machine instructions, a technique introduced by Rob Pike [38] and popularized by Russ Cox [12]. Fujinami and Hasuo [21] use a memoization approach to filling in the oracle valuations, and use a formalism based on 'NFA with sub-automata'. The backtracking-based greedy strategy of matching regexes prefers the first option r_1 in a union r_1+r_2 and prefers to lengthen the number of blocks matching r in a Kleene iteration r^*. Compatibility with the greedy strategy is a consideration in [21] and [5]. Berglund et al [7] define the semantics of lookaheads using alternating finite automata (AFA). A consequence of Berglund's approach is that the string could be matched in $O(m \cdot n)$ time by running the AFA on the string in reverse, but this approach does not directly support lookbehinds.

Zhuchko et al [45] have verified an algorithm for matching regular expressions with lookarounds in Lean based on the idea of location-based (i.e., context-dependent) derivatives from [34]. We have found that this algorithm does not run in linear time because of the unchecked growth of the size of the derivatives. Derivatives are popular for functional and verified implementations of regex matching. Coquand and Siles [11] have formalized the notion of Brzozowski derivatives [9] using Coq. Antimirov's [1] partial derivatives which correspond to states of an NFA have been verified by [26] and [33]. Stanford et al [42] have defined symbolic derivatives to deal with Boolean combinations of derivatives. This work has been extended in [34, 44] and used by Moseley et al [34] to develop a matching library in the .NET framework.

An extensive theory of regular languages, involving regular expressions, DFAs and NFAs were formalized in Coq by Doczkal et al [16, 17]. A matrix-based NFA formalization in Agda is seen in [18]. Kammar and Marek [24] have formalized a parser based on typed regular expressions in Idris. A

