École Polytechnique Fédérale de Lausanne

School of Computer and Communication Sciences

Systems and Formalisms Lab

Verification of realistic regex matching

Master's Thesis in Computer Science
(Computer Science Theory)

Author:	Marcin Wojnarowski
Supervisors:	Aurèle Barrière, Clément Pit-Claudel
Examiner:	Nate Foster
Start Date:	22.09.2025
Submission Date:	23.01.2026

Abstract

Modern regular expressions (regexes) are a powerful tool for finding patterns in text. Modern features such as capturing groups, lookarounds, and backreferences turn the problem of matching into a complex and error-prone task. Real-world implementations additionally employ a large variety of optimizations and heuristics to speed up matching in practice. In this work we don't shy away from this complexity but instead tame it by providing a formalization of realistic regex matching that includes all of the cumbersome aspects. In the Rocq proof assistant, we provide the formalization and proof of correctness of a prefix acceleration optimization in the PikeVM. We additionally formalize optimizations such as anchored regexes and impossible matches. We combine all of them to produce a realistic matching algorithm which is proven to return matches defined by the ECMAScript 2023 standard.

Preface

Before starting my thesis semester, I was in a state that caused me to have serious doubts about my ability to complete a thesis project. I was experiencing a prolonged period of time where my very privileged life somehow felt so difficult and heavy. I could see real examples of it affecting my work and causing me to have trouble delivering projects. The first tangible proof of that happened in May 2024 when I failed a semester project. From then on, it got only worse. Luckily, this thesis is something I consider to be a great success. In the last months, I saw very clear signs of my old self returning, the me that is able to focus on the work that I so enjoy. It feels good to feel good.

I cannot take much credit for arriving to this point. I credit it to the people that I have been so lucky to be surrounded with. I first want to thank Viktor and Simon for being very understanding of my situation when I failed the semester project. I want to thank all of the people that were by my side when I needed them the most, notably I want to thank Jakob, Grégoire, Biers, Victor, Derya, and most importantly my mom. I want to thank Aurèle and Clément for giving me this opportunity, for the guidance, and for being incredible people to work with. Finally, I want to thank Johanna for your invaluable friendship and for allowing me to be happy again.

All of the documentation for this thesis (including this report) is available at

https://github.com/shilangyu/pikevm-systemf

Rocq formalization	**Parser for frequency analysis**	**Rust code used for experiments**	**Benchmarks for regex matching**
https://github.com/LindenRegex/Linden/tree/104d33313a5a2360beec9e254551257eedbfc1d9	https://github.com/LindenRegex/RegElk/tree/f154c76f7fc55e67555e53451cca71d6e644ef44	https://github.com/LindenRegex/rust-regex/tree/d2c210eec94405630e3e7ebba6d6d1b19e5dc85f	https://github.com/LindenRegex/rebar/tree/72213a4ce5ff4a804ca7a542e665c4c81f2dbeed

Contents

The Report is temporarily in construction while I migrate it to the new Linden codebase.

1 Introduction

Classical regular expressions (regexes)[1] have been studied for decades and are sometimes believed to be well understood. However, modern regular expressions used in programming languages have long strayed away from just describing regular languages. This is due to the addition of many modern regex features such as *backreferences*, *capture groups*, or *lookarounds*. While classically we were interested in the question of whether a regex accepts or rejects a word, modern regexes concern themselves with whether they matched a word, where they matched it, and how they matched it. Layering these new questions on top of the various modern regex features turn the matching problem into a much more difficult one. Combined with the need for regex matching to be fast in practice, real-world implementations are often buggy [Reg25, And21] and vulnerable to attacks [Joh19, SP18].

Real-world regex matching implementations are fast thanks to the large variety of optimizations and heuristics they employ. They skip parts of the haystack[2], they rewrite the regex into an equivalent one that is more efficiently matched, they combine multiple string search algorithms, they specialize commonly used patterns, they change strategies depending on memory usage, and much more. Unfortunately, this growing complexity and the resulting subtle interactions between those optimizations lead to implementations that are difficult to reason about. Formal verification gives us tools to address these issues by having a structured approach to reasoning. We can produce implementations that are provably correct (absence of bugs) and provably efficient (absence of performance vulnerabilities). Unfortunately, the real-world optimized algorithms full of heuristics go beyond what has been done in the state-of-the-art verified regex matching.

[1] A *classical* regex is a pattern describing a regular language. A *modern* regex is a pattern used in programming languages that may include features going beyond regular languages, such as backreferences and lookarounds. It is used to find matches and to extract substrings from text.

[2] In the context of regex **matching**, the haystack is the input text in which we search for occurrences of patterns defined by regular expressions. When we say that we want to match the regex `/a*b{3}/` against "abc", the string "abc" is the haystack. Newlines in the haystack are represented with the "⏎" character. Already seen characters in the haystack are underlined "qwerty". Positions in the haystack are marked with an arrow "qwer↑ty". Match ranges in the haystack are highlighted "qwerty".

The goal of this work is to formally verify **real-world regex optimizations**. Since we do not wish to verify a toy regex matching algorithm, we will use the *Linden* [BDP26] project. *Linden* provides a **complete** and **practical** mechanization of regex matching in the Rocq prover. It formalizes both the semantics of matching in addition to formalizing verified matching algorithms following these semantics for a subset of regexes. Linden's semantics have been proven to be equivalent to Warblre [DBP24], a faithful line-by-line translation of the ECMAScript 2023 [ECM23] chapter on regexes into a Rocq mechanization. Using Linden as the base gives us confidence that we are working with real-world regexes which include all of the ugly and cumbersome aspects. Building on this project we proved the correctness of the **prefix acceleration** optimization which can potentially skip very large portions of the haystack, **anchored searches** optimization which detects that a match can only appear at the start of the haystack, and **impossible matches** optimization which detects that a regex can never match any haystack.

In practice real-world implementations utilize multiple regex matching algorithms (engines)[3] to specialize searches for given regexes and haystacks. The choice of an engine is guided by heuristics. These heuristics use information such as which regex features were used and the length of the haystack. To protect ourselves from Regular expression Denial of Service (ReDoS)[4] attacks, we focus on engines that have worst-case linear execution with respect to the regex and haystack size. *Linden* formalizes two such engines, the PikeVM [BDP26] and the memoized backtracker. In this work we define and prove correctness of a **Meta engine** which deploys heuristics and optimizations using those base linear engines to considerably speed up matching.

The main contributions of this work are:

1. Formalization of **literal extraction** Chapter 3, a static analysis of constant strings in a regex
2. Formalization of **substring search** Section 3.2, a generic description of substring search algorithms
3. Formal proof of **prefix acceleration** Chapter 4 in the PikeVM engine, an optimization that uses literal extraction and substring search to skip parts of the haystack

[3] An algorithm used to perform matching of a regex against a haystack. It supports a specific subset of regex features and has some performance characteristics. Examples include the PikeVM, LazyDFA, Backtracking.

[4] An exploit of unfavorable regex matching performance characteristics. When a regex comes from user input, it can be used to attack by crafting a regex which makes matching take exponential time. This most commonly affects backtracking engines which have worst-time exponential runtime.

4. Formalization of **anchored regex** Section 5.3 and **impossible matches** Section 5.2.2 optimizations
5. Formalization of a **Meta engine** Chapter 5 that combines multiple linear engines with heuristics and optimizations to speed up matching in practice

All of the results are mechanized in the Rocq proof assistant and operate on real-world regexes as defined by the ECMAScript 2023 standard.

2 Background

2.1 Regexes

r	::=	ε	Epsilon	cd	::=	c	Single character
	|	cd	Character descriptor		|	$[c_1{-}c_2]$	Range
	|	$r_1 r_2$	Sequence		|	$[cd_1 cd_2]$	Union
	|	$r_1 \vert r_2$	Disjunction		|	$\cdot$	Dot
	|	$(_g r)$	Capturing group		|	\w | \W | \d	
	|	$\langle r \rangle$	Non-capturing group			| \D | \s | \S	Character classes
	|	$\backslash g$	Backreference			| \p{property}	
	|	$r\{min, \Delta, \gamma\}$	Quantifier			| \P{property}	
	|	a	Anchor		|	$[\hat{}cd]$	Inversion
	|	$(?lk\ r)$	Lookaround		|	$[\hat{}]$	All
γ	::=	$\top$	Greedy		|	$[]$	Empty
	|	$\bot$	Lazy				
				a	::=	$\hat{}$	Start
lk	::=	$=$	Positive lookahead		|	\$	End
	|	!	Negative lookahead		|	\b	Word boundary
	|	<=	Positive lookbehind		|	\B	Non-word boundary
	|	<!	Negative lookbehind				

Figure 1: Abstract syntax of Linden regexes

The abstract syntax of Linden regexes is shown in Figure 1. This syntax slightly differs from the syntax of ECMAScript regexes, but regardless every ECMAScript regex can be expressed using the abstract regex syntax of Linden. We now outline the most important aspects of the syntax and underline the differences between the Linden and ECMAScript representation.

Epsilon ε. The regex that matches the empty string. The ECMAScript regex `/|abc/` is equivalent to $\varepsilon|$abc in Linden.

Capturing group $(_g r)$. A capturing group (capture)[5] numbered by g that captures the substring matched by the regex r. In ECMAScript captures are not parametrized by their number. Instead, captures can be either named `/(?<name>r)/`, or left unnumbered `/(r)/`. We can, however, easily rewrite those captures into numbered ones by incrementally assigning a number starting from 1 for each opening parenthesis of captures from left to right.

[5] A feature in modern regexes that allows parts of the matched text to be captured by a subpatterns and extracted later. They are annotated using parentheses. For instance, given `/(a(bc))e/`, there are two capture groups: the outer group captures , and the inner group captures . When this regex matches the string "abce", the first capture group will contain "abc" and the second will contain "bc".

The ECMAScript regex `/e(a(?<in>bc))(f)/` is equivalent to $e({}_1\mathrm{a}({}_2\mathrm{bc}))({}_3\mathrm{f})$ in Linden.

Non-capturing group $\langle r \rangle$**.** A grouping construct that does not create a capture. The ECMAScript regex `/(?:abc)/` is equivalent to $\langle \mathrm{abc} \rangle$ in Linden.

Quantifier $r\{min, \Delta, \gamma\}$**.** Specifies that the regex r is to be repeated at least $min \in \mathbb{N}$ times, and at most $min + \Delta$ times. If Δ is ∞, then there is no upper bound on the number of repetitions. The parameter γ specifies whether the quantifier is greedy ($\top$) which means it should repeat r as many times as possible or lazy ($\bot$) which means it should repeat r as few times as possible. For instance, the ECMAScript regex `/a{2,5}b?c*e+?/` is equivalent to $a\{2, 3, \top\}b\{0, 1, \top\}c\{0, \infty, \top\}e\{1, \infty, \bot\}$ in Linden.

Character descriptor cd**.** Describes sets of characters. If they match something in the haystack, they will always match exactly one character. They can be single literal characters, ranges of characters, unions of character descriptors, the dot (which matches any character except line terminators), character classes (such as word characters, digits, whitespace, Unicode properties), inversion of character descriptors, the set of all characters, or the empty set (it never matches). For instance, the ECMAScript regex `/[a-c0-7]\d./` is equivalent to $[\mathrm{a{-}c}[0{-}7]]\backslash \mathrm{d} \cdot$ in Linden. For brevity, we will flatten descriptors in $[\,]$, giving us $[\mathrm{a{-}c0{-}7}]$ instead of $[\mathrm{a{-}c}[0{-}7]]$

Anchor a**.** Specifies a position in the haystack rather than characters. It is merely an assertion, it does not consume characters. `^` asserts that we are at the start of the haystack[6] , and `$` asserts that we are at the end of the haystack[6]. `\b` asserts that we are at a word boundary and `\B` that we are not. A word boundary is a position in the haystack where exactly one of the two adjacent characters is a word character and the other character is not, for example `/\ba\Bf \bc/` matches "af c". The ECMAScript regex `/^abc$/` is equivalent to $\hat{}\mathrm{abc}\$$ in Linden.

[6] Behavior can be altered by a flag, see Section 2.1 Flags.

Lookaround $(?lk\ r)$**.** A zero-width assertion that r matches or not at the current position. A lookahead (lk is $=$ or !) asserts something about the text following the current position, while a lookbehind (lk is $<=$ or $<!$) asserts something about the text preceding the current position. Positive lookarounds (lk is $=$ or $<=$) assert that r has to match, negative lookarounds (lk is ! or

<!) that it has to not match. The ECMAScript regex `/(?=abc)mno(?<!xyz)/` is equivalent to (?= abc)mno(?<! xyz) in Linden.

Discussion of backreferences is omitted as backreference matching is NP-hard [AM99] and here we focus on engines that run in worst-case linear time.

Throughout the report we will tend to use the ECMAScript syntax for regexes for brevity and familiarity, ie. `/^this+|syntax*(?=!!!)/`. The theorems will, however, be stated in terms of the Linden abstract regex syntax.

2.1.1 Regex size

Throughout this work we will often refer to the size of a regex. Most importantly, the size of a regex plays a role in defining complexity characteristics. We therefore define the precise notion of the size of a regex by Listing 1. From now on, whenever talking about the *"regex size"* or $|r|$, this is the precise definition we are referring to. As seen, the regex size is equal to the size of the unfolded regex, i.e. one in which quantifier repetitions are unfolded. This does mean that the regex size can be exponentially larger than the size of the textual representation. This can be seen in the example of nested quantification `/((a{5}){5}){5}/`. By just wrapping any regex r with `/(r){n}/` for some number n, we increased its textual size by just $4 + \log_{10} n$ while the regex size increased by a factor of n.

Semantics/Regex.v#73-89

```
Fixpoint regex_size (r: regex) : nat :=
  match r with
  | Epsilon => 1
  | Regex.Character _ => 1
  | Disjunction r1 r2 => 1 + regex_size r1 + regex_size r2
  | Sequence r1 r2 => 1 + regex_size r1 + regex_size r2
  | Quantified _ min delta r1 =>
      1 + min * regex_size r1 +
        (match delta with
        | NoI.Inf => regex_size r1
        | NoI.N n => n * regex_size r1
        end)
  | Lookaround _ r1 => 1 + regex_size r1
  | Group _ r1 => 1 + regex_size r1
  | Anchor _ => 1
  | Backreference _ => 1
  end
```

Listing 1: Regex size definition.

This unfolded regex definition of the size is used to bound the engine executions, together with the haystack size which is just the length of the input string.

2.1.2 Matching semantics

We follow the discussion of regexes by outlining the relevant details of the matching semantics. The semantics on which we base our results are known by several names: **backtracking**-, **PCRE**-, or **leftmost-greedy**- semantics. Many regex implementations in standard libraries of programming languages follow these semantics. Examples include Python, Rust, Java, Golang, and of course, ECMAScript[7]. We describe the important aspects of these semantics below.

[7] ... and .NET, Perl, PHP, Dart, Ruby.[8]

[8] ... aaand C++, Raku, Julia, D, Erlang

Priority. A regex could be matched in multiple ways. Consider the regex `/(aa|a)c+/` on the haystack "aaccc". We could imagine the following results of matching:

- "aaccc", `aa` chosen in the disjunction and `+` matches `c` three times
- "aaccc", `a` chosen in the disjunction and `+` matches `c` three times
- "aaccc", `aa` chosen in the disjunction and `+` matches `c` once

However, in our semantics every potential choice during matching has priority assigned to its branches. For disjunction, the alternative that is syntactically written first has priority over the second alternative. In our example that means that the `aa` alternative must be considered for the match. For quantifiers, the priority is given to the branch that tries to match the repeated regex again. The branch that wants to finish repetitions has lower priority. When using the lazy specifier ($\perp$) on quantifiers, the priority is inverted. In our example that means that the `c+` construct must try to match `c` as many times as possible. Hence, only the first listed option ("aaccc") is a valid match according to our semantics.

Leftmost matches. When looking for a pattern anywhere in the haystack, multiple matches may exist. In such cases, the match which is of relevance to us is the first match, called the leftmost match. For instance, the match of `/a+(c|d)/` in the haystack "xxaadyyaac" is "xxaadyyaac", not "xxaadyyaac".

Backtracking. During matching when making decisions on which branch to take (disjunction, quantifiers) we follow the priority and leftmost principles outlined above. However, if during matching our choices have led us to a point where no match is found, we backtrack to the last decision and retry with the lower-priority choice. This backtracking is performed until a match is found

or all choices are exhausted. For instance, when matching `/a+ab/` on "aaaab", the initial choice of trying to match all "a"s with `a+` would not lead to a match because one more following "a" required by the regex would not be found. So we backtrack and try again by making `+` match one less "a", which would lead to a successful match.

Flags. Matching of a regex can be configured by a handful of boolean flags. These modify the semantics of matching in small but sometimes significant ways. In ECMAScript syntax, these flags appear after the closing `/`. Each flag is represented by a single character. Its presence means that the flag is **enabled**, otherwise it is **disabled**. For instance, the regex `/a*c/im` has the flags `i` and `m` enabled, but all others disabled. Below we mention three of the flags which are of importance for this work.

- **IgnoreCase**, `i` – when true, the matching is case-insensitive. That means the regex `/aB/i` matches the strings "ab", "aB", "Ab", and "AB", while `/aB/` matches only "aB".
- **Multiline**, `m` – when true, the anchors `^` and `$` additionally match respectively the start and the end of a line. That means the regex `/^abc$/m` matches the string "xyz⏎abc⏎mno", while `/^abc$/` does not.
- **DotAll**, `s` – when true, the dot character descriptor matches line terminators as well. That means the regex `/a.c/s` matches the string "a⏎c", while `/a.c/` does not.

