Formal Verification for JavaScript Regular Expressions: A Proven Mechanized Semantics and Its Applications

AURÈLE BARRIÈRE, EPFL, Switzerland
VICTOR DENG, EPFL, Switzerland and DI ENS, École Normale Supérieure, PSL, CNRS, France
CLÉMENT PIT-CLAUDEL, EPFL, Switzerland

We present the first mechanized, succinct, practical, complete, and proven-faithful semantics for a modern regular expression language with backtracking semantics. We ensure its faithfulness by proving it equivalent to a preexisting line-by-line embedding of the official ECMAScript specification of JavaScript regular expressions. We demonstrate its practicality by presenting two real-world applications. First, a new notion of contextual equivalence for modern regular expressions, which we use to prove or disprove rewrites drawn from previous work. Second, the first formal proof of the PikeVM algorithm used in many real-world engines. In contrast with the specification and other formalization work, our semantics captures not only the top-priority match, but a full *backtracking tree* recording all possible matches and their respective priority. All our definitions and results have been mechanized in the Rocq proof assistant.

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

Additional Key Words and Phrases: Regex, Semantics, Rocq, JavaScript, Formal Verification

ACM Reference Format:
Aurèle Barrière, Victor Deng, and Clément Pit-Claudel. 2026. Formal Verification for JavaScript Regular Expressions: A Proven Mechanized Semantics and Its Applications. *Proc. ACM Program. Lang.* 10, POPL, Article 68 (January 2026), 30 pages. https://doi.org/10.1145/3776710

1 Introduction

Despite sharing a name and some features, the modern regular expressions found in many programming languages and libraries are fundamentally different from traditional regular expressions. For one, with the addition of new features such as backreferences, modern regular expressions (which we call *regexes* in the rest of this paper) no longer correspond to regular languages, nor even to context-free languages [Nogami and Terauchi 2023]. More importantly, with features such as capture groups, the matching problem changes. It no longer amounts to a recognition problem (checking that a string belongs to the language of a regular expression), but instead to a segmentation problem (extracting parts of a string that match individual subexpressions of a regex). For instance, matching the regex `a(b)` on string `"abc"` in JavaScript returns that the whole regex matched substring `"ab"`, and that the subexpression in parentheses matched substring `"b"`. From these features arises a new problem: when there are several ways to match a regex, which substring should be returned? One possible answer is given by the POSIX standard, favoring the *leftmost-longest* match of the regex. Another more popular answer is to prioritize the match that would be found by a backtracking algorithm. Many modern regex languages, such as the

Authors' Contact Information: Aurèle Barrière, aurele.barriere@epfl.ch, EPFL, Lausanne, Switzerland; Victor Deng, victor.deng@epfl.ch, EPFL, Lausanne, Switzerland and DI ENS, École Normale Supérieure, PSL, CNRS, Paris, France; Clément Pit-Claudel, clement.pit-claudel@epfl.ch, EPFL, Lausanne, Switzerland.

ACM 2475-1421/2026/1-ART68
https://doi.org/10.1145/3776710

ones of Perl, JavaScript, Python, Go, Rust, Java, or .NET follow that disambiguation policy, called *backtracking semantics* (sometimes also called leftmost-greedy, or PCRE semantics [Hazel 1997]).

Between the new features and this disambiguation policy, the semantics of these modern regexes has become surprisingly large and complex. ECMAScript [ECMA 2025], the official JavaScript standard, dedicates more than 30 pages to the semantics of JavaScript regexes. This specification is so complex that semantics bugs are still regularly found in regex engines [V8 2023, 2025]; that the JavaScript implementation used in Firefox gave up on writing a regex engine and uses Google Chrome's instead [Ireland 2020]; and in this work we exhibit semantics errors in the optimizer of a real-world JavaScript regex processor, `regexp-tree` [Soshnikov 2025].

In this work, we present a new formal semantics for JavaScript regexes, that is **mechanized, succinct, complete, practical, and proven** to be correct. Despite decades of modern regexes being in wide use (including in more than 30% of JavaScript and Python packages [Davis et al. 2018]), and multiple previous formalization efforts [Chattopadhyay et al. 2025; Chen et al. 2022; Chida and Terauchi 2023, 2024; De Santo et al. 2024; Loring et al. 2019; Zhuchko et al. 2024], this is the first time that a semantics enables practical mechanized verification for a real-world modern regex language. Our semantics is **mechanized** in the Rocq proof assistant (formerly known as Coq). It is **proven** to be faithful to the JavaScript semantics with a verified equivalence to Warblre [De Santo et al. 2024], an existing line-by-line embedding of the ECMAScript regex specification into Rocq that is easily auditable but verbose and impractical. It is **succinct**: the core rules of matching fit in a single page. It is **complete**: it supports all regex features of the 14$^{\text{th}}$ edition of ECMAScript. Finally, it is **practical** for formal verification. We demonstrate this with two applications. First, we present a new formalized definition of contextual equivalence for regexes with backtracking semantics. Second, we verify the PikeVM algorithm, which simulates a NFA (*non-deterministic finite automaton*). This is the first formal verification of this widely used matching algorithm. Interestingly, beyond the usual considerations of mechanized semantics design (convenient induction principles, strict positive occurrences etc.), we find that a key ingredient for a practical regex backtracking semantics is to not follow a backtracking algorithm. Instead, we formalize not only the one match returned by a backtracking algorithm, but instead all possible matches, ordered by priority. We claim the following novel contributions:

- A new succinct, complete, practical **formal semantics**, mechanized in Rocq.
- Evidence that this semantics is **faithful** to the ECMAScript regex standard, by proving it equivalent to the Warblre embedding [De Santo et al. 2024].
- A new **formal definition of equivalence** for JavaScript regexes. We use it to prove the correctness of regex equations from the literature, as well as prove and disprove optimizations used in a real-world JavaScript regex processor.
- The first **formal verification of the PikeVM algorithm**, a linear-time regex matching algorithm used in many real-world regex engines. This is the first time that NFA simulation with capture groups and backtracking semantics comes with a formal proof.

The theorem relating our semantics model to the official ECMAScript standard makes this the first formal verification work directly connected to a real-world regex specification.

Section 2 informally introduces JavaScript regexes. Section 3 presents our new inductive semantics and its properties. Section 4 shows its faithfulness by proving it equivalent to the Warblre embedding. We then present two formally verified applications. First, Section 5 defines our new regex equivalence and our case studies of correct and incorrect regex rewrites. Second, Section 6 presents our formal verification of the PikeVM algorithm. Finally, Section 7 discusses adapting and extending our work to other languages and applications, as well as related and future work.

Regular Expressions:

r	::=	ε	Empty
	|	cd	Character
	|	$r_1 \mid r_2$	Disjunction
	|	$r_1\ r_2$	Sequence
	|	$(_g\ r)$	Capture group
	|	$\langle r \rangle$	Non-capturing group
	|	a	Anchor
	|	$\backslash g$	Backreference
	|	$r\{min, \Delta, \top\}$	Greedy Quantifier
	|	$r\{min, \Delta, \bot\}$	Lazy Quantifier
	|	$(?lk\ r)$	Lookaround

Character descriptors:

cd	::=	$c \mid \cdot \mid [c_0 c_1 - c_2] \mid [\hat{}\, c_0 c_1 - c_2] \mid$ \w | \W | \d | \D | \s | \S | ...

Anchors:

a	::=	^	|	$	Input boundary
	|	\b	|	\B	Word boundary

Lookarounds:

lk	::=	=	Positive lookahead
	|	!	Negative lookahead
	|	<=	Positive lookbehind
	|	<!	Negative lookbehind

Fig. 1. Abstract JavaScript regex syntax

Our Rocq development, consisting of over 15k lines of definitions and proofs, is available as supplementary material. Appendix A of the extended version of this paper [Barrière et al. 2025b] links each individual paper definition to its Rocq counterpart. Some definitions have been simplified in the paper for presentation purposes.

2 Background on JavaScript Regexes

2.1 Syntax

The abstract syntax of JavaScript regexes we use in this work is summarized on Figure 1, with some modifications from concrete JavaScript syntax for presentation and convenience purposes. These differences are as follows. We use ε instead of no string at all for the empty regex. In concrete JavaScript syntax, capture groups are pairs of parentheses that are implicitly associated with an index depending on their position in the regex. Here, we directly annotate each group with its index, g. For instance, `(a(b))` in JavaScript syntax corresponds to $(_1 \mathtt{a}(_2 \mathtt{b}))$. JavaScript also uses syntactic sugar `(?<`*name*`>`r`)` (*named group*) and `\k<`*name*`>` (*named backreference*) to refer to groups using names instead of indices. Group names can refer to one group at most, so there is a direct correspondence from names to indices. To group subexpressions together without defining a capture group, JavaScript uses non-capturing groups, denoted as `(?:`r`)` and for which we use the notation $\langle r \rangle$. These non-capturing groups are here to help parsing, and are not reflected in the AST. Disjunction is right-associative and has the lowest parsing precedence, meaning for instance that a|ab|abc is exactly the same regex as a|⟨ab⟩|⟨abc⟩.

Quantifiers in concrete JavaScript syntax are written $r\{min, max\}$ where max is either a natural number or infinity. This represents any number of repetitions of r between min and max. Since every valid regex quantifier in JavaScript has $min \leq max$, we instead represent quantifier upper bounds with a Δ representing $max - min$, simplifying the semantics. Quantifiers can either be greedy or lazy (the latter by appending a ? in concrete syntax); we represent this with a $\top$ for greedy or a $\bot$ for lazy. JavaScript also defines useful syntactic sugar for some common quantifiers: $r*$ for $r\{0, \infty, \top\}$, $r*?$ for $r\{0, \infty, \bot\}$, $r+$ for $r\{1, \infty, \top\}$, $r+?$ for $r\{1, \infty, \bot\}$, $r?$ for $r\{0, 1, \top\}$, and $r??$ for $r\{0, 1, \bot\}$.

This syntax covers all regexes from the 14th edition of ECMAScript [ECMA 2023].

2.2 Informal Semantics

We summarize here how JavaScript regexes differ from traditional regular expressions.

Character descriptors. There are many ways to describe a set of characters in JavaScript regexes: the following list is not exhaustive. One can write the character directly: any character c matches itself. Then, $\cdot$ matches all characters (except line terminator characters depending on the `s` flag, described below). Character classes match several ranges of characters. For instance `[af−z]` matches either a, or any letter in between f and z. They can also be negated, for instance `[^af−z]` matches any character that cannot be matched by the previous example. Finally, character class escapes match common sets of characters, for instance `\d` matches numerical digits.

Capture groups. Capture groups record the last match of each subexpression inside parentheses. For instance, matching $a({}_1 \cdot *)c$ on the string `"abcd"` returns a match on the substring `"abc"`, and captures the substring `"b"` between indices 1 and 2 in group 1.

Backreferences. A backreference $\backslash g$ matches the content of capture group g again. For instance, the regex $({}_1 \backslash d)\backslash 1$ matches any digit repeated twice, like `"33"`.

Anchors. Anchors enforce a local condition on the current input position, for instance checking for being at the end of the input (with `$`), or being at a word boundary (with `\b`). For instance, `a$` does not match anything in `"ab"` but matches the substring `"a"` in `"ba"`.

Lookarounds. Lookarounds describe a zero-width condition described as a regex. For instance, the regex `a(?= b)` matches any `"a"` only if it is followed by a `"b"`, but the `"b"` itself is not part of the returned substring. Lookarounds can be either positive or negative; negative lookarounds require that there is no match for the subexpression at the current input position. Positive lookarounds can define capture groups. For instance, matching $a(?= ({}_1 b|c))$ on string `"ab"` matches the substring `"a"` and sets capture group 1 to `"b"`. Lookbehinds allow to match the condition backwards. For instance, matching `(?<= Year:)\d+` on string `"Year:2026"` matches the substring `"2026"`.

Backtracking semantics. As ECMAScript uses a backtracking algorithm to describe their semantics, JavaScript regexes follow backtracking semantics. As a result, when there are several ways to match a regex, the leftmost match has priority (for instance, matching `a|b` on `"dba"` returns `"b"`). Then, the left branch of each disjunction has priority over the right one (for instance, matching `a|ab` on `"ab"` returns `"a"`). Finally, greedy quantifiers give priority to a match that has iterated as many times as possible, while lazy ones try to iterate as few times as possible (within what is allowed by the *min* and Δ parameters).

Unanchored and anchored matching. The kind of regex matching we have discussed so far, where a match can start at any position in the input string (but priority is given to the leftmost-starting match), is also known as *unanchored matching*. However, the problem of unanchored matching is typically reduced to the problem of *anchored matching*, where given a regex, an input string and an input position, one needs to find the top-priority match of the regex that starts precisely at this position (but may end at any position). In the ECMAScript specification, unanchored matching is performed by repeatedly trying an anchored matcher at every possible starting position. In implementations, the top-priority unanchored match of a regex r is sometimes obtained by computing the top-priority anchored match of $\odot *?({}_0\, r)$, where $\odot$ is a regex which matches any character [Cox 2009]. The laziness of the star ensures we find the leftmost-starting match, and the capture group with index 0 extracts the start and end positions of that match. As a result, our semantics focuses on anchored matches; in the rest of this paper, *matching* refers to anchored matching.