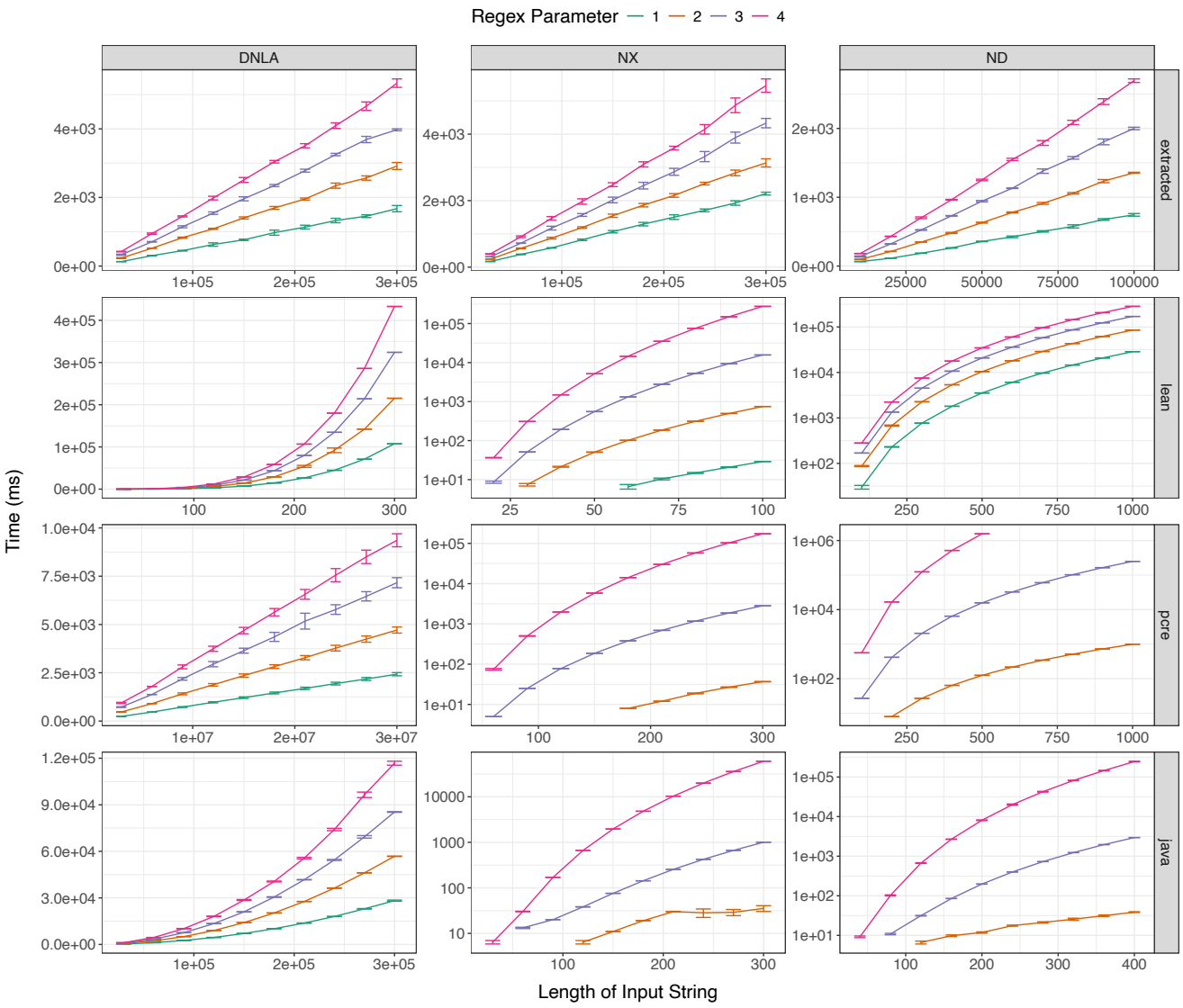

Figure 7. Performance of our extracted Haskell code (`extracted`), the Lean tool in [45] (`lean`), PCRE (`pcre`), and Java (`java`) on the three different regex families `DNLA`, `NX`, and `ND`

Coq formalization of finite state Automata is also discussed in [8] which they have verified in the context of Kleene Algebras. Recently, the semantics of Javascript regular expressions (including lookarounds) have been comprehensively mechanized in Coq [15].

8 Conclusion and Future Work

In this paper, we have presented our formalization of a linear time algorithm for matching regular expressions with lookarounds. We have also empirically verified our claims of time complexity. Our formalization is around 10k lines of Coq code (1.5k lines of definitions and 8.5k lines of proofs). The code compiles, is axiom-free, and is available at [10].

As future work, we plan to leverage our equational reasoning framework to verify the correctness of rewriting-based optimizations, such as the removal of lookarounds of bounded length. A more thorough evaluation of the performance of our extracted code would involve measurement of space usage, and comparison with additional tools such as the one developed in [5]. It would also be interesting to extend our mechanization to include algorithms for bounded repetition (e.g., [30]) and for match extraction with greedy or POSIX disambiguation (see, e.g., [20, 22, 32, 35]).

Acknowledgments

This research was supported in part by the US National Science Foundation award 2319572.

References

[1] 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

[2] Alasdair Armstrong, Georg Struth, and Tjark Weber. 2013. Program Analysis and Verification Based on Kleene Algebra in Isabelle/HOL. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–212.

[3] Andrea Asperti. 2012. A Compact Proof of Decidability for Regular Expression Equivalence. In *Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–298.

[4] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. 2010. Regular Expressions, au point. arXiv:1010.2604 [cs.FL]

[5] Aurèle Barrière and Clément Pit-Claudel. 2024. Linear Matching of JavaScript Regular Expressions. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 201 (June 2024), 25 pages. https://doi.org/10.1145/3656431

[6] 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

[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

[8] Thomas Braibant and Damien Pous. 2010. Deciding Kleene Algebras in Coq. In *ITP (LNCS, Vol. 6172)*. Springer, Edinburgh, United Kingdom, 163–178. https://doi.org/10.1007/978-3-642-14052-5_13

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

[10] Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround - Coq Source Code. https://github.com/Agnishom/lregex/.