In Linden, those flags are stored in the `RegExpRecord` record type. An instance of this type will be implicitly available as the variable `rer`.

2.2 Backtracking trees

The precise semantics on which we operate are defined by *backtracking trees*. A backtracking tree describes all possible ways one can match a regex against a concrete haystack during backtracking matching if we were to not stop as soon as we found a match. The nodes of this tree represent choices, assertions, setting values of the captures, consumption of a character. Two special nodes `Match` and `Mismatch` indicate that this exploration path has led to a match and mismatch respectively. These nodes can only ever be leaves of the tree. Once we materialize a tree for some regex and haystack, we say that the result of matching is the path from the root of the tree to the leftmost `Match` leaf. If no such leaf exists, the matching has no result.

Haystacks in Linden are represented by the `input` type[9] and additionally encode the current position within it. This is done by storing the characters that are ahead of us as well as the reverse of the characters before us. For example, if we wish to write the positioned haystack "abc↑ttt" as the `input` type, it would be `Input "tt" "tcba"`.

To track the values of captures, Linden stores them in a `group_map`[10]. A `group_map` is a mapping from capture indices to optional pairs of positions within the haystack. A pair of positions indicates the start and end of the substring captured by the corresponding capture. An unset capture is represented by `None`.

In Linden, the backtracking tree is represented by the `tree`[11] inductive. To create a backtracking tree for some regex and haystack, we use the `is_tree` relation[12]. Finally, to traverse the tree to find the leftmost match, we use the `tree_res`[13] (sometimes abstracted to the wrapper function `first_leaf`[14]). These traversing functions return an optional `leaf` type[15], which is simply a new `input` indicating where the match ends and the resulting `group_map`.

Theorems will be stated in terms of these definitions, thus we dedicate the next paragraphs to illustrating each with an example.

`is_tree [Areg r] inp gm forward tr` states that `tr` is the backtracking tree for the regex `r` and haystack `inp` with an initial group map `gm` (all captures unset), in the forward direction. The direction is needed to encode lookbehinds. It is safe to ignore this parameter in our definitions, as we will not be dealing with them. The interesting looking `[Areg r]` is a singleton list of a so called `action`[16]. To relate a regex and input to a tree, the `is_tree` relation operates on a list of actions. The `Areg r` action instructs the `is_tree` relation to construct a tree for the regex `r`. Such an action can potentially produce two new actions. Consider the regex `Sequence r1 r2`. The `tree_sequence` rule of the `is_tree` inductive will produce two new actions to handle each regex one after the other: `[Areg r1, Areg r2]`[17]. The remaining kinds of actions are not relevant to our discussion, hence we omit them.

Now that we have `tr`, we can extract the *`option`* `leaf` from it using `tree_res tr GroupMap.empty inp forward` or equivalently `first_leaf tr inp`. The group map we provide is the initial value for the traversal. During it,

[9] Defined in Semantics/Chars.v#17-18

[10] Defined in Semantics/Groups.v#116

[11] Defined in Semantics/Tree.v#68-79

[12] Defined in Semantics/Semantics.v#161-264

[13] Defined in Semantics/Tree.v#145-172

[14] Defined in Semantics/Tree.v#178-179

[15] Defined in Semantics/Tree.v#129

[16] Defined in Semantics/Semantics.v#146-149

[17] If the direction is `backwards`, the order of these two actions would be swapped

the group map will be populated with the values of the captures as they are encountered in the tree.

We conclude this discussion by showing two concrete rules of the `is_tree` relation to give better intuition about how trees are constructed. We omit the group map and the direction. Consider the rule for the `+` and for a successful character descriptor match:

$$\frac{\texttt{is\_tree (Areg r1 :: Areg (Quantified greedy 0 Inf r1) :: cont) inp titer}}{\texttt{is\_tree (Areg (Quantified greedy (S 0) Inf r1) :: cont) inp titer}}\ \textsc{Plus}$$

$$\frac{\texttt{read\_char cd inp = Some (c, inp')} \quad \texttt{is\_tree cont inp' tcont}}{\texttt{is\_tree (Areg (Character cd) :: cont) inp (Read c tcont)}}\ \textsc{Star}$$

3 Literal extraction

One characteristic of regexes which is often exploited for optimization in practice is the knowledge that certain parts of the regex match literals (constant strings). Take for example the regex `/abc\w+/`. We can see that any match this regex can produce must contain the constant string "abc". So if a haystack does not contain "abc", we can immediately conclude no match can be found. To speed up looking for matches one can also just look for matches in the neighborhoods of occurrences of "abc" in the haystack. Unfortunately, "*searching in the neighborhoods*" is too general of a notion to be useful in practice. Consider `/.*abc\w+/`. We still know that any match must contain "abc", but what is the neighborhood of "abc" in this case? Because of the `/.*/` preceding the "abc", **everything** before "abc" can potentially be part of the match. This means we lost any benefit gained from knowing that there is this constant string there.

Let's go back to the original example of `/abc\w+/`. A more precise information which we can extract here is that not only does any match has to contain "abc", but any match **must start** with "abc". This stronger property can be used more directly: first we look for occurrences of "abc" and at this position exactly we try to match the regex. If we fail, we can simply move on to the next occurrence. This family of optimizations where prefixes are leveraged to skip parts of the haystack is commonly referred to as "prefix acceleration" and is deployed in many real-world regex engines. Informed by the frequency at which such prefix constant strings appear in practice (Section 6.1) and by the very large speedups in matching they can provide (Section 6.2), we conjecture

that this is the single **most important optimization** for regex matching in practice. Hence, the prefix acceleration optimization is the primary focus of this work.

To get the proof of correctness for the prefix acceleration optimization we must first formalize this constant-strings-in-regex analysis, called *literal extraction* (Section 3.1). Then we formalize the notion of looking for those literals in a haystack (Section 3.2). These formalizations allows us to state and prove the correctness properties of literal extraction (Section 3.3). And finally, we tackle the prefix acceleration proof together with the peculiarities of the PikeVM engine (Chapter 4).

3.1 Literals

The literal extraction phase is a form of static analysis of the syntactic structure of the regex. As such, various implementations of the analysis could produce varying degrees of preciseness in the result. For instance, we said that for the example of `/abc\w+/` we can extracted a prefix of "abc". But a less precise yet still correct analysis could only extract the prefix "a". In this work we implement an analysis resembling abstract interpretation that tries to extract as much information as possible. When giving examples of what an extracted literal of some regex could be, we always mean the most precise literal that our analysis can extract.

```
18 Variant literal : Type :=
     (* the entire match is exactly `s` *)
   | Exact (s : string)
     (* the match starts with `s` *)
   | Prefix (s : string)
     (* this indicates a match cannot exist, as opposed to Prefix [] which means we do not know
     anything about the match *)
   | Impossible

19 Notation Nothing := (Exact [])

20 Notation Unknown := (Prefix [])
```

18 Engine/Prefix.v#389-395

19 Engine/Prefix.v#397

20 Engine/Prefix.v#398

Listing 2: Literal definition together with two aliases. The *string* type is a list of characters.

A literal as defined in Listing 2 is what we extract from a regex r. It is defined by its three constructors:

1. `Exact s` – any match of r is exactly the string s. For instance, the literal of `/p{3}(a|a)c/` is `Exact "pppac"`,
2. `Prefix s` – any match of r starts with the string s. For instance, the literal of `/abc\w+/` is `Prefix "abc"`,

3. `Impossible` – the regex r can never match. For instance, the literal of `/[]/` (empty character class) is `Impossible`.

Additionally we define two aliases, `Nothing` (standing for `Exact ""`) which means r matches exactly the empty string like the empty regex `//`, and `Unknown` (standing for `Prefix ""`) which means we cannot tell anything useful about the matches of r like for the regex `/.*abc\w+/`.

```
Definition prefix (l : literal) :=
  match l with
  | Exact s => s
  | Prefix s => s
  | Impossible => []
  end
```

Engine/Prefix.v#406-411

Listing 3: Literal weakening defintion into just the prefix information.

We will often want to weaken a literal into just the information of what prefix it represents. The definition is given in Listing 3.

```
Definition chain_literals (l1 l2 : literal) : literal :=
  match l1 with
  | Exact s1 => match l2 with
    | Exact s2 => Exact (s1 ++ s2)
    | Prefix s2 => Prefix (s1 ++ s2)
    | Impossible => Impossible
  end
  | Prefix s1 => match l2 with
    | Impossible => Impossible
    | _ => Prefix s1
  end
  | Impossible => Impossible
  end
```

Engine/Prefix.v#414-426

```
Fixpoint common_prefix (s1 s2 : string) : string :=
  match s1, s2 with
  | h1 :: t1, h2 :: t2 => if h1 == h2 then h1 :: common_prefix t1 t2 else []
  | _, _ => []
  end
```

Engine/Prefix.v#464-468

```
Definition merge_literals (l1 l2 : literal) : literal :=
  match l1, l2 with
  | Impossible, l2 => l2
  | l1, Impossible => l1
  | l1, l2 =>
    if l1 == l2 then l1 else Prefix (common_prefix (prefix l1) (prefix l2))
  end
```

Engine/Prefix.v#471-477

Listing 4: Literal chaining and merging definitions.

We must additionally define two important operators on literals. The first one is *chaining* defined in Listing 4. It computes the literal resulting from joining two immediately consecutive literals, most commonly from a sequence. If any of the inputs is `Impossible`, the result is also `Impossible`. The intuition is that if one of the literals says that no match can exist, then adding anything to

the left or right of that does not change this fact. Naturally, chaining is not commutative, it is akin to string concatenation. If the left operand is `Prefix`, then the right operand cannot be used because there might be some unknown characters in between. The second operator is *merging* defined in Listing 4. It computes the result of joining two overlapping literals, most commonly from a disjunction. If either of the literals is an `Impossible`, then the result is that equal to the other literal. If the literals are the same, the result is that literal. Otherwise we must degrade into a `Prefix` with the string that is the longest common prefix of both. We can see that in both operators `Exact` literals are quite brittle. To get an `Exact` literal out of the operators, a `Prefix` literal cannot be present. This is expected, as `Exact` literals encode a very strong property about matches.

What follows are a couple of useful lemmas about those operators.

Lemma (chain_literals_assoc) : Engine/Prefix.v#434-436

```
forall l1 l2 l3,
  chain_literals l1 (chain_literals l2 l3) = chain_literals (chain_literals l1 l2) l3
```

Lemma (chain_literals_impossible) : Engine/Prefix.v#442-444

```
forall l1 l2,
  chain_literals l1 l2 = Impossible <-> (l1 = Impossible \/ l2 = Impossible)
```

Lemma (merge_literals_comm) : Engine/Prefix.v#512-514

```
forall l1 l2,
  merge_literals l1 l2 = merge_literals l2 l1
```

Lemma (merge_literals_impossible) : Engine/Prefix.v#520-522

```
forall l1 l2,
  merge_literals l1 l2 = Impossible <-> (l1 = Impossible /\ l2 = Impossible)
```

We need one more definition before we can fully define literal extraction. Listing 5 defines the operation of repeatedly chaining a literal l for n times with a base case of a literal *base* which will be used at the very end of the chaining. For example, `repeat_literal (Exact "a") (Prefix "b") 3` reduces to

```
⇝* chain_literals (Exact "a") (chain_literals (Exact "a") (chain_literals (Exact "a") (Prefix "b")))
⇝ chain_literals (Exact "a") (chain_literals (Exact "a") (Prefix "ab"))
⇝ chain_literals (Exact "a") (Prefix "aab")
⇝ Prefix "aaab"
```

Engine/Prefix.v#428-432

```
Fixpoint repeat_literal (l: literal) (base: literal) (n: nat) : literal :=
  match n with
  | 0 => base
  | S n' => chain_literals l (repeat_literal l base n')
  end
```

Listing 5: Repeated chaining of a literal.

With this setup in place, we can now define the literal extraction function as seen in Listing 6. A common pattern appearing here is that if the thing we are analyzing can potentially match two different characters, we cannot extract any certain literal and so we return with an `Unknown`. This is the case with character classes like `/\d/` (`CdDigits`), which can match as many as ten characters. The only case which gives us the `Impossible` literal is the empty character class `/[]/`, which can never match any character[24]. `Exact` literals are produced one character at the time from character descriptors; the single constant character like `/c/`, ranges over one character like `/[c-c]/`, and unions of character descriptors which all match the same character like `/[cc-c]/`. Whenever we want combine literals from different parts of the regex, we either want to chain them (`Sequence`, `Quantified`) or merge them (`Disjunction`, `CdUnion`).

[24] This analysis could be extended to find more `Impossible` cases, such as `/(?=b)a/`. This however can be instead handled by a regex-equivalence rewrite pass, which would turn that regex into `/[]/` and then would be detected by the literal extraction.

Depending on Δ, we must distinguish between the cases of quantifiers. If $\Delta = 0$, this quantification has a fixed number of repetitions, so we can analyze it just as if the quantified regex was repeated that many times. This will allow us to extract `Exact` literals from regexes like `/a{4}/`. If $\Delta > 0$, we can at best extract a `Prefix` literal, as the number of repetitions is not fixed. This will allow us to extract `Prefix "aaaa"` from `/a{4,5}/`. Computing this literal is done using Listing 5 with the repetition count equal to *min* and the base case differing based on whether Δ is zero or not.

```
Fixpoint extract_literal_char (cd: char_descr) : literal :=
  match cd with
  | CdEmpty => Impossible
  | CdSingle c => Exact [c]
  | CdRange l h => if l == h then Exact [l] else Unknown
  | CdUnion cd1 cd2 => merge_literals (extract_literal_char cd1) (extract_literal_char cd2)
  | CdDot | CdAll | CdDigits | CdNonDigits | CdWhitespace | CdNonWhitespace | CdWordChar
  | CdNonWordChar | CdUnicodeProp _ | CdNonUnicodeProp _ | CdInv _ => Unknown
  end
```
Engine/Prefix.v#545-553

```
Fixpoint extract_literal (r: regex) : literal :=
  if RegExpRecord.ignoreCase rer then Unknown else
  match r with
  | Epsilon => Nothing
  | Regex.Character cd => extract_literal_char cd
  | Disjunction r1 r2 => merge_literals (extract_literal r1) (extract_literal r2)
  | Sequence r1 r2 => chain_literals (extract_literal r1) (extract_literal r2)
  | Quantified _ min (NoI.N 0) r1 => repeat_literal (extract_literal r1) Nothing min
  | Quantified _ min _ r1 => repeat_literal (extract_literal r1) Unknown min
  | Lookaround _ r1 => Unknown
  | Group _ r1 => extract_literal r1
  | Anchor _ => Nothing
  | Backreference _ => Unknown
  end
```
Engine/Prefix.v#563-576

Listing 6: Literal extraction from a character descriptor and from an entire regex.

The entire extraction must be guarded on the `ignoreCase` flag (see Section 2.1 Flags). If this flag is set, we cannot extract any certain literals from the regex, rendering this optimization useless. This weakness can be addressed by extracting multiple literals per regex rather than just one. More on this improvement can be found in Section 7.2.

One could extract literals from backreferences by expanding them into the literal extracted from the referenced capture. Then, the literal of `/(abc)\1/` would be `Exact "abcabc"`. However, since in this work we are interested in linear time regex engines, and linear engines do not support backreferences, we do not implement this feature. With the help of a few lemmas,

Lemma (chain_literals_length) :
```
  forall l1 l2,
    length (prefix (chain_literals l1 l2)) <= length (prefix l1) + length (prefix l2)
```
Engine/Prefix.v#1177-1179

Lemma (repeat_literal_length) :
```
  forall l base n,
    length (prefix (repeat_literal l base n)) <=
      n * length (prefix l) + length (prefix base)
```
Engine/Prefix.v#1185-1188

Lemma (merge_literals_length) :
```
  forall l1 l2,
    length (prefix (merge_literals l1 l2)) <= Nat.max (length (prefix l1)) (length
(prefix l2))
```
Engine/Prefix.v#1206-1208

we show that the length of the extracted literal is bounded by the size of the regex.

THEOREM (extract_literal_size_bound) : Engine/Prefix.v#1218-1220

```
forall r,
  length (prefix (extract_literal r)) <= regex_size r
```

Proof. Induction on r, using Lemma chain_literals_length, Lemma repeat_literal_length, and Lemma merge_literals_length where applicable.

3.2 Substring search

Substring search is the problem of finding occurrences of a given substring ss in a larger string s. This is a well studied problem in computer science, one for which many efficient algorithms exist. Some of the most well known ones like the Rabin-Karp [KR87] algorithm runs on average in $O(|s|)$, at worst in $O(|ss| \cdot |s|)$. Despite having the same complexity as our linear regex engines, in practice substring search algorithms are much faster and simpler. This makes sense; substring searching is a subproblem of regex matching, and so we can expect specialized algorithms to outperform general ones. Nowadays, highly optimized substring search implementations use Single Instruction, Multiple Data (SIMD)[27] further speeding up the search. While SIMD-accelerated regex engines exist [Wan+19], they are much more complex and implement different semantics than the ones we are interested in. Due to substring searches being much faster than general regex engines, we want to leverage them whenever possible in our engine. To that end we will leverage substring searches in prefix acceleration. The correctness of substring searches will yield the correctness of prefix acceleration. Thus, we must first formalize substring searches. We are careful to define a generic description of substring search algorithms which fits most of the existing algorithms. This way, our formalization can be applied to any of them.

[27] A parallel computing paradigm where a single instruction operates on multiple data points simultaneously. Modern CPUs often support SIMD instructions that can process multiple pieces of data in parallel, speeding up operations by a healthy constant factor.