Top-level API. JavaScript provides several top-level functions to use regexes: we can find an unanchored match with `match`, find all matches with `matchAll`, replace the result of matching in a string with another substring with `replace` etc. All of these functions can be implemented using an anchored matcher and common string manipulation operations.

Regex flags. Regex flags can be used to change the behavior of matching. The Unicode flag `u` changes the underlying alphabet of characters. Then, the three flags `i` (case-insensitivity), `m` (multiline mode for anchors) and `s` (make $\cdot$ match line terminators) affect the way some characters descriptors and anchors are matched. Finally, the other flags affect the behavior of top-level functions and are not directly relevant to our semantics for anchored matching. The `g` flag enables global matching (finding all matches of a regex in a string). With the *sticky* flag `y`, matching a regex on a string starts from the position where a match of that same regex was previously found. When the flag `d` is active, an engine should not only return the substring matched by each capture group, but also the corresponding string indices.

JavaScript semantic peculiarities. While the rules above hold for any regex language with backtracking semantics, JavaScript semantics also differs from other such languages in two ways. First, to prevent infinite repetitions in regexes like $\varepsilon*$, the **Nullable Quantifier** property of JavaScript means that, when $min = 0$, iterations of a quantifier cannot match the empty string. Other languages use different criteria to prevent infinite repetitions. This unique property requires adapting standard matching algorithms (see section 6).

Second, the **Capture Reset** property means that at each quantifier iteration, the values of capture groups inside that quantifier are reset. For instance, matching $\langle(_1\,\mathtt{a})|\mathtt{b}\rangle*$ on the string `"ab"` matches the whole string, but the value of capture group 1 is set to None after the match: the regex is iterated twice, the first iteration defines group 1, but the second iteration resets it.

3 Tree Semantics

3.1 An Inductive Relation for a Backtracking Execution Trace

A key insight of our semantics is to depart from the backtracking algorithm used to specify JavaScript regex semantics in ECMAScript. In contrast, our semantics is an inductive relation that connects a regex and a string to an execution trace of a backtracking algorithm that *would not stop* at the first accepting result. This execution trace is represented with a tree, where each backtracking decision (for disjunctions and quantifiers) is represented as a branching node, and other nodes correspond to some operations of the algorithm, such as reading a character or a backreference, opening or closing a capture group, or checking the validity of an anchor or lookaround.

Figure 2 showcases the backtracking tree of the regex $\langle\mathtt{a}|\mathtt{a}(_1\,\mathtt{b})|\mathtt{a}\rangle\mathtt{bc}$ on the input `"abbc"`. The tree starts by branching on two possibilities: either matching the first or the second branch of the disjunction. On the left, we see the operations corresponding to exploring the first alternative. First, an `"a"` is read, then the disjunction has finished matching. We resume by matching a `"b"`, but then matching fails because in this string it is not possible to read a `"c"` at the current position. The other two branches correspond to the deepest disjunction, and the middle branch does find a match in the string, opening and closing a capture group along the way. Observe that the backtracking tree includes the full third branch, even though a backtracking algorithm would stop after reaching the match in the second branch.

The syntax of these backtracking trees is defined in Figure 3. A Mismatch node means that the algorithm has failed to find a match, for instance because the next character in the input does not correspond to the character descriptor found in the regex, or because some anchor was not satisfied, or because a progress check failed, etc. For Choice nodes, the subtrees are ordered by priority, meaning that the result of a backtracking algorithm corresponds to the leftmost Match leaf of the corresponding tree. Finally, nodes LK lk t_{look} t and LKMismatch lk t_{look} include the entire tree t_{look} corresponding to matching the lookaround lk at the current input position.

Advantages of our semantics. The core of our semantics comprises only 21 rules shown in Figure 4, achieving a convenient succinctness compared to the ECMAScript specification, while providing

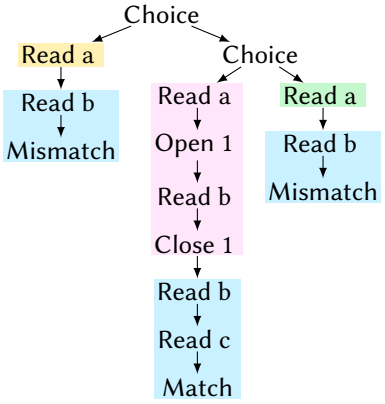


Fig. 2. Backtracking tree t of the regex $\langle \mathtt{a} | \langle \mathtt{a} (_1 \mathtt{b}) | \mathtt{a} \rangle \rangle \mathtt{bc}$ on input "abbc"

Tree:

t	::=	Match	Successful match
	|	Mismatch	Match failure
	|	Choice t_1 t_2	Branching
	|	Read c t	Character read success
	|	BackrefPass s t	Backreference success
	|	Progress t	Progress check success
	|	AnchorPass a t	Anchor success
	|	Open g t	Group opening
	|	Close g t	Group closing
	|	Reset gl t	Group resetting
	|	LK lk t_{look} t	Lookaround success
	|	LKMismatch lk t_{look}	Lookaround failure

Fig. 3. Backtracking Trees

a complete formalization of JavaScript regexes. To match individual characters, we reuse some definitions from the Warblre mechanization, but reasoning on the core matching algorithm can be done without unfolding them. The rules are straightforward to mechanize, in particular we materialize the full tree of lookarounds and scan it for a match, thereby avoiding workarounds for non-strict positivity that are needed in other work (see Section 7). Our semantics also supports the relevant regex flags: although we omit them from the paper definitions, our Rocq definitions are parameterized by the value of the flags `i`, `m` and `s`. To support the Unicode mode (flag `u`), our semantics is parameterized by an alphabet of characters, like Warblre. Although we focus on anchored matching, our semantics could also support the flags used for top-level functions. Essentially, the flags `g` and `y` require changing the inital input position, and we already compute the group indices required by the `d` flag.

More importantly, our semantics also formalizes all possible matches of a regex on an input, not just the top-priority one. Later, we show that this is not only helpful to verify algorithms that explore more paths than a backtracking algorithm (Section 6), but also to reason about contextual equivalence in backtracking semantics (Section 5).

Semantics statement. Figure 4 presents the rules of our semantics. This relation naturally provides an induction principle that we use extensively in our mechanized proofs. We define the big-step relation $(l, i, gm, d) \Downarrow t$, to mean that t is the backtracking tree for the list of actions l, the input i, the group map gm and the reading direction d.

With lookbehinds, it is possible to read the input in reverse. As a result, there are two possible reading directions d: forward ($\rightarrow$) and backward ($\leftarrow$). To support bidirectional reading, the input string i is represented with a zipper (a pair of the list of next characters to consume, and the reversed list of characters already consumed). The relation $i_2 <_d i_1$ on inputs means that input i_1 is the same as i_2 after having read one or more characters following the direction d. We define $\mathsf{idx}(i)$ to be the number of characters already read in i, and $\mathsf{next}(i)_d$ corresponds to i after reading one character.

The list l represents a stack of remaining actions, each of which can take one of three shapes. First, regexes are actions: initially for a regex r, the list of actions is simply $[r]$. Second, Aclose g is the action that closes the group g. For instance, in Figure 2, after reading the first character b in the middle branch, one needs to first close group 1 before matching the rest of the regex. Third, Acheck i means that we need to check that the current input has made some progress compared

$$\frac{}{([], i, gm, d) \Downarrow \mathsf{Match}}\ \textsc{Match} \qquad \frac{(l, i, \mathsf{GM}_{\mathsf{close}}(gm, g, \mathsf{idx}(i)), d) \Downarrow t}{(\mathsf{Aclose}\ g :: l, i, gm, d) \Downarrow \mathsf{Close}\ g\ t}\ \textsc{Close}$$

$$\frac{i_{check} <_d i \qquad (l, i, gm, d) \Downarrow t}{(\mathsf{Acheck}\ i_{check} :: l, i, gm, d) \Downarrow \mathsf{Progress}\ t}\ \textsc{Check} \qquad \frac{\neg(i_{check} <_d i)}{(\mathsf{Acheck}\ i_{check} :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{CheckFail}$$

$$\frac{\mathsf{advance}(cd, i, d) = \mathsf{Some}\ (c, i') \qquad (l, i', gm, d) \Downarrow t}{(cd :: l, i, gm, d) \Downarrow \mathsf{Read}\ c\ t}\ \textsc{Read} \qquad \frac{\mathsf{advance}(cd, i, d) = \mathsf{None}}{(cd :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{ReadFail}$$

$$\frac{(r_1 :: l, i, gm, d) \Downarrow t_1 \qquad (r_2 :: l, i, gm, d) \Downarrow t_2}{((r_1|r_2) :: l, i, gm, d) \Downarrow \mathsf{Choice}\ t_1\ t_2}\ \textsc{Disj}$$

$$\frac{(r_1 :: r_2 :: l, i, gm, \rightarrow) \Downarrow t}{(r_1\ r_2 :: l, i, gm, \rightarrow) \Downarrow t}\ \textsc{SeqForward} \qquad \frac{(r_2 :: r_1 :: l, i, gm, \leftarrow) \Downarrow t}{(r_1\ r_2 :: l, i, gm, \leftarrow) \Downarrow t}\ \textsc{SeqBackward}$$

$$\frac{(l, i, gm, d) \Downarrow t}{(\varepsilon :: l, i, gm, d) \Downarrow t}\ \textsc{Epsilon} \qquad \frac{(r :: \mathsf{Aclose}\ g :: l, i, \mathsf{GM}_{\mathsf{open}}(gm, g, \mathsf{idx}(i)), d) \Downarrow t}{((_g\, r) :: l, i, gm, d) \Downarrow \mathsf{Open}\ g\ t}\ \textsc{Group}$$

$$\frac{\mathsf{check\_anchor}(a, i) = \top \qquad (l, i, gm, d) \Downarrow t}{(a :: l, i, gm, d) \Downarrow \mathsf{AnchorPass}\ a\ t}\ \textsc{Anchor} \qquad \frac{\mathsf{check\_anchor}(a, i) = \bot}{(a :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{AnchorFail}$$

$$\frac{\mathsf{advance\_bkrf}(gm, g, i, d) = \mathsf{Some}\ (s, i') \qquad (l, i', gm, d) \Downarrow t}{(\backslash g :: l, i, gm, d) \Downarrow \mathsf{BackrefPass}\ s\ t}\ \textsc{Backref} \qquad \frac{\mathsf{advance\_bkrf}(gm, g, i, d) = \mathsf{None}}{(\backslash g :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{BackrefFail}$$

$$\frac{(r :: r\{min, \Delta, p\} :: l, i, \mathsf{GM}_{\mathsf{reset}}(gm, \mathcal{G}(r)), d) \Downarrow t}{(r\{min + 1, \Delta, p\} :: l, i, gm, d) \Downarrow \mathsf{Reset}\ \mathcal{G}(r)\ t}\ \textsc{Forced} \qquad \frac{(l, i, gm, d) \Downarrow t}{(r\{0, 0, p\} :: l, i, gm, d) \Downarrow t}\ \textsc{Done}$$

$$\frac{(l, i, gm, d) \Downarrow t_{skip} \qquad (r :: \mathsf{Acheck}\ i :: r\{0, \Delta, \top\} :: l, i, \mathsf{GM}_{\mathsf{reset}}(gm, \mathcal{G}(r)), d) \Downarrow t_{iter}}{(r\{0, \Delta + 1, \top\} :: l, i, gm, d) \Downarrow \mathsf{Choice}\ (\mathsf{Reset}\ \mathcal{G}(r)\ t_{iter})\ t_{skip}}\ \textsc{Greedy}$$

$$\frac{(l, i, gm, d) \Downarrow t_{skip} \qquad (r :: \mathsf{Acheck}\ i :: r\{0, \Delta, \bot\} :: l, i, \mathsf{GM}_{\mathsf{reset}}(gm, \mathcal{G}(r)), d) \Downarrow t_{iter}}{(r\{0, \Delta + 1, \bot\} :: l, i, gm, d) \Downarrow \mathsf{Choice}\ t_{skip}\ (\mathsf{Reset}\ \mathcal{G}(r)\ t_{iter})}\ \textsc{Lazy}$$

$$\frac{\mathsf{dir}(lk) = d' \qquad \mathsf{lk\_result}(lk, t_{look}, gm, \mathsf{idx}(i)) = \mathsf{Some}\ gm' \qquad ([r], i, gm, d') \Downarrow t_{look} \qquad (l, i, gm', d) \Downarrow t}{((?lk\ r) :: l, i, gm, d) \Downarrow \mathsf{LK}\ lk\ t_{look}\ t}\ \textsc{Lookaround}$$

$$\frac{\mathsf{dir}(lk) = d' \qquad \mathsf{lk\_result}(lk, t_{look}, gm, \mathsf{idx}(i)) = \mathsf{None} \qquad ([r], i, gm, d') \Downarrow t_{look}}{((?lk\ r) :: l, i, gm, d) \Downarrow \mathsf{LKMismatch}\ lk\ t_{look}}\ \textsc{LookaroundFail}$$

Fig. 4. Inductive tree semantics

to the input i; this is used in quantifier semantics to check that each optional iteration has not matched the empty string.