[11] 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

[12] Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html.

[13] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027

[14] James C. Davis, Francisco Servant, and Dongyoon Lee. 2021. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, USA, 1–17. https://doi.org/10.1109/SP40001.2021.00032

[15] Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024. A Coq Mechanization of JavaScript Regular Expression Semantics. *Proceedings of the ACM on Programming Languages* ICFP (Sept. 2024), 30 pages. https://doi.org/10.48550/arXiv.2403.11919

[16] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. 2013. A Constructive Theory of Regular Languages in Coq. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham, 82–97.

[17] Christian Doczkal and Gert Smolka. 2016. Two-Way Automata in Coq. In *Interactive Theorem Proving*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer International Publishing, Cham, 151–166.

[18] Denis Firsov and Tarmo Uustalu. 2013. Certified Parsing of Regular Languages. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham, 98–113.

[19] Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A Play on Regular Expressions: Functional Pearl. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/1863543.1863594

[20] 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

[21] Hiroya Fujinami and Ichiro Hasuo. 2024. Efficient Matching with Memoization for Regexes with Look-around and Atomic Grouping. In *Programming Languages and Systems: 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part II* (Luxembourg City, Luxembourg). Springer-Verlag, Berlin, Heidelberg, 90–118. https://doi.org/10.1007/978-3-031-57267-8_4

[22] 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

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

[24] Ohad Kammar and Katarzyna Marek. 2023. Idris TyRE: a dependently typed regex parser. arXiv:2305.04480 [cs.PL]

[25] Donald M. Kaplan. 1969. Regular expressions and the equivalence of programs. *J. Comput. Syst. Sci.* 3, 4 (Nov. 1969), 361–386. https://doi.org/10.1016/S0022-0000(69)80027-9

[26] 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

[27] 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

[28] 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

[29] Dexter Kozen. 2001. *Automata on Guarded Strings and Applications*. Technical Report. USA.

[30] 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

[31] Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 92 (2024), 31 pages. https://doi.org/10.1145/3632934

[32] Konstantinos Mamouras, Alexis Le Glaunec, Wu Angela Li, and Agnishom Chattopadhyay. 2024. Static Analysis for Checking the Disambiguation Robustness of Regular Expressions. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 231 (2024), 25 pages. https://doi.org/10.1145/3656461

[33] 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

[34] 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. *Proc. ACM Program. Lang.* 7, PLDI, Article 148 (jun 2023), 24 pages. https://doi.org/10.1145/3591262

[35] 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

[36] Tobias Nipkow and Dmitriy Traytel. 2014. Unified Decision Procedures for Regular Expression Equivalence. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 450–466.

[37] Oracle. [n. d.]. Java Regex Matching. https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html. [Online; accessed Feb 28, 2024].

[38] Rob Pike. 1987. The text editor sam. *Software: Practice and Experience* 17, 11 (1987), 813–845.

[39] Damien Pous. 2013. Kleene Algebra with Tests and Coq Tools for while Programs. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 180–196.

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

[41] Matthieu Sozeau. 2023. Generalized Rewriting. https://coq.inria.fr/doc/V8.19.2/refman/addendum/generalized-rewriting.html. [Online; accessed Aug 7, 2024].

[42] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 620–635. https://doi.org/10.1145/3453483.3454066

[43] The PCRE2 Developers. 2022. Perl-compatible Regular Expressions (revised API: PCRE2). https://pcre2project.github.io/pcre2/doc/html/index.html. [Online; accessed Feb 26, 2023].

[44] Ian Erik Varatalu, Margus Veanes, and Juhan-Peep Ernits. 2023. Derivative Based Extended Regular Expression Matching Supporting Intersection, Complement and Lookarounds. arXiv:2309.14401 [cs.FL]

[45] Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. 2024. Lean Formalization of Extended Regular Expression Matching with Lookarounds. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 118–131. https://doi.org/10.1145/3636501.3636959

Received 2024-09-17; accepted 2024-11-19