To get started we describe what it means for a string to start with a different string. For this we define the `starts_with` inductive seen in Listing 7. We say that "ss is the prefix of s" to mean that s starts with ss, or more precisely `starts_with ss s`. The definition has a base case stating that an empty string is the prefix of any string, and an inductive case stating that if both strings start with the same character and the rest of the first string is the prefix of the rest of the second string, then the first string is the prefix

of the second string. One can easily see that this definition is decidable[28] and that it defines a preorder[29] on strings[30].

```
Inductive starts_with: string -> string -> Prop :=
| sw_nil: forall s, starts_with [] s
| sw_cons: forall h t1 t2, starts_with t1 t2 -> starts_with (h :: t1) (h :: t2)
```

Then, we can describe substring search procedures. For this we define a typeclass seen in Listing 8. Each instance of this typeclass must provide an implementation of a substring search algorithm together with proofs of three axioms that this search must satisfy. On the high-level, the search function takes two strings, the substring to search ss and the haystack s, and returns the earliest position in s where ss appears. If no such position exists, it returns `None`. This is captured by the three axioms:

```
Class StrSearch := {
  str_search : string -> string -> option nat;

  (* the found position starts with the searched substring *)
  starts_with_ss: forall s ss i,
    str_search ss s = Some i ->
    starts_with ss (List.skipn i s);
  (* there is no earlier position that starts with the searched substring *)
  no_earlier: forall s ss i,
    str_search ss s = Some i ->
    forall i', i' < i -> ~ (starts_with ss (List.skipn i' s));
  (* if the substring is not found, it cannot appear at any position of the haystack *)
  not_found: forall s ss,
    str_search ss s = None ->
    forall i, i <= length s -> ~ (starts_with ss (List.skipn i s))
}
```

1. `starts_with_ss` asserts that if the search returns some position, then indeed at that position the haystack starts with the substring,
2. `no_earlier` asserts that if the search returns some position, then there is no earlier position where the haystack starts with the substring,
3. `not_found` asserts that if the search returns `None`, then there is no position in the haystack that starts with the substring.

To show that the requirements of this typeclass can be fulfilled, we proceed by showing that we can exhibit an instance of it by implementing a naive brute-force substring search algorithm. Its somewhat cumbersome definition given in Listing 9 stems from it being primarily written to facilitate the ease of proofs.

[28] Proven in Engine/Prefix.v#25-26

[29] A binary relation that is both reflexive and transitive.

[30] Proven in Engine/Prefix.v#49

Engine/Prefix.v#21-23

Listing 7: Definition of what it means for a string to be a prefix of another.

Engine/Prefix.v#83-98

Listing 8: Substring search typeclass.

```
Function brute_force_str_search (ss s: string) (i: nat) {measure (fun i => S (length s) - i)
 i} : option nat :=
  match Nat.leb i (length s) with
  | true => match starts_with_dec ss (List.skipn i s) with
    | left _ => Some i
    | right _ => brute_force_str_search ss s (S i)
    end
  | false => None
  end
```

Engine/Prefix.v#166-173

```
Instance BruteForceStrSearch: StrSearch := {
  str_search ss s := brute_force_str_search ss s 0
}
```

Engine/Prefix.v#221-223

Listing 9: Instance of StrSearch through a naive brute-force implementation.

From here we can prove the three axioms. We do it by proving a more generalized lemma for each of the axioms, which proves the result for any i, not only for $i = 0$. All proofs follow from inducting over the iteration of `brute_force_str_search`.

Engine/Prefix.v#178-181

```
Lemma (brute_force_str_search_starts_with) :
  forall ss s i j,
    brute_force_str_search ss s i = Some j ->
    starts_with ss (List.skipn j s)
```

Engine/Prefix.v#190-194

```
Lemma (brute_force_str_search_no_earlier) :
  forall ss s i j,
    brute_force_str_search ss s i = Some j ->
    forall k, i <= k < j ->
    ~ starts_with ss (List.skipn k s)
```

Engine/Prefix.v#205-209

```
Lemma (brute_force_str_search_not_found) :
  forall ss s i,
    brute_force_str_search ss s i = None ->
    forall k, i <= k <= length s ->
    ~ starts_with ss (List.skipn k s)
```

Finally, sometimes rather than operating on raw strings, we want to use Linden's `input` type. As such, we implement a simple wrapper function around the `StrSearch` typeclass which operates on `input`s rather than raw strings. Its definition is given in Listing 10.

Engine/Prefix.v#233-237

```
Definition input_search {strs: StrSearch} (p: string) (inp: input): option input :=
  match str_search p (next_str inp) with
  | Some i => Some (advance_input_n inp i forward)
  | None => None
  end
```

Listing 10: Wrapper around substring search operating on `input`s.

Once acquiring the offset to the occurrence of a substring, we simply advance the input by that many characters. For this input version we prove theorems analogous to the axioms of `StrSearch`, but expressed in terms of `input`s[33].

[33] Engine/Prefix.v#248-249, Engine/Prefix.v#279-283, Engine/Prefix.v#312-315

Additionally we prove that the result of `input_search` is always a suffix of the original input[34].

[34] Proven in Engine/Prefix.v#240-241

3.3 Correctness of literal extraction

We now prove that the extracted literals gives us some useful information about the matches of a regex. The properties which we care about are those which will allow us to accelerate regex matching. For that, we will consider three useful properties separately. For any literal whose prefix (Listing 3) is `s`, we want to show that any match of a regex r must start with the string s. Through the contrapositive (if a match does not start with s, it is not a match of r) we will be able to do prefix acceleration by skipping haystack positions where s does no occur. For `Impossible` literals, we want to show that no match of r whose literal is `Impossible` can exist. This will allow us to immediately say that for such a regex and any haystack, there is no match. Finally, for `Exact s` literals, we want to show that any match of a regex r whose literal is `Exact s` is exactly the string s. This will allow us to skip running regex engines entirely and just use a much faster substring search.

[35] Engine/Prefix.v#578-583

```
Definition extract_action_literal (a : action) : literal :=
  match a with
  | Areg r => extract_literal r
  | Acheck _ => Nothing
  | Aclose _ => Nothing
  end
```

[36] Engine/Prefix.v#585-589

```
Fixpoint extract_actions_literal (acts : list action) : literal :=
  match acts with
  | [] => Nothing
  | a :: rest => chain_literals (extract_action_literal a) (extract_actions_literal rest)
  end
```

Listing 11: Definitions of literal extractions generalized to tree actions.

We want the theorems to be stated in terms of the `is_tree` inductive[37], preferably talking about a specific regex, ie. `is_tree [Areg r]`. Recall, however, that some of its rules such as the `tree_sequence` rule talk about more than just the head of the tree actions. That means that in the proofs during induction over `is_tree` we must generalize over the entire list of tree actions, otherwise we will get stuck on those rules. To avoid this, we generalize literal extraction over an action and a list of actions in Listing 11 and state the theorems in terms of these.

[37] See Section 2.2

For `extract_action_literal` we return `Nothing` for non-regex actions, since they do not consume any characters from the input. For

`extract_actions_literal`, we chain the literals of each action. When the list of actions is empty, we return the same literal as we would for ε, `Nothing`. The choice of chaining becomes apparent when we look again at the `tree_sequence` rule: `is_tree (Areg (Sequence r1 r2) :: cont)` holds if `is_tree (Areg r1 :: Areg r2 :: cont)`[38] does.

[38] This is when the direction is `forward`. When the direction is `backward`, the condition is `is_tree (Areg r2 :: Areg r1 :: cont)`, leading to the same illustration of the argument.

With those definitions in place, we can now state the correctness theorems for each literal variant.

3.3.1 Correctness of the prefix of literals

We first want to state the correctness lemma of `extract_literal_char`. We want to say that given a character descriptor `cd` and a character `c` that matches it, the extracted literal from `cd` is the prefix of `c`. This is formalized by Lemma chain_literals_extract_char. In that statement we additionally generalize over the tail of the string where `c` is the head of it. Since the `ignoreCase` flag was not checked in `extract_literal_char`, we additionally add it to our hypotheses.

Lemma (chain_literals_extract_char) : Engine/Prefix.v#788-793

```
forall rest s c cd,
  RegExpRecord.ignoreCase rer = false ->
  starts_with (prefix rest) s ->
  char_match rer c cd = true ->
  starts_with (prefix (chain_literals (extract_literal_char cd) rest)) (c :: s)
```

Proof. We induct on `cd` yielding cases for each character descriptor. Each case is closed directly or by cases analysis of `rest` and by the induction hypotheses.

With that lemma we can now state and prove the general lemma about the correctness of the prefix of extracted literals for regexes. Given a tree `tree` of actions `acts` over the input `inp`, if `tree` contains a match then `inp` starts with the prefix of the literal extracted from `acts`.

Lemma (extract_literal_prefix_general) : Engine/Prefix.v#831-835

```
forall acts tree inp gm,
  is_tree rer acts inp Groups.GroupMap.empty forward tree ->
  (exists result, tree_res tree gm inp forward = Some result) ->
  starts_with (prefix (extract_actions_literal acts)) (next_str inp)
```

Proof. We induct on the `is_tree` hypothesis and use Lemma chain_literals_extract_char.

Given the generalized lemma we can now specialize it to the case where the list of actions is exactly just the regex `r` itself. This gives us Theorem extract_literal_prefix.

THEOREM (extract_literal_prefix) : Engine/Prefix.v#888-892

```
forall r tree inp,
  is_tree rer [Areg r] inp Groups.GroupMap.empty forward tree ->
  (exists result, first_leaf tree inp = Some result) ->
  starts_with (prefix (extract_literal r)) (next_str inp)
```

Proof. This holds directly from Lemma extract_literal_prefix_general with `acts = [Areg r]`.

In practice, however, this theorem will be of little direct use. We wish to instead have a theorem which would talk about the matches of a tree given some information about whether the input starts with the prefix of the literal. What we precisely want is the contrapositive of Theorem extract_literal_prefix which is stated in Corollary extract_literal_prefix_contra.

COROLLARY (extract_literal_prefix_contra) : Engine/Prefix.v#920-924

```
forall r tree inp,
  is_tree rer [Areg r] inp Groups.GroupMap.empty forward tree ->
  ~(starts_with (prefix (extract_literal r)) (next_str inp)) ->
  first_leaf tree inp = None
```

3.3.2 Correctness of `Impossible` literals

Similarly we must first state the correctness lemma of `extract_literal_char` for when it returns `Impossible`. Given a character descriptor `cd` for which we extract the literal `Impossible`, no character `c` can be matched with `cd`. This is formalized by Lemma extract_literal_char_impossible_no_match.

LEMMA (extract_literal_char_impossible_no_match) : Engine/Prefix.v#656-659

```
forall cd c,
  extract_literal_char cd = Impossible ->
  ~(char_match' rer c cd = true)
```

Proof. By induction on `cd`.

With that lemma we prove the general lemma about the `Impossible`. Given a tree `tree` of actions `acts` over the input `inp`, if the literal extracted from `acts` is `Impossible`, then no match can exist in `tree`.

LEMMA (extract_literal_impossible_general) : Engine/Prefix.v#674-678

```
forall acts tree inp gm gm',
  is_tree rer acts inp gm' forward tree ->
  extract_actions_literal acts = Impossible ->
  tree_res tree gm inp forward = None
```

Proof. We induct on the `is_tree` hypothesis and use Lemma extract_literal_char_impossible_no_match.

And finally, we specialize it to the case where the list of actions is exactly just the regex `r` itself. This gives us Theorem extract_literal_impossible.

THEOREM (extract_literal_impossible) : Engine/Prefix.v#774-778

```
forall r tree inp,
  is_tree rer [Areg r] inp Groups.GroupMap.empty forward tree ->
  extract_literal r = Impossible ->
  first_leaf tree inp = None
```

Proof. This holds directly from Lemma extract_literal_impossible_general with `acts = [Areg r]`.

3.3.3 Correctness of `Exact` literals

4 Prefix acceleration of the PikeVM

In this chapter we implement a practical optimization based on the literals extracted from regexes. During the runtime of a regex engine, we want to skip positions of the haystack that cannot possibly lead to a match. The information from literals gives us an under-approximation of which positions cannot lead to a match. Corollary extract_literal_prefix_contra states a condition which can determine positions of the haystack where the match cannot exist. Using that, we implement prefix acceleration. Unfortunately, we quickly realize that it is quite unclear how to implement prefix acceleration if we treat the regex engine as a black-box[39] while wanting to preserve the runtime asymptotic characteristics. Thus, we extend the existing formalization of a regex engine called the PikeVM in Linden [BDP26], by incorporating prefix acceleration directly into it. We prove that this extension is correct. Before we can notice the problem with the black-box assumption, we must discuss the difference between **anchored** and **unanchored** matching.

[39] A model where the internal workings are not known by the user. The only interaction with it are possible through its public interface.

Until this point, the semantics of matching we have presented correspond to the questions of *does* a regex match a haystack exactly at this current position and if so, *how*? Let us illustrate this by introducing a running

example. We want to match the regex `/a(?:b|be)w+/` against the haystack "ababccccccabeww". Say we want to check if the regex matches the haystack at the position marked by the arrow "ababccccccabeww". The answer is no, we fail to match due to the missing "w" which required by the regex. What about at "ababccccccabeww"? Also not, the regex requires "a" to be present at the starting position. Finally, what about at "ababccccccabeww"? Here we succeed with the matching being the substring "abeww". Each time, we have performed an **anchored** match. Approaching the specification of how matching works from the angle of anchored matching gives rise to cleaner semantics hence why these serve as the basis of formalization. In fact, the ECMAScript standard too defines the semantics in terms of anchored matching. In practice however, we are much more interested in **unanchored** matching, which asks the question of *does* a regex match anywhere in the haystack and if so, *how*? Since we are interested in leftmost greedy semantics[40], we specifically care about the leftmost match. Performing unanchored matching on our running example even at the start position of the haystack would succeed since as we saw there is a match at the end of the string, "ababccccccabeww". Before this work, the engines which were formalized in Linden were anchored ones.

[40] See Section 2.1.2

Luckily, every anchored engine can be easily adjusted to perform unanchored matching. Say we want to find the unanchored match of the regex `r`. Instead of running the anchored engine with `r`, we run it with `/[^]*?r/`. The prepended `[^]*?` construct is called the lazy prefix. Recall that the character descriptor `[^]` matches any character. This character descriptor is iterated an arbitrary number of times (`*`) in a lazy manner (`?`). This means matching `r` always has highest priority, but the lazy prefix allows us to advance the haystack by one each time `r` failed to match! As soon as `r` matches, we count this as the result.[41]

[41] This simplified explanation is made formal and proven true in Section 5.1.

Let us now assume we have access to a black-box anchored regex engine with a runtime characteristic which is of interest to us, namely being linear in the size of the regex and the haystack. We want to use it together with prefix acceleration to perform unanchored matching while staying linear. For now we can assume that finding all occurrences of a substring in a haystack (which is what prefix acceleration boils down to) takes linear time. This assumption is made formal and proven true in Section 4.3. We immediately run into a limitation: we are allowed to invoke this engine only a constant number of

times! Otherwise, the runtime complexity of our algorithm would be degraded. On the other hand, when we run the anchored engine with the lazy prefix, the engine will run until the exhaustion of the haystack not giving us any opportunities to perform acceleration. In fact, the best strategy which is known under this block-box assumption is to perform prefix acceleration a single time at the start, and from that found position run the anchored engine with the lazy prefix. This strategy is formalized and proven correct in Section 5.2. This strategy is unsatisfactory since it allows us to skip positions in the haystack only once. This drawback becomes apparent when we look at our running example. The literal which we can extract for `/a(?:b|be)w+/` is `Prefix "ab"`. Trying to accelerate once on the haystack "ababccccccabeww" immediately yields a false-positive: the prefix "ab" is found at the very beginning of the haystack but it leads to no match of the entire regex. The rest of the haystack must be matched without any help from prefix acceleration. We must thus dive into the internals of an engine to take full advantage of prefix acceleration.

We start by presenting the most popular linear regex engine called PikeVM in Section 4.1, which has already been previous verified in Linden. Next in Section 4.2, we introduce a new version of the PikeVM which turns the previously formalized **anchored** version into an **unanchored** one by integrating prefix acceleration. This form of prefix acceleration is novel and differs from the usual one due to its *filtering* optimization. Then, we prove that the resulting engine retains the linear runtime characteristics in Section 4.3. Finally, by diving deeper into Linden we discuss the proof of correctness of this new unanchored PikeVM in Section 4.4.

4.1 The PikeVM

In this work we have emphasized several times the importance of linear engines and why they are of interest to us. One such engine that has become ubiquitous in practical implementations of linear regex engines is the PikeVM [Rus09]. For a regex r and a haystack s, it has a runtime complexity of $O(|r| \cdot |s|)$ and a space complexity of $O(|r|^2)$. It is implemented by popular libraries such as Rust's `regex` crate [And16], RE2 [RP06], V8 [V8], or Golang [Gol]. Some of these libraries additionally implement other linear engines which tend to perform better on specific classes of regexes and haystacks, but have a drawback of either not supporting all of the desired regex features or having

a worse space complexity. Consider, for example, a different popular linear engine called the memoized backtracker. In various benchmarks it outperforms the PikeVM, but it incurs a higher space complexity of $O(|r| \cdot |s|)$ which makes it impractical for large haystacks. Another example is the LazyDFA engine which also outperforms the PikeVM, but currently there is no known way to incorporate support for general capture groups into it. Hence, the PikeVM is still the fallback engine of choice in those libraries. The following section presents how the PikeVM searches for matches.