The values of all capture groups are stored in a group map, gm, associating to each capture group index g the optional range that the group last matched. In the initial group map $\mathsf{GM}_\emptyset$, none of the groups are defined. The function $\mathsf{GM}_{\mathsf{open}}(gm, g, n)$ updates the group map gm to record that capture group g is open at position n. Then, $\mathsf{GM}_{\mathsf{close}}(gm, g, n)$ and $\mathsf{GM}_{\mathsf{reset}}(gm, gl)$ respectively close the group g and reset the value of each group in the list gl.

Explanations of rules. We discuss here the most interesting rules of the semantics. To pass a progress check (Check), the semantics ensures that some progress has been made in the current input i compared to the input i_{check} at which the progress check was generated. For character descriptors (Read), the semantics checks with the function $\mathsf{advance}(cd, i, d)$ that the next character c (for direction d) in the input i can be matched by the character descriptor cd, and returns the next input zipper i'. In contrast with traditional regular expressions, matching the sequence $r_1 r_2$ does not mean that we can split the input into disjoint parts (one that matches r_1 and the other that matches r_2): a lookahead in r_1 might explore the same part of the input that is matched by r_2. In SeqForward, we instead match r_1 at the current input, and add r_2 to the list of actions to match it afterwards. Handling the sequence also depends on the current direction of matching (SeqBackward). For anchors (Anchor), $\mathsf{check\_anchor}(a, i)$ returns $\top$ when the input i satisfies the anchor a (for instance, when the input is in the final position and the anchor is \$), and $\bot$ otherwise. For backreferences (Backref), the function $\mathsf{advance\_bkrf}(gm, g, i, d)$ looks up the value of group g in gm and checks that the next characters of i match. When the group g is not defined, matching the backreference corresponds to reading the empty substring.

The rules for quantifiers come in three shapes. When a quantifier has forced repetitions ($min > 0$, Forced), the corresponding tree consists in resetting the groups inside the subregex, then matching the regex followed by the quantifier where the min has decreased. The function $\mathcal{G}(r)$ returns the list of groups defined in r. When both the min and the Δ of a quantifier are 0, the quantifier is forced to be skipped (Done). Finally, when $min = 0$ and $\Delta > 0$, there is a Choice in the tree, between doing one more iteration or skipping the quantifier. For a greedy quantifier (Greedy), the extra iteration has higher priority, while for a lazy quantifier (Lazy), skipping has higher priority.

For lookarounds (Lookaround) the semantics considers a tree t_{look} of the regex inside the lookaround from the current input position and the direction of lk ($\rightarrow$ for lookaheads, $\leftarrow$ for lookbehinds). Then, the function $\mathsf{lk\_result}(lk, t_{look}, gm, i)$ checks that this tree t_{look} contains an accepting branch if the lookaround is positive, in which case it returns an updated group map with the groups defined in the first accepting branch of that tree. If the lookaround is negative, the function checks that the tree has no accepting branch, and then returns the group map unchanged.

Semantics result. We call *accepting branches* the branches of a tree that end with a Match. The *leaf* of an accepting branch (a final input position and a final group map), is obtained by replaying the group operations in the branch. The result of matching a regex on a string corresponds to the leaf of the first accepting branch in the corresponding tree. When there is no such accepting branch, there is no match for the regex on the string. We define the function $\mathcal{L}_0(t, i)$ to return either the first leaf of t for input i, or None.

In Figure 2, $\mathcal{L}_0(t, \mathsf{Input}[\mathsf{a};\mathsf{b};\mathsf{b};\mathsf{c}][]) = \mathsf{Some}\,(\mathsf{Input}[][\mathsf{c};\mathsf{b};\mathsf{b};\mathsf{a}], \mathsf{GM}_{\mathsf{close}}(\mathsf{GM}_{\mathsf{open}}(\mathsf{GM}_\emptyset, 1, 1), 1, 2))$, where $\mathsf{Input}\ l_1\ l_2$ denotes an input zipper, with l_1 being the list of next characters and l_2 the list of previously seen characters.

3.2 Semantics Properties

In this section, we show that for any list of actions, input, group map and direction, there exists a unique backtracking tree. We first prove that our semantics is deterministic.

Theorem 1 (Determinism). $\forall\ l,\ i,\ gm,\ d,\ t_1,\ t_2.\ \ (l, i, gm, d) \Downarrow t_1 \wedge (l, i, gm, d) \Downarrow t_2 \implies t_1 = t_2.$

Proof. By induction over a derivation of $(l, i, gm, d) \Downarrow t_1$. □

We then prove its productivity. To do so, we define a function that computes a backtracking tree, and show that this tree respects the relation defined in Figure 4. However, the termination of this function is not straightforward: the termination of quantifiers relies on the fact that each non-forced iteration must consume at least one character, and that the input itself is finite. This is made even harder with lookarounds allowing to change the direction of matching.

We first define a function $\mathcal{T}(l, i, gm, d, n)$ with a fuel n, allowing only n recursive calls to ensure termination, but which may return None when not provided with enough fuel. Next, we define a function $||l||_d^i$ which computes an upper bound on the minimal amount of recursive calls needed for the function to terminate. The fuel of a list of actions l and a regex r, respectively $||l||_d^i$ (left) and $||r||_d^i$ (right), is defined as follows:

$$||[]||_d^i \triangleq 1 \qquad ||\varepsilon||_d^i = ||cd||_d^i \triangleq 1$$

$$||r :: l||_d^i \triangleq ||r||_d^i + ||l||_d^i \qquad ||a||_d^i = ||\backslash g||_d^i \triangleq 1$$

$$||\text{Aclose } g :: l||_d^i \triangleq 1 + ||l||_d^i \qquad ||r_1|r_2||_d^i = ||r_1\ r_2||_d^i \triangleq 1 + ||r_1||_d^i + ||r_2||_d^i$$

$$||\text{Acheck } i_{check} :: l||_d^i \triangleq 0 \qquad ||(_g\, r)||_d^i \triangleq 2 + ||r||_d^i$$

(when i_{check} is at the end for d) $\qquad ||(?lk\ r)||_d^i \triangleq 2 + ||r||_{\text{dir}(lk)}^{\text{worst}(lk,i)}$

$$||\text{Acheck } i_{check} :: l||_d^i \triangleq 1 + ||l||_d^{\text{next}(i_{check})_d} \qquad ||r\{min, \Delta, p\}||_d^i \triangleq (2 + ||r||_d^i) \times (1 + min + |i|_d)$$

(otherwise)

There are three interesting cases. For quantifiers, we know that each non-forced repetition has to make progress. As a result, there can be at most $1 + min + |i|_d$ iterations of the quantifiers, where $|i|_d$ is the number of characters left to be read in the string for direction d. For lookarounds, we cannot know ahead-of-time where they will be matched from, so we define $\text{worst}(lk, i)$ to be the worst possible input for the lookaround direction (at the beginning for a lookahead, at the very end for a lookbehind). For Acheck i_{check} actions, we know that if the input being compared to is at the end position (for instance, when the current direction is $\rightarrow$ and the input zipper is of the form Input [] l_2), then the progress check cannot succeed. And otherwise, we know that in the worst case, the input after passing the check will be $\text{next}(i_{check})_d$, after having read exactly one character.

With these definitions, we can prove that this fuel computation is indeed an upper bound on the number of recursive calls sufficient to finish the computation of the tree.

Theorem 2 (Termination). $\forall\ l,\ i,\ gm,\ d,\ n.\ \ n > ||l||_d^i \implies \mathcal{T}(l, i, gm, d, n) \neq \text{None}.$

Proof. By induction over n. In the inductive case, we prove that each recursive call strictly decreases the fuel computation. □

From here on we omit the fuel argument. Finally, we prove that the tree we compute is a correct backtracking tree for its arguments. Combined with the determinism of Theorem 1, this shows that for any list of actions, input, and direction, there exists a unique backtracking tree.

Theorem 3 (Functional Semantics Correctness). $\forall\ l,\ i,\ gm,\ d.\ (l, i, gm, d) \Downarrow \mathcal{T}(l, i, gm, d).$

Proof. By induction over the number of recursive calls. □

4 Semantics Faithfulness: Equivalence to Warblre

In this section, we prove that our tree semantics is faithful to the 14th edition of the ECMAScript specification [ECMA 2023]. To do so, we provide a Rocq proof that it is equivalent to Warblre [De Santo et al. 2024], a faithful, manually audited, line-by-line translation of the ECMAScript regex chapter into Rocq. The ECMAScript regex chapter consists of more than 30 pages of a pseudocode algorithm. The main function of that algorithm, *CompilePattern* (that we name compile(r) for short), compiles a regex into a pseudocode *matcher function* that implements a backtracking search. This matcher function takes as input a string and a starting index position, and returns a *match result*, which either indicates that no match was found at that starting position, or otherwise returns the final string position and the set of capture group values. We prove that the first accepting branch of our backtracking tree semantics corresponds to the result of the Warblre matcher function.

Regex equivalence. First, to facilitate proofs and provide concise rules, the regex type we use in our semantics differs slightly from the AST used in ECMAScript and Warblre. While our quantifiers use convenient *min* and Δ parameters, in Warblre each quantifier has a *min* and a *max* parameter, which corresponds to $min + \Delta$. At first glance, our type appears less expressive, but ECMAScript starts compilation by checking that the regex is well-formed, which includes that in all quantifiers, $max \geq min$. We denote by $\mathcal{W}$ the set of well-formed Warblre regexes (those that pass *Early Errors* in ECMAScript parlance). We define a function, $\downarrow r_w \downarrow$ translating a well-formed regex $r_w \in \mathcal{W}$ from the Warblre AST into ours. Another interesting difference is the indexing of capture groups. In Warblre, capture groups have no indices in the AST. Instead, at each manipulation of a group, its index is recomputed by counting the number of open parentheses that occurred before in the original regex. This process requires carrying around a cumbersome compilation context in Warblre definitions. To facilitate definitions and proofs instead, our $\downarrow r_w \downarrow$ function annotates in the AST the index of each group. Similarly, it translates every named group or named backreference into the corresponding indexed group and indexed backreference. Finally, there are other minor differences in the way characters are represented. For instance, to facilitate formal proofs, we use the same type to represent both single characters and Unicode or identity escapes.

Finally, we can prove the equivalence below (Theorem 4). The Warblre definitions use strings (list of characters) and natural numbers to encode position. Our zipper inputs are more convenient to express the tree semantics, and we note str(i) the original string of the zipper input i. Finally, as our group map type slightly differs from the Warblre definition, we define a function $\uparrow res \uparrow$ that transforms our result into a Warblre one.

Theorem 4 (Faithfulness). $\forall\ i,\ r_w \in \mathcal{W}$,
$\mathsf{compile}(r_w)(\mathsf{str}(i), \mathsf{idx}(i)) = \uparrow \mathcal{L}_0(\mathcal{T}([\downarrow r_w \downarrow], i, \mathsf{GM}_\emptyset, \rightarrow), i) \uparrow$

Proof. We summarize a few key insights of this considerable proof (over 6000 lines of Rocq). In order to define the function compile(r), ECMAScript generates *matcher continuations*, functions that can match a regex in a particular direction, open and close a capture group, or check for progress in the string. Our proof starts by defining an equivalence relation, parameterized by a direction, between these continuations and our lists of actions. We prove an intermediate theorem: the matcher continuation of a regex r_w and a direction d is equivalent, for d, to the list of actions $[\downarrow r_w \downarrow]$. Theorem 4 then follows as a corollary.

The proof of that intermediate theorem proceeds by induction. For quantifiers, we then proceed by induction on their maximum number of iterations (the sum of *min* and the size of the remaining string for the current direction). Groups are also handled differently in the two semantics. In our semantics, we generate an Open node before matching the subregex, but in ECMAScript, the corresponding matcher continuation remembers the initial input position, matches the regex, then

opens and closes the group at the same time. As a result, we prove as an invariant that whenever our group maps have partially defined groups (opened but not yet closed), the corresponding matcher continuations in Warblre will later open the same groups at the same index. □

5 Formally Verified Regex Equivalence

To demonstrate the practicality of our semantics, we use it to define a new notion of contextual regex equivalence for backtracking semantics. We use it to prove and disprove some seemingly intuitive equivalences, and even highlight bugs in a real-world JavaScript regex processor.

Reasoning about modern regex equivalence has many real-world uses. Derivative-based engines compute regex equivalence classes among derivatives during matching [Moseley et al. 2023; Owens et al. 2009; Varatalu et al. 2025]. Regex-optimization libraries (like `regexp-tree` [Soshnikov 2025] for JavaScript regexes) rewrite regexes into equivalent, *optimized* regexes that are expected to be faster to match. The topic of traditional regular expression equivalence is well studied, even in proof assistants [Asperti 2012; Coquand and Siles 2011; Krauss and Nipkow 2012; Moreira et al. 2012; Nipkow and Traytel 2014], but these definitions are not applicable to modern regexes with backtracking semantics and nontraditional features. For these, the literature is much more sparse. Freydenberger [2013] has shown that backreferences make regex equivalence undecidable. Recent work has also established a few results for regex languages with restricted feature sets. For instance, Zhuchko et al. [2024] show that some anchors can be expressed in terms of lookarounds and Mamouras and Chattopadhyay [2024] establish regex equivalence formulae about lookarounds.

However, even these new equivalences may not hold for a full real-world language with backtracking semantics. For instance, we found that some lookaround equivalences from Mamouras and Chattopadhyay [2024], while correct in a language without backreferences, do not hold in JavaScript. This is because lookarounds do not commute: the regex $(?=(_1\,\mathrm{a}))(?=\mathrm{a}\backslash 1)\mathrm{a}$ does not match the string `"a"`, as the first lookaround sets group 1 to `"a"`, but then the second one fails to read `"a"` twice. In contrast, $(?=\mathrm{a}\backslash 1)(?=(_1\,\mathrm{a}))\mathrm{a}$ does match the string `"a"`: since group 1 is not defined yet, the backreference matches the empty string.

We present a new way to formally verify regex equivalence using our backtracking tree semantics. To facilitate comparison to previous work, we restrict this discussion to the default JavaScript semantics, leaving out the `i` (`ignoreCase`), `m` (`multiline`), and `s` (`dotAll`) flags.

5.1 A New Definition of Regex Equivalence

Observational equivalence and its limitations. The most natural way to define regex equivalence is to say that r_1 and r_2 are equivalent when for any input i, matching r_1 on i returns the same result as matching r_2 on i. We refer to this as *observational equivalence*, and denote it as $r_1 \approx r_2$. While observational equivalence is enough to replace the matching of a regex with another, it is not strong enough to be used locally, to replace a subregex within a bigger context.

For instance, consider the regexes $\varepsilon|\mathrm{a}$ and $\varepsilon|\mathrm{b}$. These two regexes are observationally equivalent: on any input, they will both match the empty substring since this corresponds to the match with the highest priority. However, we cannot replace $\varepsilon|\mathrm{a}$ with $\varepsilon|\mathrm{b}$ in a bigger context and preserve equivalence. For instance, $\mathrm{c}\langle\varepsilon|\mathrm{a}\rangle\mathrm{c}$ matches the input `"cac"`, but $\mathrm{c}\langle\varepsilon|\mathrm{b}\rangle\mathrm{c}$ does not.

Directional contextual equivalence. Instead, we need a notion of equivalence that allows local rewriting of regexes. We refer to this as *contextual equivalence*, and use the notation $r_1 \sim r_2$. We would like this definition to imply observational equivalence in a bigger context: $r_1 \sim r_2 \implies \forall\, C.\ C[r_1] \approx C[r_2]$, where C is a regex context (a regex with a hole).

But in fact, we show that we can define a more precise equivalence with the following novel observation: some regex equivalences only hold in a given matching direction. As a result, in the

rest of this section we define two notions of equivalence, $r_1 \sim_{\rightarrow} r_2$ and $r_1 \sim_{\leftarrow} r_2$, for equivalences that can be rewritten inside different contexts.

We distinguish three types of contexts. *Forward contexts*, denoted as $\overrightarrow{C}$, have their hole inside a lookahead, meaning that we know the regex will be matched in the forward direction. *Backwards contexts*, denoted as $\overleftarrow{C}$, have their hole inside a lookbehind. In cases where lookarounds are nested, the deepest lookaround sets the direction of the context. Finally, *bidirectional* contexts, denoted as $C^{=}$, are contexts in which the hole does not appear in a lookaround.

Using leaves to define contextual equivalence. We now show that *leaves* of backtracking trees of regexes are sufficient to define this directional contextual equivalence. Crucially, the top priority match of $c\langle\varepsilon|a\rangle c$ on "cac" (our previous example) uses the *second* priority match of subexpression $\varepsilon|a$. To define a contextual equivalence, we must reason on several possible matches of a regex, which our tree semantics was designed to formalize. We first define $\mathcal{L}(t, i, d)$ to return the order-preserving list of leaves (pairs of a final input and a final group map at a Match node) of the tree t for input i and direction d. We say that two lists of leaves are equivalent, noted $leaves_1 \equiv leaves_2$, when they are equal after removing lower-priority duplicates in each list. For instance, $[(i_1, gm_1); (i_2, gm_2); (i_1, gm_1)] \equiv [(i_1, gm_1); (i_2, gm_2)]$. Finally, we define directional contextual equivalence as follows:

$r_1 \sim_d r_2 \triangleq \mathcal{G}(r_1) = \mathcal{G}(r_2) \wedge \forall\, i,\ gm.\ \mathcal{L}(\mathcal{T}([r_1], i, gm, d), i, d) \equiv \mathcal{L}(\mathcal{T}([r_2], i, gm, d), i, d)$

We require the two regexes to define exactly the same groups, so that replacing r_1 with r_2 will not turn a well-formed regex into an ill-formed one with duplicated groups. We use notation $r_1 \sim_{\leftrightarrow} r_2$ for equivalence in both directions. Removing duplicates in the list of leaves allows to prove equivalences like $c|c \sim_{\leftrightarrow} c$: even though the first regex has more leaves, any result that can be obtained from the second branch could also be obtained from the first one.

Using these definitions, we were able to prove the following theorems:

Theorem 5 (Bidirectional Equivalence). $\forall\, r_1,\ r_2,\ d.\ \ r_1 \sim_d r_2 \implies \forall C^{=}.\ C^{=}[r_1] \sim_d C^{=}[r_2]$

Theorem 6 (Forward Equivalence). $\forall\, r_1,\ r_2.\ \ r_1 \sim_{\rightarrow} r_2 \implies \forall\, \overrightarrow{C},\ d.\ \overrightarrow{C}[r_1] \sim_d \overrightarrow{C}[r_2]$

Theorem 7 (Backward Equivalence). $\forall\, r_1,\ r_2.\ \ r_1 \sim_{\leftarrow} r_2 \implies \forall\, \overleftarrow{C},\ d.\ \overleftarrow{C}[r_1] \sim_d \overleftarrow{C}[r_2]$

Theorem 8 (Contextual to Observational Equivalence). $\forall\, r_1,\ r_2.\ \ r_1 \sim_{\rightarrow} r_2 \implies r_1 \approx r_2$

Proof. Most proofs proceed by induction over the context. In the case of quantifiers for the bidirectional equivalence, we proceed by induction over the length of the input remaining to match (giving an upper-bound for the number of remaining iterations). □

5.2 Formally Verified and Invalid Equivalences

Using these theorems, we can now locally rewrite subregexes, using the forward equivalence when inside a lookahead or outside of lookarounds, and the backward equivalence when inside a lookbehind. As a case study, we have proved or disproved rewrites from the literature. We consider three sets of regex rewrites: rewriting anchors as lookarounds, quantifier merging, and the traditional associativity and distributivity of sequence and disjunction.

Associativity and distributivity. The regex equivalences we have proved correct are shown on Figure 5. The missing directions for distributivity are not shown because they are not correct: we provide counter-examples in Figure 6 and Appendix B of the extended version of this paper [Barrière et al. 2025b]. We prove associativity for both the sequence and the disjunction. These proofs illustrate that our semantics eliminates some unnecessary complexity from the Warblre definitions. Consider for instance the associativity of disjunction. In Warblre, we would need to prove that even though

Associativity	Anchors as lookarounds
$r_1\|\langle r_2\|r_3\rangle \sim_{\leftrightarrow} \langle r_1\|r_2\rangle\|r_3$	$\hat{} \sim_{\leftrightarrow} (?<!\odot)$
$r_1\langle r_2 r_3\rangle \sim_{\leftrightarrow} \langle r_1 r_2\rangle r_3$	$\$ \sim_{\leftrightarrow} (?!\odot)$
Distributivity (when r_1 has no group)	$\backslash b \sim_{\leftrightarrow} (?<!\backslash w)(?=\backslash w)\|(?<=\backslash w)(?!\backslash w)$
$r_1\langle r_2\|r_3\rangle \sim_{\leftarrow} \langle r_1 r_2\rangle\|\langle r_1 r_3\rangle$	$\backslash B \sim_{\leftrightarrow} \langle(?<=\backslash w)\|(?!\backslash w)\rangle\langle(?<!\backslash w)\|(?=\backslash w)\rangle$
$\langle r_2\|r_3\rangle r_1 \sim_{\rightarrow} \langle r_2 r_1\rangle\|\langle r_3 r_1\rangle$	

Fig. 5. Correct regex equivalences — associativity, distributivity and anchors

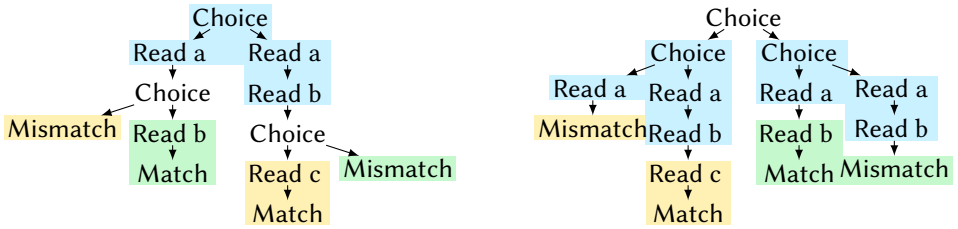


Fig. 6. Backtracking trees of $\langle$a|ab$\rangle\langle$c|b$\rangle$ and $\langle$a|ab$\rangle$c|$\langle$a|ab$\rangle$b on string "abc"

r_2 is compiled in different contexts, these two contexts are equivalent in the sense that they do not change the ordering of capture groups, requiring a new general property for matcher functions compiled in equivalent contexts. In contrast, with our equivalence definition, this associativity boils down to the associativity of the concatenation of lists of leaves, and the proof concludes in five lines. We also find that contrary to traditional regular expressions, concatenation and disjunction can only distribute under certain conditions. First, the distributed regex must not define any groups, to preserve the property that each capture group is defined at most once in JavaScript regexes. Second, depending on the context direction, distributing can change the top priority result. For instance, in Figure 6, we can see that distributing a|ab over c|b changes which disjunction is considered first in their respective backtracking trees, as a result the two middle branches are inverted and the regexes are not equivalent.

Anchors as lookarounds. We also proved that anchors can be rewritten as lookarounds, where $\odot$ is a regex that matches all characters and \w matches all word characters. These equivalences had been formulated by Varatalu et al. [2023] and the first two had been mechanized in Lean by Zhuchko et al. [2024], but for a regex language without backtracking semantics. Now, using our semantics and equivalence definitions, we can prove that these rewrites hold for JavaScript regexes.

Quantifier merging. Finally, the JavaScript regex processor library `regexp-tree` [Soshnikov 2025] optimizes repeated quantifiers in a regex. It replaces every instance of $r\{min_1, \Delta_1, p\}r\{min_2, \Delta_2, p\}$ with $r\{min_1 + min_2, \Delta_1 + \Delta_2, p\}$, where + is defined such that $n + \infty \stackrel{\Delta}{=} \infty + n \stackrel{\Delta}{=} \infty + \infty \stackrel{\Delta}{=} \infty$. While trying to prove the correctness of this equivalence, we have found counter-examples. But we also found that, when r defines no group, depending on the *min* and Δ parameters, their order, and the direction of the context, it can be correct to perform the optimization. First, we prove the following equivalence, allowing to change the greediness of a quantifier with only forced iterations: $r\{min, 0, \top\} \sim_{\leftrightarrow} r\{min, 0, \bot\}$. Then, we prove the eight correct equivalences shown on Figure 7. We give counter-examples for incorrect ones in Appendix B of the extended version of this paper [Barrière et al. 2025b]. The ✓ and ✗ labels indicate correct and incorrect rewrite

directions, and n/a indicates that merging is not applicable since it mixes non-forced repetitions of different priorities. Since `regexp-tree` performed all rewrites in both directions, all ✗ marks are `regexp-tree` bugs. We have reported the issue to the `regexp-tree` maintainers.[1]

1st quantifier \ 2nd quantifier	$r\{min_2, 0, p\}$	$r\{0, \Delta_2, \top\}$	$r\{0, \Delta_2, \bot\}$
$r\{min_1, 0, p\}$	$\leftrightarrow^{\checkmark}$	$\rightarrow^{\checkmark}\ \leftarrow^{\times}$	$\rightarrow^{\checkmark}\ \leftarrow^{\times}$
$r\{0, \Delta_1, \top\}$	$\leftarrow^{\checkmark}\ \rightarrow^{\times}$	$\leftrightarrow^{\checkmark}$	n/a
$r\{0, \Delta_1, \bot\}$	$\leftarrow^{\checkmark}\ \rightarrow^{\times}$	n/a	$\leftrightarrow^{\times}$

Fig. 7. Correct and incorrect regex quantifier merging equivalences

To provide an insight into these equivalence proofs, consider for instance proving one direction of the middle case of Figure 7: $r\{0, \Delta_1, \top\}r\{0, \Delta_2, \top\} \rightsquigarrow r\{0, \Delta_1 + \Delta_2, \top\}$. In the case where both Δ_1 and Δ_2 are natural numbers, this is proved by induction on Δ_1. The simplified trees of each regex are shown on Figure 8. By induction hypothesis, the two framed subtrees have equivalent leaves. However, when $\Delta_2 > 0$, the tree on the left has an extra subtree when the first quantifier is skipped, but the second one iterates. Since $\Delta_2 - 1 \leq \Delta_1 + \Delta_2$, we can prove that each leaf of this extra subtree is a leaf of the subtree of $r\{0, \Delta_1 + \Delta_2, \top\}$, and then by induction hypothesis that this is a leaf of $r\{0, \Delta_1, \top\}r\{0, \Delta_2, \top\}$ as well. As a result, all leaves of this extra subtree are duplicates of the leaves of the leftmost subtree, and the leaf equivalence between the two regexes hold.