Compilation. Before the PikeVM can execute a regex r on any haystack s, the PikeVM first compiles the regex down into a bytecode representation. This bytecode is a sequence of instructions that represent the operations needed to perform matching. It describes the exploration of an extended form of NFA corresponding to the regex r. This extended NFA encodes things that are not present in traditional NFAs, such as priority and capture groups. Each instruction has a label that uniquely identifies it within the bytecode.

Execution. Once we have the bytecode, we can execute it on a particular haystack s. The execution proceeds by simulating the exploration of the NFA using the bytecode. To track its progress, the PikeVM maintains a state containing so-called *"threads"*. Each thread is essentially a bytecode label that can be thought of as a program counter (PC)[42]. All threads are synchronized to the same position in s. The threads that are currently exploring this position in the haystack are stored in an *"active set"*. Since our backtracking semantics have a notion of priority, the active set is a list ordered by highest priority. To recall, the regex `/aa|aab/` produces a match of "zaab", where the last "b" is not included because the first alternative of the disjunction matches first. So during exploration, the thread corresponding to the first alternative is ordered before the thread corresponding to the second alternative in the active set. Whenever performing work on an active thread, it can lead to three situations.

[42] An integer value that indicates the current position of execution within a larger sequence of instructions. Storing it allows resuming execution from that point later.

1. Since all active threads are synchronized to the same position in the haystack, as soon as an active thread needs to advance to the next character, it is transferred to a *"blocked set"* that is similarly ordered by priority.
2. A thread might complete a unit of work and indicate that more work needs to be done by producing a list of new threads that should be prepended to the active set. For example, if a thread is executing the bytecode responsible

for performing a disjunction, it will produce two new threads, one for each branch. To indicate that a thread failed to match (for example due to expecting to see "z" at "zaab"), it can produce an empty list of new threads, effectively ending the thread.

3. As soon as an active thread reaches the accepting state of the NFA, it dies and produces the current *"best match"* to be stored in the PikeVM's state. The best match represents the current highest priority match found so far. Since we are interested in leftmost semantics, storing the best match so far allows us to wait for a potentially higher priority match to be found later. Notably, a higher priority match can be produced from threads that are currently in the blocked set.

Finally, there is one additional item in the state of a PikeVM: a *"seen set"*. The seen set stores labels of instructions which have already been executed at the current haystack position. A thread is discarded when it revisits one of those labels. This caching is crucial for giving the PikeVM a linear runtime.

We can now summarize the execution of the PikeVM. It repeatedly performs work on the active set until it is exhausted. If the blocked set is empty, it returns the best match found so far. Otherwise, it advances the haystack position by one and transfers the blocked set into the new active set. We illustrate this execution model with an example trace seen in Figure 1. For ease of understanding, we visualize the NFA as a simplified state machine instead of bytecode instructions. The nodes annotated with Greek letters represent the labels and the arrows are annotated with the characters they expect to see in the current haystack position. When relevant, we indicate the priority of the arrows with the smallest number being the highest priority.

Despite a match existing at the tail end of the exemplified haystack, no match is found due to the PikeVM being inherently an anchored engine. In the next section we will devise a way to turn it into an unanchored engine all whilst integrating prefix acceleration.

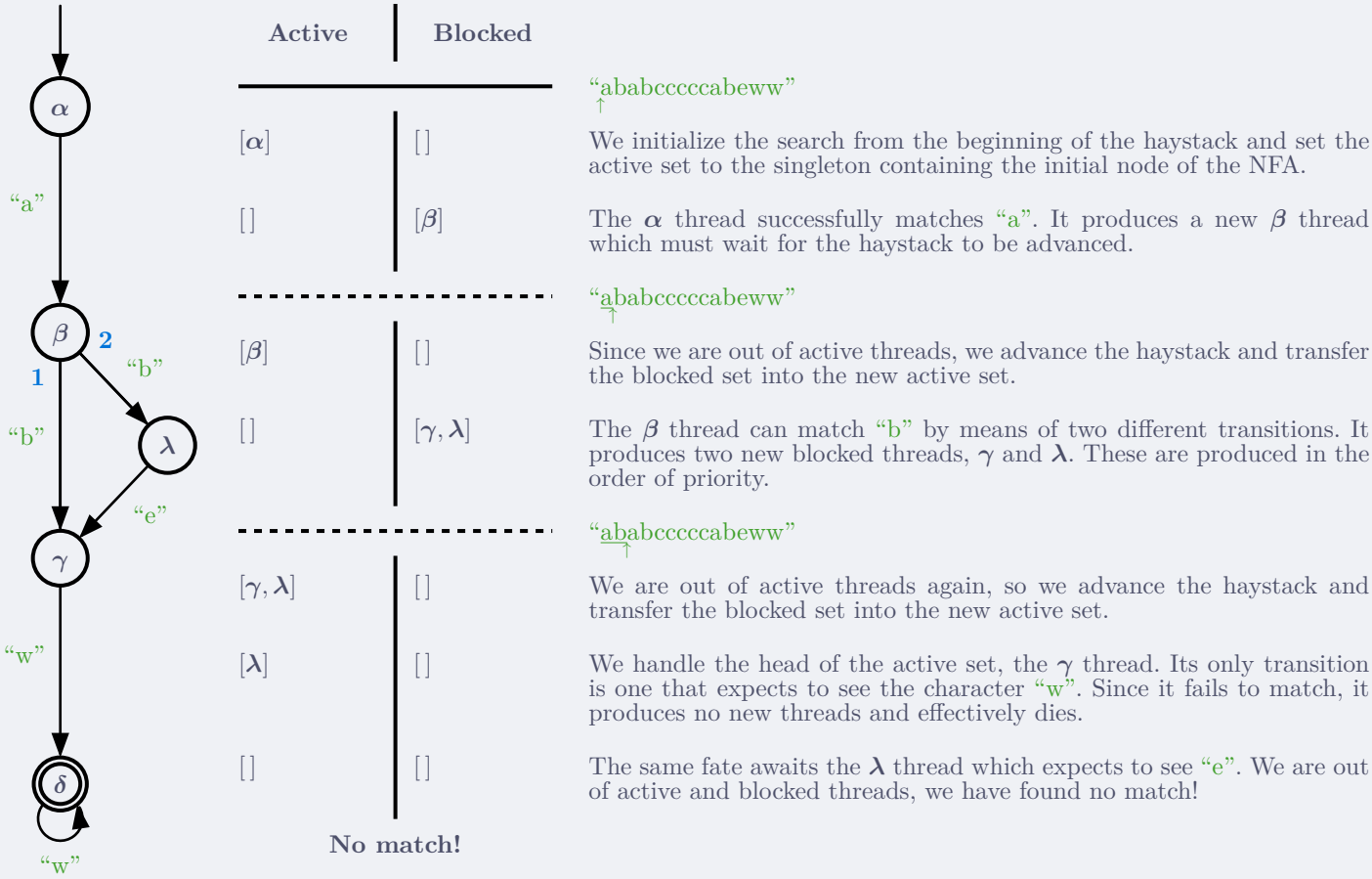

Figure 1: Left: the NFA of the regex `/a(?:b|be)w+/`. Right: execution trace of the PikeVM on "ababcccc-cabeww".

4.2 The unanchored PikeVM

We wish to turn the PikeVM into an unanchored engine. Since we also would like to take advantage of prefix acceleration, we must do more than just run the anchored PikeVM with a lazy prefix. Instead, we design a new version of the PikeVM which will perform unanchored matching while integrating prefix acceleration directly into its execution model.

We first make an observation that the lazy prefix can be simulated by the PikeVM itself. Recall that the lazy prefix for some regex r simply allows us to try to match r at every position of the haystack while prioritizing the leftmost match. Notice, that to attempt a match at some position in the haystack, the PikeVM must execute the thread initialized to the initial label of the NFA of r. Let l_0 be the initial label of the NFA (in Figure 1, α was the initial label l_0). Thus, to run l_0 at every position of the haystack, we modify two aspects of the PikeVM's execution model. First, we ensure that l_0 is always present in the active set when we start exploring a new position of the haystack. This is done by altering the step of advancing the haystack. Instead of just setting

the active set to the current blocked set while clearing the blocked set, we additionally append l_0 to the end of the active set. Second, when we run out of both active and blocked threads but still have more haystack positions to explore, we restart the matching by placing l_0 into the active set. To preserve the leftmost semantics, we only do this restart if no best match has been found so far.

This simulation effectively resembles the execution model of an anchored PikeVM running with a lazy prefix. However, by doing the simulation manually, we gain more fine grained control allowing us to additionally integrate prefix acceleration. We do so by implementing two kinds of optimizations, **filtering** and **accelerating**. Before running this new unanchored PikeVM, we first extract a literal ℓ from the regex r using literal extraction defined in Chapter 3.

Filtering. During the simulation of the lazy prefix, we said we want to append l_0 each time we advance the haystack. However, if we know the new haystack position cannot possibly be the start of a successful match, we can skip appending l_0 completely. Corollary extract_literal_prefix_contra establishes a concrete condition for when there can be no match. If ℓ is not the prefix of the current haystack position, we know a match cannot exist starting from here and thus we skip appending l_0. This optimization allows us to skip performing work for which we know no results will be produced.

Accelerating. Whenever the PikeVM has no more active and blocked threads, we said we want to restart the matching with l_0 at the next haystack position. But similarly to filtering, if we know that the next haystack position cannot produce a match, we can skip restarting the matching there. Instead, we jump immediately to the next position where ℓ is found, and restart with l_0 there.

Those two optimizations together allow us to potentially skip large parts of the haystack multiple times thanks to exploiting the internals of the PikeVM. We call this new version of the PikeVM the *unanchored PikeVM*. To our knowledge, the filtering optimization is a novel contribution not implemented by existing real-world engines. The drawback of performing filtering is that we must precompute the next occurrence of ℓ in advance. This makes this form of prefix acceleration non-streaming. However, the benefits can be

substantial if we accept this trade-off. The benefits of filtering are evaluated in Section 6.2.

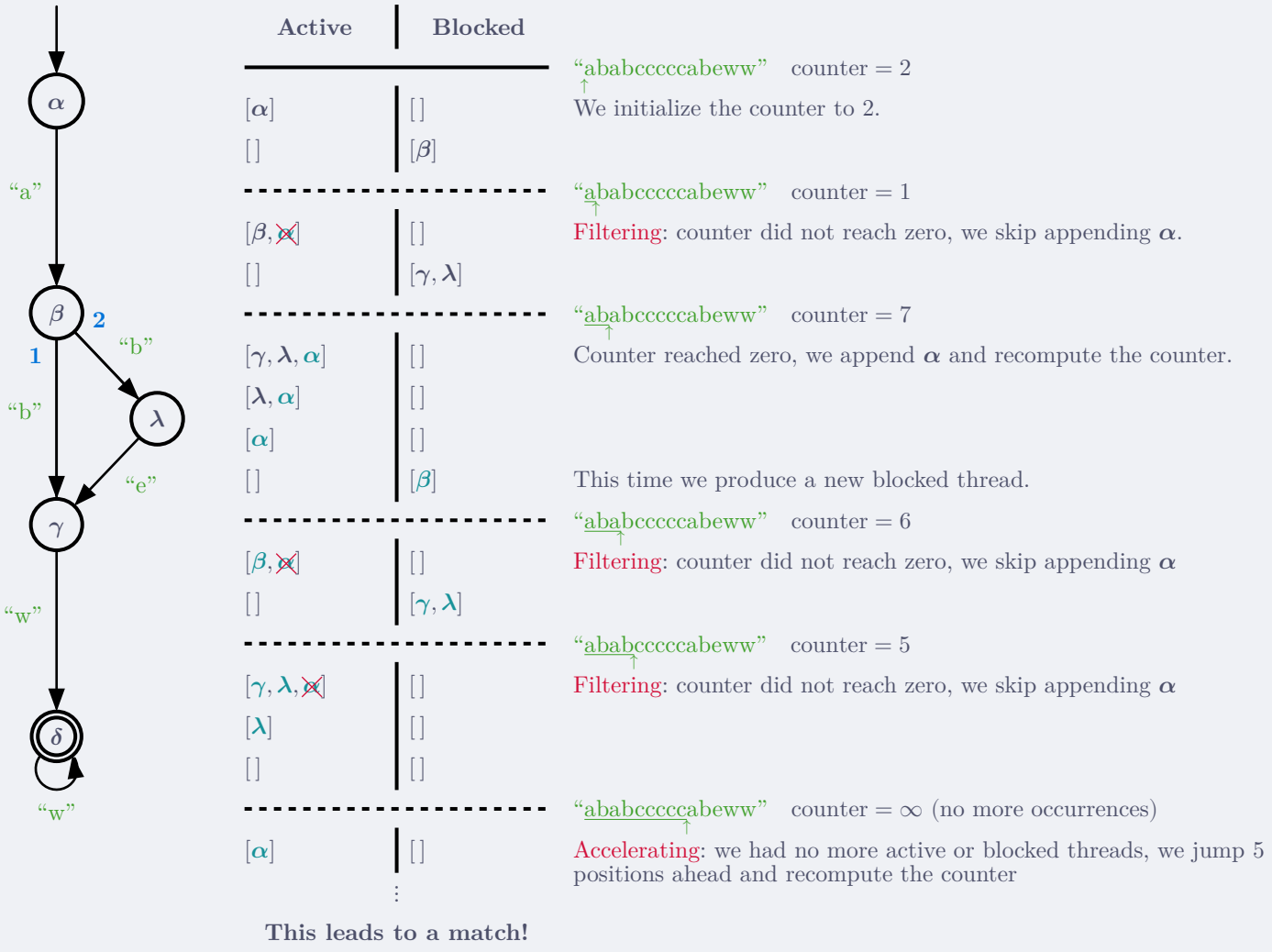

Figure 2: Left: the NFA of the regex `/a(?:b|be)w+/`. Right: execution trace of the unanchored PikeVM with the extracted literal ℓ = `Prefix "ab"` on the haystack "ababcccc-cabeww". We color-code new threads previously not created by the anchored PikeVM.

To implement these optimizations we augment the PikeVM state with a counter that indicates in how many characters we will find ℓ in the haystack. This counter's value will be always obtained from running the substring search procedure defined in Section 3.2. Each time we advance the haystack position, we decrement this counter by one. If the counter is non-zero we may perform filtering by not appending l_0 to the active set. If the counter is zero, we must append l_0 and compute the new value of the counter. To perform acceleration, when we have no more active and blocked threads we advance the haystack position by the value of the counter and recompute the counter. We illustrate this new execution model by revisiting our running example and showing the execution trace of this unanchored PikeVM in Figure 2. For our regex

`/a(?:b|be)w+/`, the literal ℓ is `Prefix "ab"` and thus each match must start with the prefix "ab".

4.3 Complexity analysis

We want to ensure that the prefix acceleration optimization does not degrade the asymptotic complexity our regex engines provide. To show no additional asymptotic complexity is incurred, we prove a more general result about prefix acceleration strategies and show that the one proven correct in this work fits into this framework.

Let ss be a string and s be a haystack. Let $\mathsf{ssearch} : \mathsf{string} \times \mathsf{string} \to \mathsf{option}\ \mathbb{N}$ be a substring search algorithm such that $\mathsf{ssearch}\ ss\ s$ returns the index of the first occurrence of the substring ss within s. If no occurrence exists, it returns `None`.

Definition (Streaming-linear substring search): We say ssearch is *streaming-linear* if its runtime has worst-case complexity $O(|ss| \cdot |s|)$ and whenever the index of the first occurrence of the substring is $k \leq |s|$, then ssearch finds it in $O(|ss| \cdot k)$.

Intuitively, the streaming-linear property ensures that when finding occurrences we do not needlessly explore more of the haystack than required. Since prefix acceleration calls substring searches multiple times, the weaker *linear* property of substring searches does not suffice to preserve linearity of prefix acceleration. Luckily, the vast majority of practical substring search algorithms are already streaming-linear.

Definition (Progressing search algorithm): An algorithm T over a haystack equipped with ssearch is called a *progressing search algorithm* if each i-th ssearch call by T with the haystack advanced to position p_i with a result k_i is such that $p_{i+1} > p_i + k_i$. We set $p_0 = -1$ and $k_0 = 0$.

The progressing search property ensures that each ssearch call makes progress in the haystack and thus ssearch does not explore the same part of the haystack more than once. Consider the following example trace of a progressing search algorithm T. We underline the haystack portion on which ssearch can no longer be called on due to the progression property, and mark the haystack position on which ssearch is called.

: T is initialized with the haystack "hello world"

"hello world" : T calls ssearch at position 0 with substring "lo" $\rightarrow 3$

"hello world" : T does some work which advances the haystack position by 1

"hello world" : T calls ssearch at position 5 with substring "o" $\rightarrow 2$

"hello world" : T calls ssearch at position 8 with substring "rld" $\rightarrow 0$

"hello world" : T terminates

We are now ready to state and prove the main theorem of this section.

THEOREM (Progressing search algorithm complexity) Let T be a progressing search algorithm over a haystack s equipped with a streaming-linear substring search ssearch. Let Q be the runtime complexity of T under the assumption that ssearch calls have a cost of $O(1)$. Let m_i be the substring used in the i-th ssearch call. Then, the runtime complexity of T is $O(Q + \max_i |m_i| \cdot |s|)$.