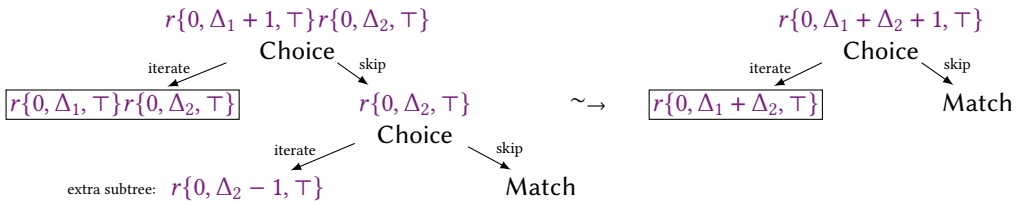


Fig. 8. Inductive case when proving $r\{0, \Delta_1, \top\}r\{0, \Delta_2, \top\} \rightsquigarrow r\{0, \Delta_1 + \Delta_2, \top\}$

These rewrites can be combined. For instance, the following chain of forward equivalences holds:

$$\begin{aligned} r\{min_0, 0, \bot\}r\{min_1, \Delta_1, \top\}r\{0, \Delta_2, \top\} &\rightsquigarrow r\{min_0, 0, \top\}r\{min_1, 0, \top\}r\{0, \Delta_1, \top\}r\{0, \Delta_2, \top\} \\ &\rightsquigarrow r\{min_0 + min_1, 0, \top\}r\{0, \Delta_1 + \Delta_2, \top\} \\ &\rightsquigarrow r\{min_0 + min_1, \Delta_1 + \Delta_2, \top\} \end{aligned}$$

For each invalid rewrite, we conjecture that restrictions either on the subregex r or on the contexts could restore correctness. For traditional regular expressions, previous work explored transformations valid within an optional repetition or a quantifier [Kahrs and Runciman 2022].

6 Formal Verification of the PikeVM Matching Algorithm

To implement modern backtracking semantics, most engines use a backtracking algorithm. For instance, both V8 and SpiderMonkey (the JavaScript implementations of Google Chrome and Firefox, respectively) use Irregexp, a backtracking regex engine. However, bactracking algorithms suffer from string-size exponential complexity, leading to the *REgex Denial Of Service* (ReDoS) vulnerability, a complexity attack harming many real-world programs [Staicu and Pradel 2018].

To sidestep this complexity vulnerability, some modern engines instead use a more restrictive flavor of regexes and implement linear-time matching algorithms. For instance, by removing

[1] https://github.com/DmitrySoshnikov/regexp-tree/issues/267

r	::=	ε	Empty
	|	cd	Character
	|	$r_1 \mid r_2$	Disjunction
	|	$r_1\ r_2$	Sequence
	|	$(_g\, r)$	Capture group
	|	$r*$	Greedy star
	|	$r*?$	Lazy star

Fig. 9. The subset of regexes, $\mathcal{P}$, supported by the PikeVM algorithm

instr	::=	`Accept`
	|	`Consume` cd
	|	`Jump` l
	|	`Fork` $l_1\ l_2$
	|	`RegOpen` g
	|	`RegClose` g
	|	`ResetRegs` gl
	|	`BeginLoop`
	|	`EndLoop` l

Fig. 10. NFA bytecode instructions

backreferences (which make the matching problem NP-hard [Dominus 2000]) and lookarounds, libraries like RE2 [Google 2010], HyperScan [Wang et al. 2019], the Rust `Regex` crate [Gallant 2014] or the Go `regexp` package [Go 2025] all achieve linear-time guarantees. For languages with non-linear features like .NET [Moseley et al. 2023] or JavaScript [V8 2021], implementers sometimes provide both a complete backtracking engine and a restricted linear engine for a subset of regexes.

The PikeVM algorithm [Cox 2009] is a popular linear-time matching algorithm for modern regexes with backtracking semantics. This algorithm is a descendant of NFA simulation [Thompson 1968], where the NFA is extended to encode priority and is represented as interpreted bytecode. With its encoding of priority, the PikeVM algorithm supports capture groups, and greedy and lazy quantifiers, and returns the same result as a backtracking algorithm but without the exponential complexity. In the examples above, all the linear engines supporting capture groups implement a PikeVM (among other linear algorithms): the RE2 library[2], the Rust `Regex` crate[3], the Go engine[4] and the linear engine of V8 for JavaScript regexes.[5]

However, the execution of the PikeVM algorithm is vastly different from the execution of a backtracking algorithm. It explores several paths in parallel, and discards some paths to avoid the exponential complexity. As a result, a PikeVM can be challenging to implement. Worse: small variations in semantics can affect the correctness of the base algorithm, and Barrière and Pit-Claudel [2024] have shown that the PikeVM implementation deployed in V8 used to contain a semantic bug. While traditional NFA simulation itself has been formalized and verified many times (see Section 7), handling priority requires new formal arguments, and to the best of our knowledge the PikeVM has never been formally verified despite its widespread use in modern engines.

In this section, we provide the first formal verification of the PikeVM algorithm. The proof has been fully mechanized in the Rocq proof assistant. Using an inductive backtracking tree semantics greatly facilitates the proof, as several crucial properties can be conveniently expressed on trees. We prove that the algorithm returns the result specified by the ECMAScript standard, making it the first time a modern regex matching algorithm is formally verified against a real-world specification.

6.1 The PikeVM Algorithm

In this section, we present a formal model of the PikeVM algorithm, first introduced by Rob Pike in the text editor `sam` [Pike 1987]. We formalize and prove a version of the PikeVM that slightly differs from the original algorithm to account for JavaScript-specific semantics: we generate instructions

[2] https://github.com/google/re2/blob/main/re2/nfa.cc
[3] https://github.com/rust-lang/regex/blob/master/regex-automata/src/nfa/thompson/pikevm.rs
[4] https://cs.opensource.google/go/go/+/master:src/regexp/exec.go
[5] https://github.com/v8/v8/tree/main/src/regexp/experimental

for capture reset, and we use the extension described in Barrière and Pit-Claudel [2024] to support JavaScript nullable quantifiers. Both of these extensions have been used in the PikeVM engine of V8 (we discuss adapting these definitions for other languages in Section 7). This algorithm supports a subset of JavaScript regexes, $\mathcal{P}$, depicted on Figure 9. This corresponds to the traditional features of regular expressions, extended with capture groups, and with the star being either greedy or lazy. We extend this subset to lists of actions (all Acheck and Aclose actions belong to the subset). We also extend this subset to trees (rejecting trees containing nodes corresponding to features that are not in $\mathcal{P}$, like BackrefPass or LK).

Regex r	**Label**	NFA(r)
ε		*no instruction*
cd		`Consume` cd
$r_1\ r_2$		NFA(r_1)
		NFA(r_2)
$r_1 \mid r_2$		`Fork` $l_1\ l_2$
	l_1 :	NFA(r_1)
		`Jump` l_3
	l_2 :	NFA(r_2)
	l_3 :	…
$(_g\, r)$		`RegOpen` g
		NFA(r)
		`RegClose` g
$r*$	l_s :	`Fork` $l_{in}\ l_{out}$
	l_{in} :	`BeginLoop`
		`ResetRegs` $\mathcal{G}(r)$
		NFA(r)
		`EndLoop` l_s
	l_{out} :	…
$r*?$	l_s :	`Fork` $l_{out}\ l_{in}$
	l_{in} :	`BeginLoop`
		`ResetRegs` $\mathcal{G}(r)$
		NFA(r)
		`EndLoop` l_s
	l_{out} :	…

Fig. 11. Compiling a regex to its bytecode extended NFA

Compilation. The first step of the PikeVM algorithm consists in computing a bytecode representing the NFA of the regex. An NFA is encoded as a list of bytecode instructions. Each bytecode instruction corresponds to a state of the NFA; these instructions are represented on Figure 10. All instructions are labeled with a natural number l indicating their position in the list. Edges of the NFA are represented in two ways: either the instructions directly contain the labels of their successor states, or there is an implicit edge between each instruction without a label and the following instruction in the list. The NFA bytecode compilation function, shown on Figure 11, is an extension of the traditional Thompson NFA construction [Thompson 1968]. The recursive compilation function, NFA(r), transforms a regex into a list of bytecode instructions. The dots "…" indicate a fresh label, to be used for the next instruction. At the end of compilation, an `Accept` instruction (the accepting state of the NFA) is appended to the end of the list of instructions. To encode priority, the labels in the `Fork` instruction are ordered: the first label corresponds to the top priority branch to explore. As a result, the compilation of $r*$ and $r*?$ differ in their first instruction, the greedy star giving more priority to doing one more iteration, and the lazy one giving more priority to exiting.

Execution. After the regex is compiled to bytecode, PikeVM interprets this bytecode on a string of characters. The algorithm reads each character of the string one at a time. In between each character, it builds a list of possible incomplete paths (ordered by priority) in the bytecode NFA.

To explore different parts of the bytecode simultaneously, PikeVM keeps track of several threads. In a thread (pc, gm, b), pc is the current label, gm is a group map, and b is a Boolean indicating whether the thread has consumed a character since entering its last quantifier. In this section, we describe the execution algorithm with small-step semantics, shown on Figure 12.[6] States of the PikeVM algorithm are either of the form $\mathsf{VM}_{\mathsf{end}}(best)$ or tuples $(i, best, \mathcal{A}, \mathcal{B}, \mathcal{S})$. In these states, i is the current input (although PikeVM explores several threads in parallel, they are all synchronized at the same position in the original string). $best$ is the top priority result found so far if any (a

[6]For a more traditional imperative description of the algorithm, we refer the reader to Cox [2009].

$$\frac{}{(i, best, [], [], \mathcal{S}) \rightarrow_{code} \mathsf{VM}_{\mathsf{end}}(best)}\ \textsc{Final} \qquad \frac{\mathsf{next}(i) = \mathsf{Some}\ i' \qquad \mathcal{B} \neq []}{(i, best, [], \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i', best, \mathcal{B}, [], \emptyset)}\ \textsc{NextChar}$$

$$\frac{(pc, b) \in \mathcal{S}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, \mathcal{A}, \mathcal{B}, \mathcal{S})}\ \textsc{Skip}$$

$$\frac{code\ \#\ pc = \mathtt{Accept} \qquad (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, \mathsf{Some}\ (i, gm), [], \mathcal{B}, \mathcal{S}')}\ \textsc{Match}$$

$$\frac{\begin{array}{c} code\ \#\ pc = \mathtt{Consume}\ cd \qquad \mathsf{advance}(cd, i, \rightarrow) \neq \mathsf{None} \\ (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\} \end{array}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, \mathcal{A}, \mathcal{B} +\!\!+ [(pc + 1, gm, \top)], \mathcal{S}')}\ \textsc{Block}$$

$$\frac{\begin{array}{c} code\ \#\ pc = \mathtt{Consume}\ cd \qquad \mathsf{advance}(cd, i, \rightarrow) = \mathsf{None} \\ (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\} \end{array}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{FailBlock}$$

$$\frac{code\ \#\ pc = \mathtt{Jump}\ pc' \qquad (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, (pc', gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{Jump}$$

$$\frac{code\ \#\ pc = \mathtt{Fork}\ pc_1\ pc_2 \qquad (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, (pc_1, gm, b) :: (pc_2, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{Fork}$$

$$\frac{\begin{array}{c} code\ \#\ pc = \mathtt{RegOpen}\ g \qquad \mathsf{GM}_{\mathsf{open}}(gm, g, \mathsf{idx}(i)) = gm' \\ (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\} \end{array}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, (pc + 1, gm', b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{Open}$$

$$\frac{\begin{array}{c} code\ \#\ pc = \mathtt{RegClose}\ g \qquad \mathsf{GM}_{\mathsf{close}}(gm, g, \mathsf{idx}(i)) = gm' \\ (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\} \end{array}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, (pc + 1, gm', b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{Close}$$

$$\frac{\begin{array}{c} code\ \#\ pc = \mathtt{ResetRegs}\ gl \qquad \mathsf{GM}_{\mathsf{reset}}(gm, gl) = gm' \\ (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\} \end{array}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, (pc + 1, gm', b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{Reset}$$

$$\frac{code\ \#\ pc = \mathtt{BeginLoop} \qquad (pc, b) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, b)\}}{(i, best, (pc, gm, b) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, (pc + 1, gm, \bot) :: \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{Begin}$$

$$\frac{code\ \#\ pc = \mathtt{EndLoop}\ pc' \qquad (pc, \top) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, \top)\}}{(i, best, (pc, gm, \top) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, (pc', gm, \top) :: \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{End}$$

$$\frac{code\ \#\ pc = \mathtt{EndLoop}\ pc' \qquad (pc, \bot) \notin \mathcal{S} \qquad \mathcal{S}' = \mathcal{S} \cup \{(pc, \bot)\}}{(i, best, (pc, gm, \bot) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \rightarrow_{code} (i, best, \mathcal{A}, \mathcal{B}, \mathcal{S}')}\ \textsc{EndStuck}$$

Fig. 12. PikeVM small-step semantics

match with even higher priority may be found later when not in a $\mathsf{VM}_{\mathsf{end}}()$ state). $\mathcal{A}$ is a list of active threads to explore, ordered by priority. $\mathcal{B}$ is an ordered list of blocked threads that reached a `Consume` instruction for the current input, which will become the list of active threads when the algorithm advances to the next input. Finally, $\mathcal{S}$ is a set of pairs (pc, b) that have already been visited in the execution for the current input. The initial state of the algorithm is defined to be $\mathsf{VM}_{\mathsf{init}}(i) \triangleq (i, \mathsf{None}, [(0, \mathsf{GM}_{\emptyset}, \top)], [], \emptyset)$.

Small-step rules. A final state is reached when there are no more active or blocked threads (Final). When a thread reaches an already explored state (where $(pc, b) \in \mathcal{S}$), it is discarded entirely (Skip). This mechanism is what makes the algorithm linear: each NFA state is explored at most once per input character (the $\mathcal{S}$ set is reset each time PikeVM moves to the next input (NextChar)). Since threads are always ordered by priority, when two threads eventually reach the same configuration (pc, b), the lower priority one is discarded. The other rules explain how to handle each instruction. When reaching an `Accept` instruction (Match), a new top-priority match is found and stored in *best*. In that case, the algorithm discards the remaining, lower-priority active threads, but keeps the blocked threads that could lead to a higher-priority match. When reaching a `Consume` instruction, the current active thread is added to the bottom of the blocked list (Block), and the Boolean is set to $\top$ to indicate that a character has been read since entering the last star. Conversely, when reaching a `BeginLoop` instruction, the Boolean is set to $\bot$. Finally, when reaching an `EndLoop` instruction, the thread is either kept or discarded depending on its Boolean (End, EndStuck).

An example of PikeVM bytecode for the regex $({}_1\mathtt{a}*|\mathtt{a})\mathtt{b}$ is shown on Figure 15a. An example execution for the input `"ab"` is shown on the right column of Figure 15c, showing the evolution of the lists $\mathcal{A}$ and $\mathcal{B}$ where each thread is represented only with its pc. The algorithm uses an unconventional exploring order of branches: it alternates between a breadth-first search (computing all possible reachable states for a given input position), and a depth-first search within a given input position. In that example, we can also see PikeVM skipping a branch: it visits pc 9 twice for the same input position (once per branch of the disjunction) and discards the second visit.

Correctness. To formally verify the correctness of PikeVM algorithm, one needs to prove that if a final state can be reached using the small-step semantics of Figure 12, then this result corresponds to the ECMAScript specification. As PikeVM is vastly different from a backtracking algorithm, the proof needs to address the following verification challenges:

- **Different progress checks:** Instead of comparing inputs, PikeVM uses a Boolean to encode progress in star iterations, as suggested in [Barrière and Pit-Claudel 2024]. One needs to verify that this Boolean correctly mirrors the outcomes of each progress check.
- **Different exploration schemes:** PikeVM explores several possible paths in parallel, while the backtracking semantics only explores one at a time in a depth-first search. One needs to prove that these two exploration schemes return the same result.
- **Skipping branches for linearity:** To ensure linearity, PikeVM skips entire branches when they correspond to previously visited states (pc, b) of the extended NFA. One needs to prove that skipping these paths does not change the result.
- **Compilation correctness:** The PikeVM compiles the regex to a bytecode. This compilation needs to be formally verified.

We have designed a proof methodology in three steps, addressing most of these challenges separately. In all steps, the backtracking tree semantics allows to write elegant invariants, and offers a convenient induction principle. In a first step, in Section 6.2, we handle the first verification challenge by presenting an alternative tree semantics, resembling the rules of Figure 4, but encoding progress with a Boolean instead of comparing the current input with the input in an Acheck action.

$$\frac{(l, i, \top) \mathrel{\overline{\Downarrow}} t}{(\text{Acheck } i_{check} :: l, i, \top) \mathrel{\overline{\Downarrow}} \text{Progress } t}\ \textsc{Check} \qquad \frac{}{(\text{Acheck } i_{check} :: l, i, \bot) \mathrel{\overline{\Downarrow}} \text{Mismatch}}\ \textsc{CheckFail}$$

$$\frac{\text{advance}(cd, i, d) = \text{Some } (c, i') \qquad (l, i', \top) \mathrel{\overline{\Downarrow}} t}{(cd :: l, i, b) \mathrel{\overline{\Downarrow}} \text{Read } c\ t}\ \textsc{Read}$$

$$\frac{(l, i, b) \mathrel{\overline{\Downarrow}} t_{skip} \qquad (r :: \text{Acheck } i :: r\{0, \Delta, \top\} :: l, i, \bot) \mathrel{\overline{\Downarrow}} t_{iter}}{(r\{0, \Delta + 1, \top\} :: l, i, b) \mathrel{\overline{\Downarrow}} \text{Choice } (\text{Reset } \mathcal{G}(r)\ t_{iter})\ t_{skip}}\ \textsc{Greedy}$$

Fig. 13. Boolean tree semantics — selected rules

$$\frac{}{([], i) \vdash b} \qquad \frac{(l, i) \vdash b}{(r :: l, i) \vdash b} \qquad \frac{(l, i) \vdash b}{(\text{Aclose } g :: l, i) \vdash b} \qquad \frac{(l, i) \vdash \top \qquad i_{check} <_{\rightarrow} i}{(\text{Acheck } i_{check} :: l, i) \vdash \top} \qquad \frac{(l, i) \vdash b}{(\text{Acheck } i :: l, i) \vdash \bot}$$

Fig. 14. Encoding actions with a Boolean

Then, to handle the two next challenges, we present an intermediate algorithm in Section 6.3, which follows the path exploration order and the skipping of PikeVM, but without compiling to bytecode. Finally, in Section 6.4, we tackle the final challenge, and show that the PikeVM algorithm working on the compiled bytecode returns the same result as the intermediate algorithm.

6.2 Correctness of Encoding Progress

First, we prove that encoding progress with a Boolean correctly allows PikeVM to predict the behavior of the progress checks. This property can be expressed purely using trees, without even considering the PikeVM compilation process.

We present a new semantics, the Boolean tree semantics, of which a few selected rules are depicted on Figure 13. This semantics is identical to the previous tree semantics of Section 3, with two exceptions. First, we add a Boolean parameter indicating whether a check would succeed. Rules Check and CheckFail use this parameter to decide whether to fail or not, independently from the input inside the Acheck action. The Boolean is updated in the premises of the rules that read a character (Read) or enter a non-forced iteration of a quantifier (Greedy). Second, since this semantics is used to verify the correctness of PikeVM, we make simplifications to reflect the restricted subset of supported regexes $\mathcal{P}$. We remove irrelevant parameters and rules, for instance the direction (no lookarounds) and the group map (no backreferences).

Linear algorithms rely on the uniform-futures property [Barrière and Pit-Claudel 2024], that states that while matching a regex, the future of a path is independent from the current values of each group. With the removal of the group map parameter, the Boolean tree semantics of Figure 13 has this property by construction.

To relate the two semantics, we define an invariant, $(l, i) \vdash b$ on Figure 14. To understand how it can be used to prove the correctness of the Boolean encoding, consider a backward proof of $(l, i) \vdash b$. After using the fourth rule of Figure 14, the hypothesis requires the Boolean to be $\top$. Consequently, the rest of the proof cannot use the fifth rule anymore. In other words, $(l, i) \vdash b$ implies that l can be split into two lists: first, a list where each Acheck i_{check} action has its input i_{check} equal to the current input i (corresponding to stars that have just been entered), followed by a list where each Acheck i_{check} action has its input i_{check} strictly smaller than i (for stars in which we have made progress already). When $b = \top$, it means that the first list contains no Acheck action: all

stars have progressed, and all progress checks will succeed. When $b = \bot$, there is at least one check for which progress has not been made, and the next check will fail unless we advance in the input.

Finally, we can prove that for the same regex and the same input, the two semantics build the same tree. With this proof, we have showed that the progress strategy used by PikeVM (encoding with a Boolean the validity of future checks), matches the specification.

Theorem 9 (Correctness of the Boolean semantics). $\forall\ l \in \mathcal{P},\ i,\ gm,\ t,\ b.$
$(l, i) \vdash b \wedge (l, i, gm, \rightarrow) \Downarrow t \implies (l, i, b)\ \overline{\Downarrow}\ t.$

Theorem 10. $\forall\ r \in \mathcal{P},\ i,\ t.\ ([r], i, \mathsf{GM}_\emptyset, \rightarrow) \Downarrow t \iff ([r], i, \top)\ \overline{\Downarrow}\ t.$

Proof. Theorem 9 is proved by induction on a derivation of $(l, i, gm, \rightarrow) \Downarrow t$. The left to right direction of Theorem 10 is a direct consequence of Theorem 9. The right to left direction is a consequence of the productivity of the tree semantics (Theorem 3), the determinism of the Boolean tree semantics (similar to Theorem 1) and Theorem 9. □

6.3 Correctness of the PikeVM Exploration Scheme

As the second step of our proof, we present an intermediate algorithm, resembling PikeVM but without bytecode compilation. This algorithm, which we name PikeTree, is given a backtracking tree, and finds its top priority accepting branch. Instead of exploring the tree in depth-first order like a backtracking algorithm, the algorithm emulates the mixed breadth-first and depth-first exploration order of PikeVM: it explores several paths in parallel, all synchronized on the same input position. Just like PikeVM avoids exploring the same NFA state twice, this algorithm also skips some subtrees that have already been explored. Our strategy consists in first showing that this algorithm always computes the first accepting branch of a given tree, then showing that a PikeVM execution of a regex and a string corresponds to a PikeTree execution of the corresponding tree (Section 6.4). The key advantage of this strategy is that we can prove two crucial properties without even considering compilation to a bytecode: the correctness of the exploration order, and the possibility of skipping entire branches. Here we also make use of a key property of our backtracking trees: since the PikeVM algorithm explores paths that are not explored by a backtracking algorithm that stops at the first priority match, it is crucial for our semantics to materialize these extra paths.

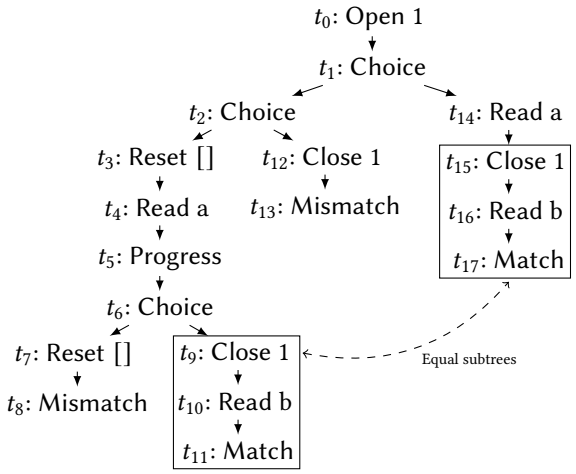
An example execution of the PikeTree algorithm is shown on Figure 15. Figure 15b shows the backtracking tree of $(_1\,\mathtt{a}*|\mathtt{a})\mathtt{b}$ on input `"ab"`, which is used to initialize the algorithm. Instead of maintaining lists of threads like PikeVM, PikeTree maintains lists of trees to explore. It can also skip trees already explored for the current input position. For instance, the subtree t_9 is equal to the subtree t_{15} (the two rectangles in Figure 15b), which allows the PikeTree algorithm to skip t_{15} in its execution on Figure 15c. On this diagram, we can see how the PikeTree algorithm is designed to match the behavior of PikeVM: each active tree corresponds to an active thread (but note that PikeTree sometimes takes fewer steps than PikeVM, see Section 6.4).