Proof. Consider an execution of T where ssearch is called c times with substrings $m_1, m_2, ..., m_c$ and results $k_1, k_2, ..., k_c$. Then the total cost of all ssearch calls is

$$O(m_1 k_1) + O(m_2 k_2) + ... + O(m_c k_c) \leq O\Big(\max_i |m_i| \cdot (k_1 + k_2 + ... + k_c)\Big)$$

due to ssearch being streaming-linear. By the progressing search property we have that

$$k_1 + k_2 + ... + k_c \leq |s|$$

Thus, the total cost of all ssearch calls is $O(\max_i |m_i| \cdot |s|)$. Since calls to ssearch are only an additive cost, the total runtime complexity of T is $O(Q + \max_i |m_i| \cdot |s|)$, yielding the desired result.

We now specialize the theorem above to our prefix acceleration. First, we take ssearch to be our substring search procedure `str_search` defined in Listing 8. The unanchored PikeVM is our progressing search algorithm T, it calls `str_search` in a progressing manner. The substrings used in calls into `str_search` are always the extracted literal ℓ of a regex r, thus $\max_i |m_i| = |\ell|$. But by Theorem extract_literal_size_bound, we get that $|\ell| \leq |r|$, and so $\max_i |m_i| \leq |r|$. Finally, to establish Q we have proven in Linden that when treating calls into `str_search` as a constant cost of $O(1)$, the runtime complexity of the unanchored PikeVM is bounded by $O(|r| \cdot |s|)$[43]. Altogether, by Theorem Progressing search algorithm complexity we get that the total runtime complexity of the prefix accelerated unanchored PikeVM is $O(|r| \cdot |s| + |\ell| \cdot |s|) \leq O(|r| \cdot |s| + |r| \cdot |s|) = O(|r| \cdot |s|)$. Thus, the asymptotic complexity of the PikeVM remains the same when extended with prefix acceleration.

[43] Proven in Engine/Complexity.v#811-817

4.4 Correctness

To convince ourselves that the new unanchored PikeVM algorithm is correct, we need a baseline to compare it against. As such, we prove that the result returned by this PikeVM variant is exactly the one defined by our backtracking tree semantics. This in turn implies that we are coherent with the matching semantics defined by ECMAScript. This result has been already established and proven for the original anchored PikeVM in Linden. The original proof is quite involved and lays out an entire pipeline of how to go from a VM executing instructions all the way down to the backtracking tree semantics. In this chapter we outline the most important aspects of that proof pipeline and we refer the reader to Chapter 6 of [BDP26] for all the details. Other than the obvious desire to get a `Qed.` on the proof of correctness for our new algorithm, there was a strong motivation to reuse as much as possible of the existing proof infrastructure from the anchored PikeVM. This has been successfully achieved by modifying the anchored PikeVM semantics to be able to operate both in an anchored fashion as well as in an unanchored one. This change of operation mode is achieved purely by picking a different initial state for the PikeVM. Due to this proof engineering effort, the vast majority of the proof pipeline is reused.

We first discuss the technical details that allowed to modify the anchored PikeVM definition to support unanchored matching without losing any of the previous functionality. The PikeVM's state is extended with a single optional parameter which we call `nextprefix`. It stores three things: the literal ℓ extracted from the regex, a counter indicating in how many characters the prefix of that literal will match the haystack, and an instance of a substring search algorithm conforming to Listing 8. If this triple is not set, the PikeVM will operate in the original anchored mode. The original small-step semantics of the PikeVM[44] are modified to take this new state parameter into account. Most of them remain unchanged, but instead new small-step rules are added to perform the work needed to, 1) simulate the lazy prefix[45], 2) perform the filtering optimization[46], 3) perform the acceleration optimization[47]. Below we explain which existing rules needed adjustments.

- One of the small-step rules[48] was responsible for handling the case when we find a match and therefore store it as our best match so far. This rule

[44] Defined in `Engine/PikeVM.v#165-226`

[45] `pvs_nextchar_generate`

[46] `pvs_nextchar_filter`

[47] `pvs_acc`

[48] `pvs_match`

must be modified to set `nextprefix` to `None` at this point to downgrade the execution to anchored mode. If this was not done, the unanchored PikeVM would continue trying to find matches further down the haystack potentially overwriting the current best match, which would violate the semantics of finding the leftmost match.

- The original rule for advancing the haystack position[49] is restricted to only apply in the anchored mode. In unanchored mode, we instead want to use the rule which additionally simulates the lazy prefix / does filtering.
- The original rule for handling the case when there are no active threads nor blocked threads[50] is also restricted to only apply in anchored mode. In unanchored mode, we instead want to use the rule which performs the acceleration optimization.

[49] `pvs_nextchar`

[50] `pvs_final`

$$\frac{\texttt{state.nextprefix = Some (0, lit, search)} \quad \texttt{advance\_input state.inp = inp'} \quad \texttt{state.blocked = thr :: blocked} \quad \texttt{state.active = []}}{\begin{array}{c}\texttt{state'.inp = inp'} \\ \texttt{state'.active = (thr :: blocked) ++ [initial\_thread]} \quad \texttt{state'.blocked = []} \\ \texttt{state'.nextprefix = compute\_nextprefix lit search inp'}\end{array}}\ \textsc{Generate}$$

$$\frac{\texttt{state.nextprefix = Some (S n, lit, search)} \quad \texttt{advance\_input state.inp = inp'} \quad \texttt{state.blocked = thr :: blocked} \quad \texttt{state.active = []}}{\begin{array}{c}\texttt{state'.inp = inp'} \\ \texttt{state'.active = thr :: blocked} \quad \texttt{state'.blocked = []} \\ \texttt{state'.nextprefix = (n, lit, search)}\end{array}}\ \textsc{Filter}$$

$$\frac{\texttt{state.nextprefix = Some (n, lit, search)} \quad \texttt{advance\_input\_n (S n) state.inp = inp'} \quad \texttt{state.blocked = []} \quad \texttt{state.active = []}}{\begin{array}{c}\texttt{state'.inp = inp'} \\ \texttt{state'.active = [initial\_thread]} \quad \texttt{state'.blocked = []} \\ \texttt{state'.nextprefix = compute\_nextprefix lit search inp'}\end{array}}\ \textsc{Accelerate}$$

Figure 23: The new small-step rules added to the PikeVM to support unanchored matching. `state` is the current state of the PikeVM and `state'` is the resulting state after applying the rule. Unmentioned fields of the state remain unchanged.

With these modifications, we extend the small-step semantics with the following three new rules, visualized in Figure 23

1) To simulate the lazy prefix, whenever `nextprefix` is set and its counter has reached zero, we advance the haystack position by one, move the blocked threads to the active set, and append to the end of the active set a new thread with the label of the initial instruction of the bytecode. We must also recompute the value of the `nextprefix` counter by using the literal and the substring search algorithm stored in `nextprefix`.
2) To perform filtering, whenever `nextprefix` is set and its counter is greater than zero, we advance the haystack position by one, move the blocked

threads to the active set, but do not add any new threads. We also decrement the `nextprefix` counter by one.

3) To perform acceleration, whenever `nextprefix` is set and there are no active nor blocked threads, we use the substring search algorithm stored in `nextprefix` to skip ahead in the haystack to the next potential match position. We reinitialize the active set with a single thread with the label of the initial instruction of the bytecode. We also recompute the value of the `nextprefix` counter.

The new unanchored PikeVM has an executable functional version[51] which is proven to follow exactly the small-step semantics[52].

[51] Defined in Engine/FunctionalPikeVM.v#98-102

[52] Proven in Engine/FunctionalPikeVM.v#181-184

With this setup we now discuss the essential aspects of the proof pipeline and how it is adjusted to accommodate the unanchored PikeVM. The main task is to adjust all of the proofs that performed any kind of case analysis on the small-step semantics of the PikeVM to these new rules. Additionally, all lemmas that stated something with regards to the initial state of the PikeVM have been duplicated to state the analogous thing for the unanchored initial state[53]. Otherwise, other theorems are fully reused without any changes. The challenge of proving the correctness of the PikeVM stems from its vastly different execution order and that it executes bytecode rather than operating on backtracking trees.

[53] These analogous lemmas are found next to the original one with a suffix of `_unanchored`

To bridge this gap, the existing proof pipeline goes through an intermediate engine called the *PikeTree* which sits somewhere between backtracking trees and the execution scheme of the PikeVM. Given a backtracking tree, the PikeTree explores it non-deterministically in an order analogous to the PikeVM. The PikeTree starts with the backtracking tree of the regex and haystack on which we are operating. It is the initial value of the active set of the PikeTree; instead of storing threads with bytecode labels, the PikeTree stores backtracking subtrees. The PikeTree then explores this tree in the same order as the PikeVM would explore its NFA, producing new subtrees in its active and blocked set along the way. The match-equivalence between the PikeTree and the backtracking trees is achieved through showing that as the PikeTree explores the backtracking tree, it maintains an invariant that the result from the PikeTree state remains the same[54]. Then, to prove the match-equivalence between the PikeVM and the PikeTree, we show that there exists a particular

[54] Initialization: Engine/PikeTree.v#386-392, Preservation: Engine/PikeTree.v#619-623

execution trace of the PikeTree which corresponds to the execution trace of the PikeVM. We show this correspondence by relating individual parts of the state of the PikeVM to the one of the PikeTree through a simulation invariant[55].

[55] Initialization: Engine/PikeEquiv.v#1087-1094, Preservation: Engine/PikeEquiv.v#1121-1136

With that simplified view of the already existing proof pipeline in mind, we now present the changes that were needed to retrofit unanchored matching to the PikeTree as well.

4.4.1 The unanchored PikeTree

We modify the small-step semantics of the PikeTree to accommodate unanchored matching in a similar fashion as we did for the PikeVM. PikeTree is not meant to be an executable or realistic engine, but merely a proof artifact. Its construction is already exponential and its execution is non-deterministic. We can use the non-determinism to our advantage, it allows for simpler modifications and abstractions. In the end, we only require that there exists an execution trace that will correspond to that of the PikeVM.

Firstly, we add a new parameter to the state of the PikeTree called `future`. It corresponds to PikeVM's `nextprefix`, but is simpler as it only stores a single backtracking subtree. When wanting to perform unanchored matching of the regex r on the haystack s, the initial value of `future` is set to be the subtree corresponding to the right branch of the backtracking tree of `/[^]*?r/` over s. We visualize the initial state of the unanchored PikeTree in Figure 24.

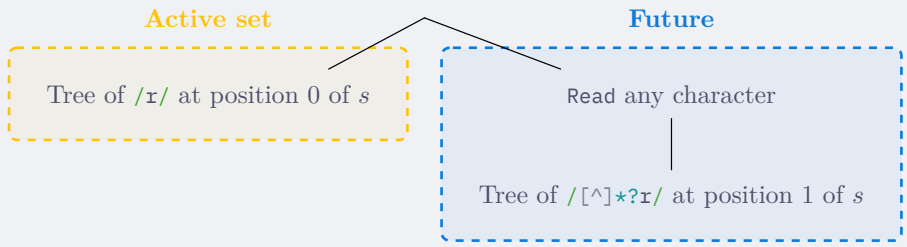

Figure 24: Backtracking tree of `/[^]*?r/` over a haystack s. Left subtree is the initial active set tree, right subtree is the initial `future` subtree.

During execution, the `future` tree will be used to simulate the lazy prefix. This `future` subtree is crucial for the PikeTree to be able to attempt matches at positions further down the haystack. Each time we want to simulate the lazy prefix, we remove the `Read` node from the `future` subtree which leaves us with a tree of the same shape as in Figure 24, but with the backtracking trees representing regexes at the next haystack position. We then set the new right subtree (which is a `Read` followed by the tree of `/[^]*?r/` at the next haystack position) as the new `future` and append the new left subtree (which is the tree of `/r/` at the next haystack position) to the active set. This way, we allow

the PikeTree to explore matches in the haystack positions that follow. When doing filtering, we do the same unfolding but we discard the new left subtree instead of appending it to the active set.

$$\frac{\text{state.future} = \begin{array}{c} \overset{t1}{\diagup\ \ \diagdown}\ \ \text{Read c} \\ \text{t2} \end{array} \quad \text{advance\_input state.inp} = \text{inp}' \quad \text{may\_erase t2} = \text{future}' \quad \text{state.blocked} = \text{tgm} :: \text{blocked} \quad \text{state.active} = []}{\begin{array}{c} \text{state}'.\text{inp} = \text{inp}' \quad \text{state}'.\text{future} = \text{future}' \\ \text{state}'.\text{active} = (\text{tgm} :: \text{blocked}) \mathbin{++} [t1] \quad \text{state}'.\text{blocked} = [] \end{array}}\ \textsc{Generate}$$

$$\frac{\text{state.future} = \begin{array}{c} \overset{t1}{\diagup\ \ \diagdown}\ \ \text{Read c} \\ \text{t2} \end{array} \quad \text{advance\_input state.inp} = \text{inp}' \quad \text{state.blocked} = \text{tgm} :: \text{blocked} \quad \text{state.active} = [] \quad \text{first\_leaf t1} = \text{None}}{\begin{array}{c} \text{state}'.\text{inp} = \text{inp}' \quad \text{state}'.\text{future} = \text{t2} \\ \text{state}'.\text{active} = \text{tgm} :: \text{blocked} \quad \text{state}'.\text{blocked} = [] \end{array}}\ \textsc{Filter}$$

$$\frac{\text{tree\_acceleration inp future} = (\text{inp}', \text{acc}, \text{t}) \quad \text{may\_erase acc} = \text{future}' \quad \text{state.blocked} = [] \quad \text{state.active} = []}{\begin{array}{c} \text{state}'.\text{inp} = \text{inp}' \quad \text{state}'.\text{future} = \text{future}' \\ \text{state}'.\text{active} = [t] \quad \text{state}'.\text{blocked} = [] \end{array}}\ \textsc{Accelerate}$$

Figure 25: The new small-step rules added to the PikeTree to support unanchored matching. `state` is the current state of the PikeTree and `state'` is the resulting state after applying the rule. Unmentioned fields of the state remain unchanged.

By having added the `future` state parameter, we now explain how it is used to create steps analogous to the new ones in the unanchored PikeVM. These rules are visualized in Figure 25. The PikeTree does not have access to the literal of regex we are executing nor does it have access to a substring search algorithm. The lack of it is not an issue, as we replaced it by having non-deterministic steps in the PikeTree semantics[56]. In the unanchored PikeVM, we simulated the lazy prefix only when it was necessary, namely when the counter of `nextprefix` reached zero indicating that the literal's prefix matches at that position. However, for correctness' sake, there is no issue in simulating the lazy prefix at every haystack position. Thus, for the PikeTree, the small-step rule responsible for simulating the lazy prefix[57] can be applied non-deterministically together with the filtering rule[58]. The only condition that is imposed on the filtering step is that we cannot filter out a subtree from `future` that did contain a successful match. This restriction does not prevent us from applying the lazy prefix simulation step even when that subtree contains no matches. Without having access to a literal or a substring search algorithm, this non-determinism is essential. We cannot make this choice deterministic by choosing to always filter whenever the `future` subtree contains no matches. Doing so would encode

[56] In fact, the choice of not including the literal and substring searches in the PikeTree was intentional. This way, it describes a broader class of optimizations that perform filtering and acceleration for potentially various reasons.

[57] `pts_nextchar_generate`

[58] `pts_nextchar_filter`

an execution strategy that takes ideal decisions. But the PikeVM's filtering rule is only an under-approximation of such an ideal strategy. The PikeVM may very well not filter out a position where a match is not present. Thus, this non-determinism allows us to say that there exists a trace of choices made by the PikeTree that corresponds to the concrete decisions made by the PikeVM. We must address one final discrepancy. In the PikeVM, the lazy prefix simulation step was recomputing the `nextprefix`. This recalculation could lead to setting `nextprefix` to `None` (no more occurrences found). Our unfolding of the `future` tree does not have this capability. To remedy this, we define a non-deterministic relation `may_erase` which can turn any tree into either itself or into `None`. We allow turning a tree into `None` under the sole condition that this tree does not contain any successful matches. This relation can be seed in Listing 12.

```
Inductive may_erase: tree -> option tree -> Prop :=
| no_erase:
  forall t, may_erase t (Some t)
| erases:
  forall t inp (NORES: first_leaf t inp = None),
    may_erase t None
```

Engine/PikeTree.v#85-90

Listing 12: Definition of the non-deterministic relation which erase trees.

We add this ability of erasing the new `future` tree to the lazy prefix simulation step. The trace of the PikeTree which corresponds to the PikeVM is such that whenever the PikeVM would recompute the `nextprefix` to `None`, `may_erase` will erase the `future` tree in the PikeTree. Having handled PikeTree's lazy prefix and filtering steps, we now turn to the acceleration step[59]. To perform acceleration, we want to be able to non-deterministically perform this unfolding of `future` multiple times, where each unfold is under the condition that the tree we are discarding does not contain any successful matches. To model this, we define a non-deterministic relation `tree_acceleration`[60] which takes as input the current haystack position and the `future` tree. It returns a new haystack position to which we have non-deterministically unfolded `future`, the `/r/` subtree `t` at that position, and the `/[^]*?/` subtree `future'` at that position. We set the new active set to be the singleton of `t`. The new `future` is the `may_erase` of `future'` for the same reason as before: after acceleration, the PikeVM can potentially recompute the `nextprefix` to be `None`. The execution trace which corresponds to the PikeVM is such that the we unfold the `future`

[59] `pts_acc`

[60] Defined in Engine/PikeTree.v#115-127

tree the same amount of times as the value of the counter in PikeVM's `nextprefix`.