Selected rules of the PikeTree small-step semantics are presented in Figure 16. The full semantics is available in Appendix C of the extended version of this paper [Barrière et al. 2025b]. States of the PikeTree algorithm are either of the form $\mathsf{PT}_{\mathsf{end}}(best)$ when the algorithm terminates, or $(i, best, \mathcal{A}, \mathcal{B}, \mathcal{S})$ otherwise. Given a tree t and an input i, the initial state of the PikeTree algorithm is defined to be $\mathsf{PT}_{\mathsf{init}}(t, i) \triangleq (i, \mathsf{None}, [(t, \mathsf{GM}_\emptyset)], [], \emptyset)$. These states closely resemble those of the PikeVM semantics of Figure 12, with the following differences. First, $\mathcal{A}$ and $\mathcal{B}$ are now lists of pairs (t, gm) of a subtree to explore and a group map. Second, the $\mathcal{S}$ set now contains subtrees that have already been explored.

More importantly, the way in which the PikeTree algorithm skips branches is different from PikeVM. In PikeVM, different threads, at different program counters, can correspond to the execution

NFA((1 a∗|a)b)	
0	RegOpen 1
1	Fork 2 8
2	Fork 3 7
3	BeginLoop
4	ResetRegs []
5	Consume a
6	EndLoop 2
7	Jump 9
8	Consume a
9	RegClose 1
10	Consume b
11	Accept

(a) Bytecode NFA of (1 a∗|a)b



(b) Backtracking tree of (1 a∗|a)b and "ab"

PikeTree		PikeVM	
$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A}$	$\mathcal{B}$
$[t_0]$	[]	[0]	[]
$[t_1]$	[]	[1]	[]
$[t_2; t_{14}]$	[]	[2; 8]	[]
$[t_3; t_{12}; t_{14}]$	[]	[3; 7; 8]	[]
		[4; 7; 8]	[]
$[t_4; t_{12}; t_{14}]$	[]	[5; 7; 8]	[]
$[t_{12}; t_{14}]$	$[t_5]$	[7; 8]	[6]
		[9; 8]	[6]
$[t_{13}; t_{14}]$	$[t_5]$	[10; 8]	[6]
$[t_{14}]$	$[t_5]$	[8]	[6]
[]	$[t_5; t_{15}]$	[]	[6; 9]
Advance after reading "a"			
$[t_5; t_{15}]$	[]	[6; 9]	[]
$[t_6; t_{15}]$	[]	[2; 9]	[]
$[t_7; t_9; t_{15}]$	[]	[3; 7; 9]	[]
		[4; 7; 9]	[]
$[t_8; t_9; t_{15}]$	[]	[5; 7; 9]	[]
$[t_9; t_{15}]$	[]	[7; 9]	[]
		[9; 9]	[]
$[t_{10}; t_{15}]$	[]	[10; 9]	[]
$[t_{15}]$	$[t_{11}]$	[9]	[11]
Skip t_{15}=t_9		**Skip** 9	
[]	$[t_{11}]$	[]	[11]
Advance after reading "b"			
$[t_{11}]$	[]	[11]	[]
Found result		**Found result**	

(c) Execution of PikeTree and PikeVM

Fig. 15. Example execution of the PikeTree and PikeVM algorithms for regex (1 a∗|a)b and input "ab"

of the same subtree. For instance, when compiling the regex $ab \mid ab$, the PikeVM algorithm will compile ab twice in different locations. Two threads of PikeVM executing these two pieces of bytecode will always correspond to the same tree, but since they are located at different places in the bytecode, PikeVM will not skip either of them. As our goal is to relate a PikeVM execution to a PikeTree execution, we make the PikeTree algorithm non-deterministic: when it sees a tree that is already in the $\mathcal{S}$ set, it can either skip it (Skip), or not skip it and process it anyways (for instance Blocked does not require that Read $c\ t \notin \mathcal{S}$). While the PikeTree algorithm is non-deterministic, we prove that each possible execution produces the correct result, and then show that the PikeVM execution corresponds to one specific PikeTree execution.

To prove the correctness of our new algorithm we define a relation, $pts \downarrow\downarrow res$, between a state of the PikeTree semantics (pts) and an optional match result (res), then show that this relation is an invariant of the execution. This invariant is a combination of several other relations.

$$\frac{}{(i, best, [], [], \mathcal{S}) \to \mathsf{PT}_{\mathsf{end}}(best)}\ \textsc{Final} \qquad \frac{t \in \mathcal{S}}{(i, best, (t, gm) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \to (i, best, \mathcal{A}, \mathcal{B}, \mathcal{S})}\ \textsc{Skip}$$

$$\frac{\mathcal{S}' = \mathcal{S} \cup \{\mathsf{Read}\ c\ t\}}{(i, best, (\mathsf{Read}\ c\ t, gm) :: \mathcal{A}, \mathcal{B}, \mathcal{S}) \to (i, best, \mathcal{A}, \mathcal{B} +\!\!+ [(t, gm)], \mathcal{S}')}\ \textsc{Blocked}$$

Fig. 16. PikeTree small-step semantics — selected rules (full version in [Barrière et al. 2025b, Appendix C])

First, we define a non-deterministic relation $(t, gm) \downarrow^i_{\mathcal{S}} res$, relating a tree t to an optional result res. Informally, $(t, gm) \downarrow^i_{\mathcal{S}} res$ holds when res is a leftmost accepting leaf of t after removing any number of subtrees found in $\mathcal{S}$, for input i and group map gm . This means that executing the PikeTree algorithm on tree t with the set $\mathcal{S}$ can produce result res (recall that PikeTree may, but does not *have to*, skip trees in $\mathcal{S}$). For instance, if res_1 is the first accepting branch of t_1 and res_2 of t_2, then both $\mathsf{Choice}\ t_1\ t_2 \downarrow^i_{\{t_1\}} res_1$ and $\mathsf{Choice}\ t_1\ t_2 \downarrow^i_{\{t_1\}} res_2$ hold.

We extend this definition to ordered lists of trees: $\mathcal{A} \downarrow^i_{\mathcal{S}} res$ means that res is one possible first result of the list $\mathcal{A}$ after removing any number of subtrees in $\mathcal{S}$. We can further extend this definition to PikeTree semantics states (where $r_1 \circ r_2$ is equal to r_1 when it is different from None, and r_2 otherwise):

$$(i, best, \mathcal{A}, \mathcal{B}, \mathcal{S}) \downarrow res \triangleq \exists res_a,\ res_b.\ \mathcal{B} \downarrow^{\mathsf{next}(i)}_{\emptyset} res_b \wedge \mathcal{A} \downarrow^i_{\mathcal{S}} res_a \wedge res = res_b \circ res_a \circ best$$

While this relation is non-deterministic (because we can choose whether to skip subtrees in $\mathcal{S}$), it turns out that the PikeTree algorithm can only evaluate to a single result. Our final invariant, $pts \downarrow\downarrow res$, captures that fact. We define the invariant below, then prove that the invariant is correctly initialized (Theorem 11) and preserved (Theorem 12).

$$\frac{}{\mathsf{PT}_{\mathsf{end}}(best) \downarrow\downarrow best} \qquad \frac{\forall r.\ (i, best, \mathcal{A}, \mathcal{B}, \mathcal{S}) \downarrow r \implies r = res}{(i, best, \mathcal{A}, \mathcal{B}, \mathcal{S}) \downarrow\downarrow res}$$

Theorem 11 (Invariant initialization). $\forall\ i,\ t \in \mathcal{P}.\ \mathsf{PT}_{\mathsf{init}}(t, i) \downarrow\downarrow \mathcal{L}_0(t, i)$.

Theorem 12 (Preservation). $\forall\ pts_1,\ pts_2,\ res.\ pts_1 \downarrow\downarrow res \wedge pts_1 \to pts_2 \implies pts_2 \downarrow\downarrow res$.

Proof. For Theorem 11, the initial state $\mathsf{PT}_{\mathsf{init}}(t, i)$, has an empty $\mathcal{S}$ set. As a result, the only possible result is the first accepting branch of t. The proof for Theorem 12 proceeds by case analysis over $pts_1 \to pts_2$. Informally, when the algorithm skips a previously seen tree t (Skip), we know that all results of the new state are results of the previous state where t was skipped. In other cases, when visiting a new tree t, PikeTree not only adds t to the $\mathcal{S}$ set, but also adds the children of t either in $\mathcal{A}$ or in $\mathcal{B}$. Because of the addition to the $\mathcal{S}$ set, PikeTree might skip more branches (corresponding to t) in the future. However, we prove that skipping these branches will not change the result of the algorithm, since any result in these branches is also a result of the children of t. □

6.4 Correctness of the PikeVM Compiler

We now prove that each execution of the PikeVM algorithm itself, as described on Figure 12, corresponds to an execution of the PikeTree algorithm. Informally, at all times during the execution, each thread of the active or blocked list in the PikeVM state corresponds to a tree in the active or blocked list of the PikeTree algorithm (see Figure 15c). We then need to define an equivalence relation between threads and trees. Whenever PikeVM skips a thread (pc, gm, b) because it has seen its program counter pc and Boolean b already, it means that PikeTree has seen the corresponding tree already and thus is allowed to skip it as well. Most steps of the PikeVM algorithm directly correspond to a similar step in the PikeTree algorithm, with a few exceptions explained below.

$$\frac{(l, i, b) \mathrel{\overline{\Downarrow}} t \qquad rep_{code}\ l\ pc}{(t, gm) \sim^{i}_{code} (pc, gm, b)} \qquad \frac{code\ \#\ pc = \texttt{BeginLoop} \qquad (t, gm) \sim^{i}_{code} (pc + 1, gm, \bot)}{(t, gm) \sim^{i}_{code} (pc, gm, b)}$$

$$\frac{code\ \#\ pc = \texttt{ResetRegs}\ gl \qquad \mathsf{GM}_{\mathsf{reset}}(gm, gl) = gm' \qquad (t, gm') \sim^{i}_{code} (pc + 1, gm', b)}{(\text{Reset}\ gl\ t, gm) \sim^{i}_{code} (pc, gm, b)}$$

Fig. 17. Equivalence between trees and PikeVM threads

$$\frac{}{\mathsf{PT}_{\mathsf{end}}(res) \sim_{code} \mathsf{VM}_{\mathsf{end}}(res)} \qquad \frac{\mathcal{A} \sim^{i}_{code} \mathcal{A}_{VM} \qquad \mathcal{B} \sim^{\mathsf{next}(i)}_{code} \mathcal{B}_{VM} \qquad \mathcal{S}_{VM} \subseteq_{\mathcal{A}} \mathcal{S}}{(i, best, \mathcal{A}, \mathcal{B}, \mathcal{S}) \sim_{code} (i, best, \mathcal{A}_{VM}, \mathcal{B}_{VM}, \mathcal{S}_{VM})}$$

Fig. 18. Equivalence between PikeTree semantic states and PikeVM semantic states

First, we formalize this notion of equivalence between threads and trees on Figure 17. The first rule corresponds to the more common case. A thread (pc, gm, b) is equivalent to a tree t when there exists a list of actions l, such that t is the tree of l for the current input i and Boolean b, and this list of actions is *represented* in the code at label pc. This representation predicate $rep_{code}\ l\ pc$ (omitted here for brevity) is inductively defined to state that the bytecode corresponding to each action in l starts at label pc in $code$, and then points to an `Accept` instruction. The representation of an Aclose g action is a `RegClose` g instruction, and the representation of an Acheck i_{check} action is an `EndLoop` instruction. In-between the bytecode representation of each action, there may be a `Jump` instruction, in case we are representing the first branch of a disjunction (see Figure 11).

The invariant also contains two more rules to cover cases where the thread does not directly correspond to the tree of a list of actions. This happens in two cases, when the thread is at a `ResetRegs` or a `BeginLoop` instruction. Finally, we naturally extend this equivalence to lists of threads and trees, and we write $\mathcal{A} \sim^{i}_{code} \mathcal{A}_{VM}$ when all elements of the lists are pairwise equivalent.

The executions of the PikeVM and the PikeTree algorithms are quite similar. As one adds seen threads to its $\mathcal{S}$ set, the other adds equivalent trees. There is however one exception: the PikeVM algorithm sometimes performs more steps than the PikeTree algorithm. This happens when executing bytecode instructions that do not correspond to operations recorded in the backtracking tree: `Jump` and `BeginLoop`. We refer to these instructions as *stuttering* instructions. In Figure 15c, when executing labels 3 and 7, PikeVM takes one more step than PikeTree. As we execute these instructions, some threads are added to the $\mathcal{S}$ set of PikeVM before the corresponding tree is added on the PikeTree side. As a result, an invariant of the executions is that each thread (pc, gm, b) of the $\mathcal{S}_{VM}$ set of PikeVM is either equivalent to a tree in the $\mathcal{S}$ set of the PikeTree algorithm, or $code\ \#\ pc$ is a stuttering instruction and (pc, gm, b) is equivalent to the current active tree of the PikeTree algorithm (the head of the $\mathcal{A}$ list). We write $\mathcal{S}_{VM} \subseteq_{\mathcal{A}} \mathcal{S}$ to express this property. Finally, we can define on Figure 18 the invariant relating the execution of PikeVM to one execution of the PikeTree algorithm, and show that the invariant is initialized (Theorem 13) and preserved (Theorem 14).