With these modifications, we have adapted the PikeTree to support unanchored matching. There is a clear correspondence between the new small-step rules of the unanchored PikeVM and the ones of the unanchored PikeTree. As for the PikeVM, the new rules for the PikeTree required adjusting existing proofs that performed case-analysis on the small-step semantics. We also duplicate all of the lemmas mentioning the initial state of the PikeTree to separately state the analogous thing about the unanchored initial state[61]. The remaining proofs did not require any changes.

[61] These analogous lemmas are found next to the original one with a suffix of `_unanchored`

We conclude this section by briefly discussing the new simulation invariant that relates PikeVM's `nextprefix` to PikeTree's `future`. Firstly, to support the anchored modes of both engines, we state that PikeVM's `nextprefix` is `None` if and only if PikeTree's `future` is `None`. Secondly, when both are set, we relate the counter in `nextprefix` to the number of left subtrees in `future` containing no successful matches. This invariant is indeed preserved; the `nextprefix` counter is initialized to the distance from the next occurrence of a literal. We know that for all haystack positions before that occurrence, there can exist no match. Therefore, the subtree most certainly also contains no successful matches.

This large bulk of work culminates in the main theorem stating the match-equivalence between the unanchored PikeVM and the backtracking tree semantics.

Engine/Correctness.v#135-144

THEOREM (pike_vm_correct_unanchored) {strs:StrSearch}:
```
 forall r inp tree result,
   (* the regex `r` is in the supported subset *)
   pike_regex r ->
   (* `tree` is the tree of the regex `[^]*?r` for the input `inp` *)
   is_tree rer [Areg (lazy_prefix r)] inp GroupMap.empty forward tree ->
   (* the result of the PikeVM is `result` *)
   trc_pike_vm (compilation r) (pike_vm_initial_state_unanchored (extract_literal rer r)
inp) (PVS_final result) ->
   (* This `result` is the priority result of the `tree` *)
   result = first_leaf tree inp
```

Proof. By transitivity of match-equivalence through the unanchored PikeTree.

5 Meta engine

Various regex engines have their strengths and weaknesses. Some operate better when specific regex features are used, others excel whenever the haystack is small. Some suffer from having a troublesome space complexity, others don't support all of the desired regex features. To get the best of all worlds, modern regex libraries implement so-called *meta engines* which combine multiple individual engines and select the best one for a given regex and haystack. The goal of the Meta engine is to take high-level decisions such as selecting the best engine for the job guided by heuristics and perform engine-independent optimizations. At times, we may even want to skip running any engine at all or combine multiple in the same matching process.

As mentioned in Section 4.1, PikeVM remains the most feature-complete engine with a linear runtime complexity and a reasonable space complexity. Its importance is undeniable. But as noted by the author of the Rust's `regex` crate[62] [And16],

> *... the less time we can spend in specifically the PikeVM, the better.*
>
> — *Andrew Gallant [And23a]*

[62] The name used to describe packages in the Rust ecosystem. Crates are most often published to and downloaded from the crates.io repository.

This harsh statement stems from the fact that while the PikeVM has favorable properties, it tends to be rather slow in practice. We know many other regex matching algorithms which are considerably faster in practice. Their speed is partially attributed to them being specialized for specific regex features or haystack characteristics. In Linden, besides the PikeVM, another engine called the memoized backtracker (MemoBT)[63] has been mechanized and verified. In benchmarks the MemoBT tends to be roughly 2× faster than the PikeVM [And23b]. However, its drawback is that its space complexity is expressed in terms of both the regex and haystack size, $O(|r| \cdot |s|)$ (as opposed to PikeVM's $O(|r|^2)$[64]), making it impractical for large haystacks. Tools like UNIX's `grep` command-line tool, which given a regex finds matches in files, often perform searches on rather large files, and so the incurred memory cost of the MemoBT is at times prohibitive.

[63] A regex engine that combines backtracking with memoization to avoid redundant computations. It explores possible matches like a backtracking engine but stores intermediate results to ensure that each unique state is only computed once. It has a $O(|r| \cdot |s|)$ runtime and space complexity.

[64] In practice, haystacks tend to be much larger than regexes.

In this chapter we formalize such a Meta engine and prove the correctness of its decision procedures and various optimizations it performs. Since at the moment of writing only the PikeVM and the MemoBT engines have been

mechanized in Linden, we focus on those. We start by formalizing what it means to be an engine in Section 5.1. Next, we revisit literal extraction to leverage `Exact` and `Impossible` literals in Section 5.2. Then, we take advantage of anchors in regexes to avoid redundant work in Section 5.3. Finally, we combine everything together into a single Meta engine in Section 5.4.

5.1 Engine interface

So far we have been mentioning regex engines in a rather informal manner. In this section we formalize this notion. Some optimizations which we perform in the following sections hold true for any engine even when treated as a black-box. Hence having an abstract description of what an engine is allows us to state more general results. Since anchored and unanchored engines return fundamentally different results of matching, we separate them into two disjoint definitions.

The typeclass describing an anchored regex engine given in Listing 13 contains three members. First, it must contain a function which will perform the actual matching, `exec`. Given a regex and a haystack, it optionally returns a result. To control which regexes the engine supports, it must additionally provide a predicate which given a regex returns whether the engine is able to execute it. Finally, there is a single correctness axiom the engine must fulfill, `exec_correct`. It states that that for any supported regex r and any haystack s, the engine (`exec r s`) returns exactly the same result as the one defined by the semantics of backtracking trees. This in turn means that the engine finds a match as defined by the ECMAScript standard.

Engine/Meta/EngineSpec.v#25-36

```
Class AnchoredEngine := {
  exec: regex -> input -> option leaf;

  (* asserts the supported subset of regexes *)
  supported_regex: regex -> bool;

  (* the execution follows the backtracking tree semantics *)
  exec_correct: forall r inp tree,
    supported_regex r = true ->
    is_tree rer [Areg r] inp Groups.GroupMap.empty forward tree ->
    first_leaf tree inp = exec r inp
}
```

Listing 13: Typeclass describing an anchored engine

To describe unanchored engines, we first define the lazy prefix[65] as simply being the sequencing of `/[^]*?/` with a regex `r`. Then, the typeclass definition of an unanchored engine seen in Listing 15 is very similar to that of an anchored

[65] A regex construct that matches any sequence of characters in a non-greedy manner. It is of the form `/[^]*?/`. It is prepended to a regex `r` to find a match for `r` anywhere in the haystack.

engine. The sole difference is that for the correctness axiom, we require that the engine, when given a regex `r` and a haystack `s`, returns the same result as the one defined by the semantics of a backtracking tree for `/[^]*?r/`! All typeclass member names are prefixed with `un_`.

```
Definition dot_star : regex :=
  lazy_star (Regex.Character CdAll)

Definition lazy_prefix (r:regex) : regex :=
  Sequence dot_star r
```

```
Class UnanchoredEngine := {
  un_exec: regex -> input -> option leaf;

  (* asserts the supported subset of regexes *)
  un_supported_regex: regex -> bool;

  (* the execution follows the backtracking tree semantics *)
  un_exec_correct: forall r inp tree,
    un_supported_regex r = true ->
    is_tree rer [Areg (lazy_prefix r)] inp Groups.GroupMap.empty forward tree ->
    first_leaf tree inp = un_exec r inp
}
```

Given those definitions we can now exhibit some instances of those typeclasses. Naturally, the anchored PikeVM defined in Section 4.1 is an instance of the anchored engine typeclass[68]. Its proof of `exec` correctness is derived from proofs previously present in Linden. We also show that the unanchored PikeVM defined in Section 4.2 is an instance of the unanchored engine typeclass[69]. Its proof of `un_exec` correctness is derived from proofs discussed in Section 4.4. For both PikeVMs, the `supported_regex`[70] predicate notably excludes regexes with backreferences (due to not having a linear-time implementation) and with lookarounds (which have yet to be verified in the PikeVM[71]). Similarly, the already verified MemoBT engine is an instance of an anchored engine[72]. There is currently, however, no verified specialized unanchored version of the MemoBT engine in Linden. Instead, we notice that every anchored engine can be turned into an unanchored one.

As stated in Chapter 4, by running an anchored engine on a regex augmented with the lazy prefix, we can perform unanchored matching. We prove this by providing a generic instance `UnanchorEngine` (Listing 16) which given any anchored engine produces an unanchored one. We must additionally have a precondition that for any regex supported by the anchored regex, its

66 Semantics/Regex.v#139-140

67 Semantics/Regex.v#143-144

Listing 14: The definition of the lazy prefix and the typeclass describing an unanchored engine

Engine/Meta/EngineSpec.v#40-51

Listing 15: The definition of the lazy prefix and the typeclass describing an unanchored engine

68 Proven in Engine/Meta/EngineSpec.v#76-82

69 Proven in Engine/Meta/EngineSpec.v#93-99

70 The PikeVM regex support predicate can be found in Engine/PikeSubset.v#96-107

71 A very recent development [BP24] has found a way to incorporate lookarounds into the PikeVM under the ECMAScript semantics. No mechanization of this fact has been yet completed.

72 Proven in Engine/Meta/EngineSpec.v#110-116

lazy prefix is also supported. With this assumption, the correctness of `un_exec` follows directly from the correctness of `exec` of the anchored engine.

```
Definition lazy_prefix_supported {engine:AnchoredEngine rer} : Prop :=
  forall r, supported_regex rer r = true -> supported_regex rer (lazy_prefix r) = true
```

Engine/Meta/EngineSpec.v#60-61

```
Instance UnanchorEngine {engine:AnchoredEngine rer} (lazy_prefix_supp: lazy_prefix_supported):
 UnanchoredEngine rer := {
  un_exec r inp := exec rer (lazy_prefix r) inp;
  un_supported_regex r := supported_regex rer r;
 }
```

Engine/Meta/EngineSpec.v#66-69

Listing 16: Turning any anchored engine into an unanchored one.

5.2 Literal optimizations

In Chapter 4 we have seen how by using the prefix of an extracted literal of a regex we can accelerate matching by skipping parts of the haystack. We did so by deeply integrating this prefix acceleration optimization into the PikeVM engine. However, we have also discussed that under the assumption that we treat engines as black-boxes, which is what our engine typeclasses effectively are, a limited but nonetheless useful variant of prefix acceleration can be performed. Additionally, in Section 3.1 we have created the theory of both `Impossible` and `Exact` literals, but have yet to leverage them for optimizations. In this section we correct this by formalizing an optimization performing this limited form of prefix acceleration as well as optimizations utilizing those two literal kinds.

5.2.1 One-time prefix acceleration

When wanting to respect the linear runtime complexity of matching, treating an engine as a black-box makes utilizing prefix acceleration difficult. Until now, the best algorithm we know is one where we perform prefix acceleration once at the start, and from that found position run an unanchored engine. This limited optimization is regardless rather useful. For one, it allows us to give prefix acceleration to engines for which we do not know how to deeply integrate it in a way that is more beneficial than this limited variant. Additionally, if this single prefix acceleration fails to find an occurrence of the prefix, we can avoid running the full regex engine entirely and just return that no match exists. We will therefore use this optimization whenever we wish to run an unanchored engine.

We therefore define this simple `search_acc_once` function in Listing 17. It extracts the literal, run a substring search for the prefix of that literal, and if

found runs an unanchored engine from that position onward. If no occurrence is found, it immediately returns with no match.

Engine/Meta/MetaLiterals.v#153-159

```
Definition search_acc_once {strs:StrSearch} {engine:UnanchoredEngine rer} (r:regex)
(inp:input) : option leaf :=
  let p := prefix (extract_literal rer r) in
  (* we skip the initial input that does not match the prefix *)
  match (input_search p inp) with
  | None => None (* if prefix is not present anywhere, then we cannot match *)
  | Some inp' => un_exec rer r inp'
  end
```

Listing 17: Definition of a function performing prefix acceleration once.

To tackle its correctness we first need an intermediate lemma. It concerns itself with what we know about the matching results when the substring search reports no found occurrences. Namely, if no occurrence of the prefix of the literal of a regex is found in the haystack, then no match can exist even when performing unanchored matching. This is formalized in Theorem str_search_none_nores_unanchored.

Engine/Prefix.v#967-971

THEOREM (str_search_none_nores_unanchored) {strs:StrSearch}:

```
forall r inp tree,
  is_tree rer [Areg (lazy_prefix r)] inp Groups.GroupMap.empty forward tree ->
  str_search (prefix (extract_literal r)) (next_str inp) = None ->
  first_leaf tree inp = None
```

Proof. Induction on the position in the haystack. We apply Corollary extract_literal_prefix_contra together with the `not_found` axiom of substring searches[75] to show at each position that a match cannot exist.

[75] Defined in Section 3.2

With that, the correctness of `search_acc_once` is expressed as returning the same result as the one described by the backtracking tree semantics of a regex with a lazy prefix. This is formalized in Theorem search_acc_once_correct.

Engine/Meta/MetaLiterals.v#192-196

THEOREM (search_acc_once_correct) {strs:StrSearch} {engine:UnanchoredEngine rer}:

```
forall r inp tree,
  un_supported_regex rer r = true ->
  is_tree rer [Areg (lazy_prefix r)] inp Groups.GroupMap.empty forward tree ->
  first_leaf tree inp = search_acc_once r inp
```

Proof. If the substring search returns `None`, proof follows from Theorem str_search_none_nores_unanchored. Otherwise, by induction over the haystack positions before the found occurrence we conclude using the `no_earlier` axiom of substring searches[76] and Corollary extract_literal_prefix_contra.

[76] Defined in Section 3.2

5.2.2 `Exact` and `Impossible` literals

To take advantage of the `Exact` and `Impossible` literals, we define the function `try_lit_search` as seen in Listing 18.

```
Definition try_lit_search {strs:StrSearch} (r:regex) (inp:input) : search_result :=
  match extract_literal rer r with
  | Prefix s => Unsupported
  | Impossible => Ok None
  | Exact s =>
      (* if it has asserts doing a string search is not enough *)
      if has_asserts r then Unsupported
      else
        match input_search s inp with
        | Some inp' =>
            (* if it has groups we must reconstruct them *)
            (* LATER: do group reconstruction with an anchored engine *)
            if has_groups r then Unsupported
            else Ok (Some (advance_input_n inp' (length s) forward, Groups.GroupMap.empty))
        | None => Ok None
        end
  end
```

Engine/Meta/MetaLiterals.v#53-69

Listing 18: Definition of the function which attempts to use literals to find definitive matches.

Other than taking as arguments the regex and a haystack, it additionally expects to have an instance of a substring search algorithm defined in Section 3.2. Notably, no engine is passed as an argument as here we focus on optimizations not requiring them. Its return type is a doubly-nested optional result of matching. The outer option indicates whether this function is able to find the match at all. The inner option indicates whether a match exists. Thus while `None` means that the `try_lit_search` function could not determine what the value of the match is, `Some None` means that the match was determined to not exist. This function starts by extracting the literal of the given regex. If that literal is a `Prefix`, without an engine we cannot determine the match so we return `None`. In the case of `Impossible`, we can immediately return that no match exists, `Some None`. But if the literal is an `Exact s`, more cases must be considered.

If the regex additionally has any assertions, we cannot leverage `Exact` literals because typically substring search algorithms do not support assertions. In case assertions are present, we thus exit with `None`. An assertion is a regex construct which does not consume any characters during matching but instead inspects the surrounding context of the match. Anchors and lookarounds precisely constitute assertions. We define `has_asserts` in Listing 19. Consider the regex `/\babc/` for which we extract the literal `Exact "abc"`. This is what we want; a match of this regex will always be exactly the string "abc". However, due to the word boundary assertion (`\b`) at the start, we cannot simply search for "abc" in the haystack. We must also ensure that at the position where "abc" is found, a word boundary exists. Without an engine to check this assertion,

we cannot proceed and thus return `None`. Luckily, this `Exact` literal will be leveraged during prefix acceleration which will exactly find the instances of "abc" while verifying the assertions.

Engine/Prefix.v#986-992

```
Fixpoint has_asserts (r:regex) : bool :=
  match r with
  | Lookaround _ _ | Anchor _ => true
  | Sequence r1 r2 | Disjunction r1 r2 => has_asserts r1 || has_asserts r2
  | Group _ r' | Quantified _ _ _ r' => has_asserts r'
  | Regex.Character _ | Epsilon | Backreference _ => false
  end
```

Listing 19: Function checking whether a regex contains assertions.

Once we know no assertions are present, we can proceed to using the substring search algorithm to find the first occurrence of the string s in the haystack. If no such occurrence exists, we can return with `Some None`. Otherwise, we have found a position which we know corresponds precisely to the leftmost-greedy match. However, if the regex contained any captures, we would have to additionally enter a "*capture reconstruction*" phase to determine the values of each capture. Capture reconstruction has not been verified in this work, so for now we will exit with `None` if any captures are present. To check for captures, we define the function `has_groups` in Listing 20.

Engine/Meta/MetaLiterals.v#22-28

```
Fixpoint has_groups (r:regex) : bool :=
  match r with
  | Group _ _ => true
  | Sequence r1 r2 | Disjunction r1 r2 => has_groups r1 || has_groups r2
  | Quantified _ _ _ r' | Lookaround _ r' => has_groups r'
  | Regex.Character _ | Epsilon | Backreference _ | Anchor _ => false
  end
```

Listing 20: Function checking whether a regex contains captures.

Finally, if no captures are present we can return a match consisting of the haystack advanced to the position found by the substring search along with an empty group map assignment.

Correctness. Before we state the correctness theorem of `try_lit_search`, we must prove an intermediate lemma. It states that the value of group maps under the assumption that no captures are present in the regex. For the same reason as in Section 3.3, we need to generalize the result over the list of tree actions. As such, we define `has_groups_action` which for the regex action it delegates to `has_groups` and for the `Aclose` action returns true since it corresponds to a capture being closed. For the last action we return false. We intuitively extend this definition to a list of actions. Both definitions are given in Listing 21.

```
Definition has_groups_action (a:action) : bool :=
  match a with
  | Areg r => has_groups r
  | Acheck _ => false
  | Aclose _ => true
  end
```
Engine/Meta/MetaLiterals.v#30-35

```
Fixpoint has_groups_actions (acts:list action) : bool :=
  match acts with
  | [] => false
  | a::t => has_groups_action a || has_groups_actions t
  end
```
Engine/Meta/MetaLiterals.v#37-41

Listing 21: Definitions checking whether a list of actions contains captures.

With that we state the lemma of empty group maps in Lemma no_groups_empty_gm. It states that given a backtracking tree `tree` of actions `acts`, if no captures are present in `acts` and `tree` contains a match, this match's group map is empty.

Engine/Meta/MetaLiterals.v#80-85

```
Lemma (no_groups_empty_gm) :
  forall acts inp gm dir tree leaf,
    has_groups_actions acts = false ->
    is_tree rer acts inp gm dir tree ->
    tree_res tree gm inp dir = Some leaf ->
    snd leaf = gm
```

Proof. Follows from induction over `is_tree`.

Having this, we formulate the correctness theorem of `try_lit_search` in Theorem try_lit_search_correct. If `try_lit_search` returns `Some`, then the contained result corresponds exactly to the result defined by the backtracking tree semantics of the regex with a lazy prefix. We are not interested in the case where `try_lit_search` returns `None` as that indicates that no optimization was possible.

Engine/Meta/MetaLiterals.v#115-119

```
Theorem (try_lit_search_correct) {strs:StrSearch}:
 forall r inp tree ol,
   is_tree rer [Areg (lazy_prefix r)] inp Groups.GroupMap.empty forward tree ->
   try_lit_search r inp = Ok ol ->
   first_leaf tree inp = ol
```

Proof. If the extracted literal is `Impossible`, proof follows from Theorem extract_literal_impossible. If the extracted literal is `Exact` and we have no assertions, we split into two cases depending on the result of the substring search.

1. The result is `None` – proof follows from Theorem str_search_none_nores_unanchored.
2. The result is `Some` – proof follows from Lemma no_groups_empty_gm and a missing lemma about `Exact` literals not completed on time in this work.

5.3 Anchored optimization

Besides using regexes to search through large amounts of text, another common use-case is to use them to validate the format of user inputs such as form fields, passwords, or simple syntaxes. In such scenarios, the regex is expected to describe the input in its entirety. For example, the regex `/^\d{4}-\d{2}-\d{2}$/` which validates date strings in the format of "YYYY-MM-DD" is *anchored* from both ends with the `^` and `$` symbols. They respectively require the matching to start at the beginning of the haystack and end at the end of the haystack. Lack of these anchors would allow the regex to match in an arbitrary place in the haystack which would not be desirable for input validation. Regexes which require a `^` for matching are called *anchored* regexes. This is related to the notion of anchored matching as both can only match at the current haystack position. In addition, anchored regexes cannot be made unanchored by the addition of a lazy prefix: `/[^]*?^r/` will fail to match at any haystack position other than the very first one. Thus, for any anchored regex r, the result of performing anchored matching is the same as the result of performing unanchored matching.

This exact observation allows us to implement an optimization for anchored regexes. If the regex is anchored, we only attempt to match at the beginning of the haystack, regardless of whether we are performing anchored or unanchored matching. On top of that, we can perform this optimization by just using anchored engines. We start our formalization by defining the simple static analysis which determines whether a regex is anchored or not, seen in Listing 22.

```
79Fixpoint is_anchored' (r:regex) : bool :=
    match r with
    | Anchor BeginInput => true
    | Disjunction r1 r2 => is_anchored' r1 && is_anchored' r2
    | Sequence r1 r2 => is_anchored' r1 || is_anchored' r2
    | Group _ r1 => is_anchored' r1
    | Quantified _ min _ r1 => (min != 0) && is_anchored' r1
    (* only positive lookaheads can potentially cause the regex to be anchored *)
    (* because then during matching the anchor is only ever tested on the current *)
    (* or upcoming input positions *)
    (* negative lookarounds do not tell us anything about being anchored *)
    (* an anchor in positive lookbehinds could extend before the current input position *)
    (* like in the regex `(?<=^.+)r` *)
    | Lookaround LookAhead r1 => is_anchored' r1
    | Anchor _ | Lookaround _ _ | Epsilon | Regex.Character _ | Backreference _ => false
    end

80Definition is_anchored (r:regex) : bool :=
    if RegExpRecord.multiline rer then false
    else is_anchored' r
```

79 Engine/Meta/MetaAnchored.v#26-41

80 Engine/Meta/MetaAnchored.v#44-46

Listing 22: Definition of anchored regexes.

For a regex to be anchored, it does not necessarily need to start with the `^` anchor. It suffices that the `^` anchor is **required** to be fulfilled for any successful match. This means that in our analysis for the case of a sequence, if either side is anchored, the entire sequence is anchored. On the other hand, for a disjunction both branches must be anchored. Consider `/^a|b/`, the first branch is indeed anchored, but the second branch could potentially match at any haystack position, so we cannot deem that regex anchored. A quantifier is anchored only if its body is anchored and the quantifier has a minimal iteration count greater than zero. This is to ensure that regexes like `/(^a)*b/` are not considered anchored since `*` allows for the body to be matched zero times. The last interesting case is that of lookarounds. A positive lookahead is indeed anchored if its body is anchored, but that does not hold true for positive lookbehinds! Consider `/(?<=^a+)b/`. The inside of the lookbehind is anchored, but the entire regex fails to match at "ab" yet succeeds at "ab", meaning the regex is not anchored. Negative lookarounds do not contribute to anchoring since they only assert the non-existence of a match. Lastly, we must guard the entire analysis on the `multiline` flag (see Section 2.1 Flags). When the flag is enabled, the anchor `^` is allowed to additionally match the beginning of lines, not just the beginning of the haystack. This means that regexes like `/^a/m` could match at multiple haystack positions, so we cannot consider them anchored.

We now define the anchored optimization in Listing 23. Given a regex, a haystack, and an anchored engine, if possible we return the result of **unanchored** matching. The return type is a doubly nested option for the same reason as in Section 5.2.2. Other than checking if the regex is anchored, we additionally check if the passed haystack is set to the beginning position. If it is, we return the result obtained from running the anchored engine. If it is not at the beginning position, we immediately know matching would fail and thus return `Some None`.

Engine/Meta/MetaAnchored.v#49-55

```
Definition try_anchored_search {engine:AnchoredEngine rer} (r:regex) (inp:input) :
search_result :=
  if is_anchored r then
    if pref_str inp == [] then
      Ok (exec rer r inp)
    else
      Ok None
  else Unsupported
```

Listing 23: Anchored search optimization definition.

Correctness. As before, we first need auxiliary definitions which generalize anchor detection to tree actions.

81 Engine/Meta/MetaAnchored.v#59-64

```
Definition is_anchored_act (act:action) : bool :=
  match act with
  | Areg r => is_anchored' r
  | Acheck _ => false
  | Aclose _ => false
  end
```

82 Engine/Meta/MetaAnchored.v#67-71

```
Fixpoint is_anchored_acts (acts:list action) : bool :=
  match acts with
  | [] => false
  | a :: rest => is_anchored_act a || is_anchored_acts rest
  end
```

Listing 24: Definitions checking whether an action and a list of actions is anchored.

Then, we say that given an anchored list of actions and a haystack at a position different from the beginning, no match can exist. This is formalized in Lemma is_anchored_match_not_begin. Recall that in Linden, inputs (which represent haystacks along with the current positions) are formed from a string of future characters and a string of past characters. So to say that a haystack is set to a position different from the beginning, it is the same as saying the string of past characters is non-empty. We represent it here by explicitly forming the Linden input type with the past characters being constructed by the List's `cons` (`::`) constructor which is by definition non-empty.

Lemma (is_anchored_match_not_begin) : Engine/Meta/MetaAnchored.v#88-93

```
forall acts c next pref tree gm,
  RegExpRecord.multiline rer = false ->
  is_anchored_acts acts = true ->
  is_tree rer acts (Input next (c::pref)) gm forward tree ->
  tree_res tree Groups.GroupMap.empty (Input next (c::pref)) forward = None
```

Proof. By induction over `is_tree`.

We specialize the lemma to the case of regexes.

Corollary (is_anchored_match_not_begin_regex) : Engine/Meta/MetaAnchored.v#136-140

```
forall r c next pref tree,
  is_anchored r = true ->
  is_tree rer [Areg r] (Input next (c::pref)) Groups.GroupMap.empty forward tree ->
  first_leaf tree (Input next (c :: pref)) = None
```

Proof. This holds directly from Lemma is_anchored_match_not_begin with `acts = [Areg r]`.

And we conclude with the correctness theorem of `try_anchored_search` in Theorem try_anchored_search_correct. If `try_anchored_search` returns `Some`, then the contained result corresponds exactly to the result defined by the backtracking tree semantics of the regex with a lazy prefix. We are not interested in the case where `try_anchored_search` returns `None` as that indicates that no optimization was possible.

Theorem (try_anchored_search_correct) {engine:AnchoredEngine rer}: Engine/Meta/MetaAnchored.v#173-178

```
forall r inp leaf tree,
  supported_regex rer r = true ->
  is_tree rer [Areg (lazy_prefix r)] inp Groups.GroupMap.empty forward tree ->
  try_anchored_search r inp = Ok leaf ->
  first_leaf tree inp = leaf
```

Proof. By the correctness of the anchored engine and by Corollary is_anchored_match_not_begin_regex.

5.4 Meta search

With all of the theory of engines, the correctness proofs of optimizations, the formalization of literals and substring searches, we can finally put everything together into a single unified regex engine which will leverage all of the mentioned components with the intention of being as efficient as possible. This Meta engine will be split into two. A smaller anchored meta engine which is only focused on picking the best engine for the job. Then, a larger unanchored Meta engine will take advantage of all of the components laid out in the previous chapters and sections.

To guide heuristics, we define a configuration type which holds parameters which influence the decisions made by the Meta engine. For now, we only store the memory limit we wish to impose on the engines during runtime. If an engine can potentially go over that limit, it will not be considered for execution. If no memory limit is provided, it is assumed that there is no limit on the memory usage. We express this limit with a natural number n and interpret it as an abstract memory usage unit. It serves as a configuration knob where a larger number allows for a bigger memory usage. We additionally define the estimated peak usage of the MemoBT as $|r| \cdot |s|$ for a regex r and a haystack s.

```
Record meta_config := {
  (* the memory limit the search should try to respect *)
  (* expressed in a somewhat arbitrary unit, attemps to be the unit of a pointer size *)
  (* if not specified, there is no limit *)
  memory_limit : option nat;
}

Definition memobt_peak_memory_usage (r:regex) (inp:input) : nat :=
  regex_size r * total_length inp
```

83 Engine/Meta/Meta.v#77-82

84 Engine/Meta/Meta.v#89-90

Listing 25: The definition of the Meta engine configuration and the estimate of MemoBT's memory usage.

The anchored Meta engine is simple and its definition can be found in Listing 26. It first checks if the provided memory limit allows for running the MemoBT engine. If yes, we run it. Otherwise, we fall back to the PikeVM. To run an engine we use its `exec` function provided by the `AnchoredEngine` typeclass instance. Running them requires providing additional instances, for instance for the substring search, the PikeVM's seen set implementation, or MemoBT's memoization implementation.

```
Definition meta_search {heuristic:meta_heuristic} (r:regex) (inp:input) : option leaf :=
  match @try_lit_search _ rer BruteForceStrSearch r inp with
  | Ok ol => ol
  | Unsupported =>
    match @try_anchored_search _ _ (heuristic.(pick_anchored) r inp) r inp with
    | Ok ol => ol
    | Unsupported => @un_exec _ _ (heuristic.(pick_unanchored) r inp) r inp
    end
  end
```

Engine/Meta/Meta.v#43-51

Listing 26: Definition of the anchored Meta search function.

The correctness follows directly from the correctness of the individual engines. The heuristic of picking an engine does not affect correctness; regardless which engine was picked, the engine will produce a correct result. Heuristics affect only the practical aspect of performing matching. We state this correctness theorem in . For any configuration, supported regex, and haystack, running the anchored Meta engine produces the same result as defined by the formal

semantics of backtracking trees. `meta_supported_regex` accepts exactly the same regexes as the PikeVM and the MemoBT. Thus, this Meta anchored matching function is another instance of the `AnchoredEngine` typeclass.

We are now ready to define the final, neat, all-encompassing unanchored Meta engine given by `search` seen in Listing 27.

This function first tries to dispatch matching methods that do not require running a full regex engine. As such, it first attempts to find the match using literal optimizations from Listing 18. For lack of a better instance at the moment of writing, we use the naive brute-force substring search algorithm defined in Listing 9. If this attempt is unable to find a match, we reach for the anchored search optimization from Listing 23. The anchored engine used here is the anchored Meta engine defined in Listing 26. If this also does not yield a match, we try the last trick before having to succumb to using a full unanchored engines. We perform prefix acceleration once using the definition from Listing 17. To decide on the underlying engine, we again use the provided memory limit configuration. If the memory limit allows for running the MemoBT engine, we do so by means of the unanchoring technique from Listing 16. This means we run the anchored MemoBT with the lazy prefix prepended to the regex. For the MemoBT we must additionally provide a proof that it indeed supports regexes with the lazy prefix. If the memory limit could potentially be exceeded, we use the prefix-accelerated unanchored PikeVM from Section 4.2.

Engine/Meta/Meta.v#135-142

```
Definition search (config:meta_config) (r:regex) (inp:input) : option leaf :=
  @meta_search {|
    meta_supported_regex := is_pike_regex;
    pick_anchored := pick_meta_anchored config;
    anchored_supported := pick_meta_anchored_supported config;
    pick_unanchored := pick_meta_unanchored config;
    unanchored_supported := pick_meta_unanchored_supported config;
  |} r inp
```

Listing 27: Definition of the unanchored Meta search function.

We conclude with proving the correctness of this unanchored Meta engine in . For any configuration, supported regex, and haystack, running the unanchored Meta engine produces the same result as defined by the formal semantics of backtracking trees for a regex with the lazy prefix. This, of course, is another instance of an `UnanchoredEngine`[85].

[85] Proven in Engine/Meta/Meta.v#145-149.

6 Evaluation

To motivate the choice of optimizations for formalization, in Section 6.1 we show that the decisions were guided by the frequency of appearance of certain regex patterns in the wild. Then, in Section 6.2 we evaluate the performance benefits of prefix acceleration by comparing three different strategies in Rust's `regex` engine. The strategy proven to be correct in this work was not present in Rust's `regex` before. In some of our benchmarks, it outperforms the existing strategy by up to 20×.

6.1 Frequency of patterns in the wild

When deciding which optimizations to formalize, it is important to consider which parts need optimizing. One obvious approach is to optimize parts which are slow in practice. This is something that is constantly being done by regex engine developers and researchers. But it is also important to be informed by what kind of regexes are being written by people and used in practice. By analyzing them we can get an idea of which patterns are common and thus worth optimizing for. Since formal verification is a time-consuming process, we focus on optimizations which are most likely to yield performance improvements for real-world regexes.

For that we analyze the large corpora of regexes collected in [Dav+18, Dav+19]. It consists of 1 755 587 regexes scraped from NPM and PyPi packages, StackOverflow posts, RegexLib, and more. To parse them we use the parser from RegElk [BP24], a linear engine and parser for ECMAScript regexes written in OCaml. Once parsed, we analyze the resulting Abstract Syntax Trees (ASTs)[86] to look for common patterns. From the regexes in the corpora, 87.50% were successfully parsed and analyzed. The rest either contained unsupported features or were malformed. One data point which we do not have is which flags were enabled for each regex. In the analysis we thus assume no flags were enabled. We believe this is a reasonable assumption to make since people tend to leave things at their defaults. For regexes, you must opt in to enable a flag.

[86] A tree representation of the syntactic structure of some source code. Usually some redundant information is omitted, hence the "abstract" part in the name.

Figure 3 shows the frequency of occurrence of patterns among the successfully parsed regexes. We can see that a significant portion (66.10%) of regexes contain meaningful literals that can be extracted. Of those, 90.50% can already benefit from the optimizations formalized in this work. We do not count literals such as `Prefix ""` which give no useful information. Regexes anchored to the start with `^` are also quite common (23.07%), and thus the optimization for them is also likely to be beneficial in practice. We also note that there are no regexes for which an `Impossible` literal could be extracted. This is expected: we do not anticipate people to write regexes that cannot produce matches. But since adding support for `Impossible` literals was simple, we still formalized it in this work. Additionally, we believe that with the help of some rewrites (as noted in margin note[24]) the number of `Impossible` instances found in regexes will grow, primarily from people accidentally creating unmatchable regexes.

Pattern	Occurrence
Extractable literals	66.10%
| Front-only	11.63%
| Back-only	4.90%
| Front and back	49.58%
| Impossible literals	0.00%
| Exact literals	25.58%
| | With no asserts	14.37%
| | | And no captures	12.98%
Offseted literals	4.41%
`^` anchored	23.07%
`$` anchored	21.51%
`^` and `$` anchored	12.69%
No captures	62.09%

Figure 3: Occurrences of various regex patterns in the regexes from the corpora, with some rows representing features that are exploited for optimizations in this work and other rows representing features that have only a partial implementation of optimizations in this work.

Exact literals were extracted from 25.58% of the regexes. However, only for 56.16% of them we can perform exact literal optimization. That is because the remaining fraction of regexes contain assertions (lookarounds or anchors) preventing us from optimizing them. Of those regexes with exact literals and no assertions, 9.64% have capture groups forcing us to enter the capture reconstruction stage. We believe, however, that most if not all of those regexes that have an exact literal and captures, have captures by mistake. A capture in an exact literal regex is not useful, as the value of those captures will be always the same across all possible matches. By examining those regexes we have identified a common source of mistake where the use of a capture was accidental and the literal characters "(" and ")" were intended to be matched instead. Examples include `/header('Content-Location') /`, `/created_at > NOW()/`, and `/"CHARACTER VARYING({0})"/`. For each of those, we suspect the parenthesis were intended to be escaped yielding `/header\('Content-Location'\) /`, `/created_at > NOW\(\)/`, and `/"CHARACTER VARYING\(\{0\}\)"/` respectively.

Some entries in the figure represent optimizations which are closely related to those implemented in this work, but are missing a full formalization. This notably includes offseted literals and back optimizations. Completing them requires some additional work which can use the foundations laid in this work. The figure additionally mentions that a very large portion of regexes contain no captures at all (62.09%). This motivates a different line of work, namely around new regex engines. All of those potential extensions are discussed in Section 7.2.

6.2 Prefix acceleration in Rust's `regex`

The `regex` crate [And16] is the official regex library for the Rust programming language. It focuses on providing an efficient and safe implementation of regex engines. Only regex features for which we know a linear-time implementation are supported. Similarly to RE2 [RP06], whose architecture served as inspiration for the `regex` crate, it implements a variety of regex engines which are then orchestrated by a single *meta* engine. It consistently ranks among the fastest linear-time regex engines in benchmarks. It achieves it by employing a large variety of heuristics and optimization which perform well in practice. One of the crucial optimizations it employs is prefix acceleration. In this section we benchmark three different strategies for prefix acceleration using the rebar [And23b] benchmarking tool. The benchmarking results underline the importance of this optimization by achieving speedups of up to 600× compared to no prefix acceleration at all. Additionally, we implement our prefix acceleration strategy that has been proven correct in this work and show that in some benchmarks it outperforms the existing implementation by up to 20×.

The three prefix acceleration strategies we compare are:

1. Our prefix acceleration presented in Chapter 4.
2. The existing prefix acceleration implemented in Rust's `regex` crate. It is similar to ours, except it does not perform the filtering optimization, only the acceleration is performed.
3. Performing prefix acceleration a single time at the start of matching. It is the strategy described in Section 5.2.1. It is the best known prefix acceleration strategy under the black-box assumption.

All of these strategies are implemented in the PikeVM of the `regex` crate. As baseline, we also include results for no prefix acceleration at all.

Experimental setup. We modify the `regex` crate and implement the two other prefix acceleration strategies that were not previously present. This fork can be found under https://github.com/LindenRegex/rust-regex/tree/d2c210eec94405630e3e7ebba6d6d1b19e5dc85f. We use the rebar [And23b] benchmarking tool. Rebar is a benchmarking framework and collection of benchmarks for regex engines. It serves as a trusted source of performance comparisons. We modify the rebar repository to add our modified `regex` crate and enable only benchmarks that test the PikeVM directly. This fork can be found under https://github.com/LindenRegex/rebar/tree/72213a4ce5ff4a804ca7a542e665c4c81f2dbeed. The benchmarks were run on an idle Macbook Air M1 with 16GB of RAM. The following commands were used to produce the results:

```
rebar build -e 'rust/regex/pikevm/(?:noAcc|accOnce|accEmptyStates|accOneAhead)'
rebar measure -e 'rust/regex/pikevm/(?:noAcc|accOnce|accEmptyStates|accOneAhead)' | tee
prefilters.csv
```

Results. The full report of the results can be found at https://github.com/LindenRegex/rebar/blob/72213a4ce5ff4a804ca7a542e665c4c81f2dbeed/prefilters.md. The summary of the results is presented in Figure 4.

Strategy	Geometric mean of speed ratios
Our strategy	**1.03**
Only acceleration (existing implementation)	1.06
Performing prefix acceleration once	1.18
No prefix acceleration	2.42

Figure 4: The summary of all the results of the benchmarks comparing different prefix acceleration strategies. Lower is better.

This summary gives us an idea of the average performance over a large variety of regexes and haystacks. We first observe that performing even just a single time prefix acceleration yields a significant speedup. This can be attributed to cases where the haystack is large and prefix acceleration reports that no occurrences were found. In these cases we skip running the PikeVM entirely, leading to large speedups. The potential speedup reach upwards of 600×, which shows the relative speed of substring search algorithms compared to the PikeVM.

We also see that our strategy performs the best on average. In a few of the worst cases it performed 2× worse than the existing implementation, but in the rest it either performed slightly better or up to 20× better. This shows that the filtering optimization is indeed beneficial in practice. We attribute

these regressions to the fact that our strategy is not streaming. We always maintain a counter to the next prefix position. But in some cases the matching could end significantly earlier, leading to wasted work.

The strategy which `regex` currently implements, one where we accelerate multiple times, performs strictly better than doing acceleration just once. This is expected, as performing acceleration does not incur a meaningful overhead. In the benchmarks we observe up to 80× speedups. The degenerative case for one-time acceleration is when the prefix appears very early in the haystack. In that case we skip only a small portion of the haystack. By performing acceleration multiple times we can skip more of the haystack in future accelerations.

These results show the undeniable value of prefix acceleration, even in its simplest form. The strategy developed in this work would be a beneficial contribution to Rust's `regex` crate. It could be an opt-in strategy for users who prioritize speed over streaming characteristics.

7 Discussion

7.1 Related work

[pạṇ] implements a regex matching library in the Lean theorem prover. It supports the same set of modern regex features as the PikeVM in this work. As opposed to our work, their algorithms are directly executable thanks to the Lean compiler. While the Lean implementation contains a proof of soundness and completeness of its matching algorithms, it does not contain a proof that the returned result is the highest priority one[87], but just one of them. In contrast, our work expresses correctness theorems in terms of the highest priority match. That work also formalizes a prefix acceleration strategy [88] different from ours. They extract a set of characters with which a match could potentially start with. This acceleration is not integrated in the matching algorithms directly, but rather sit on top of them leading to less opportunities for them to be useful. Additionally, while extracting multiple prefixes (rather than just one like in our work) is beneficial, extracting such short prefixes (1 character long) could potentially lead to a large number of false positives during acceleration.

[87] See https://github.com/pandaman64/lean-regex/blob/6756b6292659ba672644767a855156df5415999a/correctness/RegexCorrectness/Spec.lean#L26.

[88] See https://github.com/pandaman64/lean-regex/blob/6756b6292659ba672644767a855156df5415999a/regex/Regex/Regex/OptimizationInfo.lean

[CLM25, ZVE24] formalize modern regex matching algorithms, in Rocq and Lean respectively. Both formalizations support lookarounds in their linear-time engines. However, neither of them support capture groups. The engine algorithms differ from the PikeVM, they are respectively based on Marked Regular Expressions and Regex Derivatives. Neither of them formalize backtracking semantics and have no notion of priority, thus are different semantics from those of this work.

7.2 Future work

The most important item which unfortunately constitutes future work, is finishing the theorem about `Exact` literals. It is the only theorem that has been assumed to be true in the entire Rocq formalization of this work. We believe completing this proof should not require a large amount of effort, but of course requires time nonetheless. We state the assumed theorem in . Given a regex with no asserts and a haystack, the matching position of the regex on this haystack is determined by a substring search. Here, we do not say anything about the values of the group map.

Other kinds of optimizations can be built directly on top of the formalizations presented in this work. We discuss three such extensions which can benefit the most from the work already completed.

Offseted literals. The optimization targets regexes in which we can extract literals that are preceded by some fixed amount of unknown characters. For example, consider `/(.\w|\D{2})abc+/`. Normally literal extraction for that regex would return `Prefix ""`, giving us no useful information. However, we can notice that the constant string "abc" which must be always present in a match is preceded by exactly two unknown characters. One could use this information to perform prefix acceleration by searching for the string "abc" and returning its position minus two. This work lays the groundwork needed to formalize offseted literals. Careful consideration of the incurred runtime complexity is needed. This optimization was not chosen for this work due to it affecting a rather small portion of regexes (4.41%) as analyzed in Section 6.1.

Back optimizations. A dual to the optimizations formalized in this work are optimizations that analyze the back of the regex rather than the front. In Figure 3, *front-only* pattern refers to regexes from which a literal can be extracted from the front (which is what our literal extraction phase

targets) but none from the back. On the other hand *back-only* refers to regexes where a literal can be extracted from the back but none from the front. A *back* literal can be similarly exploited like *front* literals. Consider the regex `/[A-Z][a-z]+ likes cats/` whose literal could be `Suffix " likes cats"`. A *suffix* acceleration strategy could be employed which performs searches with the **reverse** of the regex on the **reverse** of the haystack. In a similar fashion, end-of-input anchors `$` could be also optimized. To complete this dual class of optimizations, a formalization of regex reversal is needed which leads to non-trivial semantic challenges, notably with captures. [BP24] tackles some of these challenges by, among other things, performing capture reconstruction.

Multiple-literal extraction. In this work we have extracted a single literal for a given regex. In practice, it can be beneficial to extract multiple. Consider the regex `/alpaca|llama/`. Our literal extraction would return `Prefix ""`, leading to no useful optimizations. Instead, one could extract a list of literals, `[Exact "alpaca", Exact "llama"]`. Using established algorithms such as Aho-Corasick [AC75], one could use a substring search algorithm that searches for multiple literals at once. This class of algorithms also performs significantly better than full regex engines. Extracting multiple literals also allows us to support case-insensitive searches. However, this change requires very careful consideration. A large amount of heuristics must go into the process to establish the balance between the number of extracted literals and the length of the strings in the extracted literals. Should we really be extracting 52 literals for `/[a-z]/i`? Rust's `regex` crate does multiple-literal extraction and is full of heuristics guiding the process.

Finally, the last class of future work is related to implementing and verifying more regex engines. These could be directly integrated into the Meta engine. As seen in Figure 3, 62.09% of regexes do not use captures. For such regexes, we can use a specialized engine which forgoes supporting captures in order to gain significant performance improvements. We believe that such an engine would lead to the greatest benefit for practical regex matching.

7.3 Conclusion

Working with realistic regexes that follow an official standard (ECMAScript 2023) rather than toy examples, we have formalized multiple optimizations that significantly speed up regex matching in practice. We back our choices

of optimizations by looking at real-world usage of regexes found in the wild and by benchmarking the novel prefix acceleration strategy in Rust's `regex` library. Our results are fully mechanized in the Rocq proof assistant and serve as a stepping stone for future, additional optimizations. Prefix acceleration reduces the places which must be explored in the haystack, impossible regexes immediately reject haystacks, anchored regexes allow running regex engines only at a single haystack position, exact literals allow using just a substring search to find matches, and finally a meta engine hides all of the heuristics and optimizations behind a single convenient function to perform matching. The continuous effort of the Linden project shows that we can embrace the complexity of modern regexes and prove meaningful results about them.

Bibliography

[And21] Andrew Gallant, "some regexes fail to satisfy that (re)+ is always equal to (re)(re)*." [Online]. Available: https://github.com/rust-lang/regex/issues/779

[Reg25] "RegExp: Inconsistent branch priority in look-behind assertions.." [Online]. Available: https://issues.chromium.org/issues/388290816

[Joh19] John Graham-Cumming, "Details of the Cloudflare outage on July 2, 2019." [Online]. Available: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/

[SP18] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 361–376. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[BDP26] A. Barrière, V. Deng, and C. Pit-Claudel, "Formal Verification for JavaScript Regular Expressions: A Proven Mechanized Semantics and Its Applications," *Proc. ACM Program. Lang.*, vol. 10, no. POPL, Jan. 2026, doi: 10.1145/3776710.

[DBP24] N. De Santo, A. Barrière, and C. Pit-Claudel, “A Coq Mechanization of JavaScript Regular Expression Semantics,” *Proc. ACM Program. Lang.*, vol. 8, no. ICFP, Aug. 2024, doi: 10.1145/3674666.

[ECM23] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 14th ed. 2023. [Online]. Available: https://262.ecma-international.org/14.0/index.html

[AM99] Abigail and Mark Jason Dominus, “Reduction of 3-CNF-SAT to Perl Regular Expression Matching.” [Online]. Available: https://perl.plover.com/NPC/NPC-3SAT.html

[KR87] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987, doi: 10.1147/rd.312.0249.

[Wan+19] X. Wang *et al.*, “Hyperscan: a fast multi-pattern regex matcher for modern CPUs,” in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, in NSDI'19. Boston, MA, USA: USENIX Association, 2019, pp. 631–648.

[Rus09] Russ Cox, “Regular Expression Matching: the Virtual Machine Approach.” [Online]. Available: https://swtch.com/~rsc/regexp/regexp2.html

[And16] Andrew Gallant, “Rust regex.” [Online]. Available: https://github.com/rust-lang/regex

[RP06] Russ Cox and Paul Wankadia, “RE2, a regular expression library.” [Online]. Available: https://github.com/google/re2

[V8] V8, “V8's Experimental Linear Regular Expression Engine.” [Online]. Available: https://chromium.googlesource.com/v8/v8/+/main/src/regexp/experimental

[Gol] “Golang's regexp package.” [Online]. Available: https://pkg.go.dev/regexp

[And23] Andrew Gallant, “Regex engine internals as a library.” [Online]. Available: https://burntsushi.net/regex-internals

[And23] Andrew Gallant, “A biased barometer for gauging the relative speed of some regex engines on a curated set of tasks..” [Online]. Available: https://github.com/BurntSushi/rebar

[BP24] A. Barrière and C. Pit-Claudel, "Linear Matching of JavaScript Regular Expressions," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, June 2024, doi: 10.1145/3656431.

[Dav+18] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "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*, in ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 246–256. doi: 10.1145/3236024.3236027.

[Dav+19] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 443–454. doi: 10.1145/3338906.3338909.

[pan] pandaman, "Lean regex." [Online]. Available: https://github.com/pandaman64/lean-regex

[CLM25] A. Chattopadhyay, A. W. Li, and K. Mamouras, "Verified and Efficient Matching of Regular Expressions with Lookaround," in *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, in CPP '25. Denver, CO, USA: Association for Computing Machinery, 2025, pp. 198–213. doi: 10.1145/3703595.3705884.

[ZVE24] E. Zhuchko, M. Veanes, and G. Ebner, "Lean Formalization of Extended Regular Expression Matching with Lookarounds," in *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, in CPP 2024. London, UK: Association for Computing Machinery, 2024, pp. 118–131. doi: 10.1145/3636501.3636959.

[AC75] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Commun. ACM*, vol. 18, no. 6, pp. 333–340, June 1975, doi: 10.1145/360825.360855.

Glossary

AST (Abstract Syntax Tree) A tree representation of the syntactic structure of some source code. Usually some redundant information is omitted, hence the “abstract” part in the name.

Black-box A model where the internal workings are not known by the user. The only interaction with it are possible through its public interface.

Capture (Capturing group) A feature in modern regexes that allows parts of the matched text to be captured by a subpatterns and extracted later. They are annotated using parentheses. For instance, given `/(a(bc))e/`, there are two capture groups: the outer group captures , and the inner group captures . When this regex matches the string “abce”, the first capture group will contain “abc” and the second will contain “bc”.

Crate The name used to describe packages in the Rust ecosystem. Crates are most often published to and downloaded from the crates.io repository.

Engine (Regex matching algorithm) An algorithm used to perform matching of a regex against a haystack. It supports a specific subset of regex features and has some performance characteristics. Examples include the PikeVM, LazyDFA, Backtracking.

Haystack In the context of regex **matching**, the haystack is the input text in which we search for occurrences of patterns defined by regular expressions. When we say that we want to match the regex `/a*b{3}/` against “abc”, the string “abc” is the haystack. Newlines in the haystack are represented with the “⏎” character. Already seen characters in the haystack are underlined “qwerty”. Positions in the haystack are marked with an arrow “qwerty”. Match ranges in the haystack are highlighted “qwerty”.

Lazy prefix A regex construct that matches any sequence of characters in a non-greedy manner. It is of the form `/[^]*?/`. It is prepended to a regex `r` to find a match for `r` anywhere in the haystack.

MemoBT (Memoized backtracker) A regex engine that combines backtracking with memoization to avoid redundant computations. It explores possible matches like a backtracking engine but stores intermediate results to ensure that each unique state is only computed once. It has a $O(|r| \cdot |s|)$ runtime and space complexity.

PC (Program counter) An integer value that indicates the current position of execution within a larger sequence of instructions. Storing it allows resuming execution from that point later.

Preorder A binary relation that is both reflexive and transitive.

ReDoS (Regular expression Denial of Service) An exploit of unfavorable regex matching performance characteristics. When a regex comes from user input, it can be used to attack by crafting a regex which makes matching take exponential time. This most commonly affects backtracking engines which have worst-time exponential runtime.

Regex (Regular expression) A *classical* regex is a pattern describing a regular language. A *modern* regex is a pattern used in programming languages that may include features going beyond regular languages, such as backreferences and lookarounds. It is used to find matches and to extract substrings from text.

SIMD (Single Instruction, Multiple Data) A parallel computing paradigm where a single instruction operates on multiple data points simultaneously. Modern CPUs often support SIMD instructions that can process multiple pieces of data in parallel, speeding up operations by a healthy constant factor.