Theorem 13 (PikeVM invariant initialization). $\forall\ r \in \mathcal{P},\ t,\ code,\ i.$
$\mathsf{NFA}(r) = code \wedge ([r], i, \top) \mathrel{\overline{\Downarrow}} t \implies \mathsf{PT}_{\mathsf{init}}(t, i) \sim_{code} \mathsf{VM}_{\mathsf{init}}(i).$

Theorem 14 (PikeVM invariant preservation). $\forall\ pts_1,\ pvs_1,\ pvs_2,\ code.$
$pts_1 \sim_{code} pvs_1 \wedge pvs_1 \rightarrow_{code} pvs_2 \implies (\exists pts_2.\ pts_1 \rightarrow pts_2 \wedge pts_2 \sim_{code} pvs_2) \vee (pts_1 \sim_{code} pvs_2).$

Proof. For Theorem 13, we show that the list of actions $[r]$ is represented at the label 0 of its compiled code $code$ by induction over r. Then, the proof of Theorem 14 proceeds by case analysis

over $pvs_1 \rightarrow_{code} pvs_2$. Most cases are proved by induction over the $(l, i, b) \mathrel{\overline{\Downarrow}} t$ predicate relating the current active tree t being executed by PikeTree and the actions l represented at the current active thread executed by PikeVM. An induction is required because this list of actions could start with an arbitrary sequence of ε actions before the action responsible for the current operation in the tree and current instruction in the bytecode. By construction of the Boolean semantics (Figure 13), we deduce that progress checks on the PikeTree side match the checks on the PikeVM side. When PikeVM decides to skip a thread, it means this thread was in its $\mathcal{S}$ set. From the inclusion between $\mathcal{S}$ sets, we know that there exists an equivalent tree that can be skipped on the PikeTree side. □

Finally, we can prove that PikeVM returns the same result as the ECMAScript specification (Theorem 16) by first proving that PikeVM returns a result of PikeTree (Theorem 15), where $\rightarrow^*$ represents the transitive reflexive closure of a small-step relation.

Theorem 15 (PikeVM to PikeTree execution). $\forall\, r \in \mathcal{P},\ i,\ t,\ res.$
$([r], i, \top) \mathrel{\overline{\Downarrow}} t \wedge\ \mathsf{VM}_{\mathsf{init}}(i) \rightarrow^*_{\mathsf{NFA}(r)} \mathsf{VM}_{\mathsf{end}}(res) \implies \mathsf{PT}_{\mathsf{init}}(t, i) \rightarrow^* \mathsf{PT}_{\mathsf{end}}(res)$

Theorem 16 (PikeVM correctness theorem). $\forall\, r_w \in \mathcal{W},\ i,\ res.$
$\lfloor r_w \rfloor \in \mathcal{P} \wedge \mathsf{VM}_{\mathsf{init}}(i) \rightarrow^*_{\mathsf{NFA}(\lfloor r_w \rfloor)} \mathsf{VM}_{\mathsf{end}}(res) \implies \mathsf{compile}(r_w)(\mathsf{str}(i), \mathsf{idx}(i)) = \lceil res \rceil.$

Proof. Theorem 15 is proved by induction over the derivation of $\mathsf{VM}_{\mathsf{init}}(i) \rightarrow^*_{\mathsf{NFA}(r)} \mathsf{VM}_{\mathsf{end}}(res)$, applying Theorem 14 at each step. Our final Theorem 16 then follows from previous results. From Theorem 15, we know that each execution of the PikeVM algorithm on a regex and an input corresponds to an execution of the PikeTree algorithm on the corresponding Boolean tree. From Theorem 9, we know that this corresponding Boolean tree is the backtracking tree of that regex and that input. From Theorems 12 and 11, we know that each result of the PikeTree algorithm corresponds to the first accepting branch of that tree. Finally, from Theorem 4, we know that this first accepting branch is precisely the result defined by the ECMAScript standard. □

Despite the PikeVM algorithm having been used widely for decades, this is the first time that it has been formally verified. Reasoning on backtracking trees greatly facilitates the proof as it enables simple ways to express crucial properties. For instance, proving the correctness of skipping branches would be much harder to prove directly on the PikeVM algorithm: two executions of the same bytecode instruction might yield completely different results when the $\mathcal{S}$ set is different.

7 Discussion and Related Work

Adapting our work to other languages. While our semantics matches that of the JavaScript regexes, it could be straightforwardly modified for other languages with backtracking semantics. We chose JavaScript for its wide use and because it is the only modern backtracking semantics regex language to come with a full mechanized specification that we can formally compare ourselves to.

For the PikeVM, we could expect the following changes. In languages without capture reset, we could simply remove the Reset node from the tree type, and the PikeVM would not generate any `ResetRegs` instruction. In languages without the special nullable quantifier semantics of JavaScript, we would remove the Acheck action, and make all nodes of the tree check for progress. The PikeVM would become simpler: we could remove the Boolean from threads entirely, and remove instructions `BeginLoop` and `EndLoop`. In the proof, we would remove the Boolean semantics of Section 6.2 entirely. In both cases, there would be one less case in Figure 17.

For contextual equivalence, most of our proofs of Section 5 do not depend on JavaScript specificities (with the exception of anchor rewriting equivalences). We expect them to hold in any

backtracking semantics language. Counter-examples are easier to generalize as they can be confirmed with tests, we discuss them in Appendix B of the extended version of this paper [Barrière et al. 2025b].

7.1 Related Work

With the exception of Warblre, mechanized semantics and verification work typically exclude backtracking semantics and capture groups. This is known to be difficult: according to Zhuchko et al. [2024], "*Formalizing even the core aspect of the capture semantics and algorithms [...] is therefore a major undertaking with many challenges*". Our semantics and proofs provide a solution.

Warblre. Warblre [De Santo et al. 2024] was the first mechanization, in an interactive theorem prover, of a modern, general-purpose regex language (ECMAScript 2023 [ECMA 2023, §22.2]). By closely following the paper specification, it traded succintness and ease-of-use for high auditability and faithfulness; accordingly, its authors described it as "*a foundation for researchers to restate the semantics in a way that better suits their field*". Our Section 3 does that, and provides a new, complete formalization of ECMAScript 2023 §22.2. It preserves the desirable properties of Warblre (in particular, Theorem 4 shows faithfulness), but without its shortcomings: our tree semantics are more succinct, and much easier to reason about (we provide an induction principle, and we eliminate the burden of reasoning about the error monad).

JavaScript regex formalizations. Several previous efforts have developed unmechanized formal semantics for JavaScript regexes. Loring et al. [2019] presented a formalized semantics for ECMAScript 2015 regexes and used it for symbolic execution of programs with regexes. Similarly, Eriksson et al. [2023] augmented the Ostrich constraint solver with a model of ECMAScript regexes, using two-way alternating automata. Recently, Chida and Terauchi [2023, 2024] presented a formal semantics for JavaScript regexes used for regex repair. Their semantic statement includes a matching direction and a group map just like ours, and their *continuation regex* resembles our list of actions, but they only formalize the top-priority match. Finally, Chen et al. [2022] formalized a large subset of JavaScript regex semantics using prioritized streaming string transducers. While there are mechanizations of the JavaScript language [Bodin et al. 2014; Park et al. 2020], none include the regular expression chapter of ECMAScript. In fact, besides Warblre, our semantics is the only mechanized one for JavaScript regexes, and also the only complete one: other formalizations exclude some JavaScript features such as lookbehinds, regex flags, or backreferences. Manually defining formal semantics for a modern regex language is a complex and error-prone task. For instance, careful inspection of the semantics proposed by Eriksson et al. [2023, Extended version, Table 5] reveals some mistakes: capture reset is not performed, and the semantics of lookarounds is incorrect. JavaScript lookarounds are supposed to be atomic, meaning that once a match is found for a lookaround, an engine cannot backtrack and match it in a different way [ECMA 2025, Section 22.2.2.4, Note 3]. Such subtle differences are easy to miss (this behavior of lookarounds can only be observed with a backreference to a capture group defined inside a positive lookaround); by contrast, our proof of Theorem 4 ensures the correctness of the semantics. Still, mechanization also entails specific challenges: for instance, some paper formalizations use *non-strict positive occurrences* by referring to the negation of matching within the definition of the matching relation itself (for instance for negative lookarounds, or for the second branch of the disjunction). Such definitions would have to be adjusted to be accepted by Rocq [Zhuchko et al. 2024].

For a different language, the original PCRE documentation [Hazel 2012] notes that "*The set of strings that are matched by a regular expression can be represented as a tree structure [...] of infinite size*" and mentions both depth-first and breadth-first traversal of the tree. We have found that specializing the tree to a particular input provides a finite tree that is more practical for formal

verification than a coinductive one, and that the PikeVM algorithm is more than a breadth-first traversal: it alternates between depth-first and breadth-first order to maintain priority.

Formalizations of modern features. Other work has also verified or formalized individual modern features, but in simplified languages. For instance, matching algorithms supporting lookarounds have recently been mechanized in Lean [Zhuchko et al. 2024] and Rocq [Chattopadhyay et al. 2025]. However, these works include neither capture groups nor backtracking semantics. Other unmechanized work specify capture groups using prioritized transducers [Berglund and van der Merwe 2017] or encode lookarounds as alternating automata [Berglund et al. 2021].

Traditional regular expressions. Traditional regular expressions and finite state automata, including NFA simulation, have been verified and mechanized many times [Braibant and Pous 2010; Cardoso et al. 2023; Coquand and Siles 2011; Doczkal et al. 2013; Doczkal and Smolka 2018; Firsov and Uustalu 2013; Jourdan et al. 2012; Krauss and Nipkow 2010; Owens and Slind 2008]. Regular expression matching algorithms have been formally verified for lexers [Chassot and Kunčak 2024; Egolf et al. 2021, 2022; Ouedraogo et al. 2024] and in particular for *leftmost-longest* semantics [Ausaf et al. 2016; Urban 2023], but not for backtracking semantics.

7.2 Future Work

Our semantics is verified to be equivalent to the latest version of Warblre, based on the 2023 edition of ECMAScript [ECMA 2023]. As of 2025, two new editions have been released [ECMA 2024, 2025], introducing new regex features: the `v` Unicode flag, modifiers, and duplicate named groups. If Warblre were to be updated, we could adapt accordingly. First, the new `v` flag adds new features in character descriptors that we could support by adapting the advance(cd, i, d) function. Second, new modifiers allow flags `i`, `m` and `s` to be set locally in a subregex. Currently, we pass the record containing the value of each flag as parameter of the $(l, i, gm, d) \Downarrow t$ predicate. To support local modifiers, we would pass this record as an index instead, and add a corresponding type of node in backtracking trees. Finally, duplicate named groups allow some capture group identifiers to appear several times in distinct disjunction branches. If Warblre's $\mathcal{W}$ set were updated, we expect that we could accordingly relax our definition of contextual equivalence to allow equivalent regexes to have duplicated named groups.

While we have proved the *correctness* of the PikeVM algorithm, we have not verified an implementation, let alone an efficient one. A realistic engine would require efficient data structures (e.g., for capture groups and group maps), matching optimizations (e.g., prefix acceleration), and a specification and implementation of top-level APIs (e.g., `matchAll`). Other bugs in implementations can lie in their complexity. Future work could prove that the PikeVM semantics terminates in a number of steps linear in the size of the input.

It would also be interesting to extend the base PikeVM algorithm with additional features, to obtain a formally verified linear-time algorithm supporting all JavaScript regex features except backreferences. We expect that adapting our proofs would be straightforward for most missing features. To support anchors, existing PikeVM implementations compile them to a new bytecode instruction that checks the surroundings of the current string position. Supporting the question mark quantifiers can be done with only `Fork`, `BeginLoop` and `EndLoop` instructions. Then, generic quantifiers $r\{min, \Delta, \mathrm{p}\}$ can be handled by duplicating the bytecode of r (for instance, compiling $r\{1, 1, \top\}$ concatenates the bytecodes of r and r?). A more challenging extension would consist in supporting lookaorunds, following the algorithm of Barrière and Pit-Claudel [2024].

Finally, other work has also explored other extended features that are not part of JavaScript, such as *atomic groups* [Fujinami and Hasuo 2024], the *shuffle* operator [Thiemann 2016], or verifying in Idris 2 [Kammar and Marek 2023] the type-safety of an engine supporting *typed regexes* [Radanne

2019]. Other work goes beyond capture groups and instead returns *parse trees* for standard regular expressions, indicating the full list of substrings a subexpression has matched [Dubé and Feeley 2000; Frisch and Cardelli 2004; Nielsen and Henglein 2011; Ribeiro and Bois 2017]. An interesting future direction would be to extend our work to support these extensions and alternative semantics.

8 Conclusion

We have presented a new semantics for JavaScript regexes, that is mechanized in Rocq, complete yet succinct, proven to be faithful to the ECMAScript standard, and practical for formal verification. We have showed that formalizing not only the first match of a regex, but also lower-priority matches, makes the semantics practical. We have used it to provide novel proofs of real-world applications: the first verification of the PikeVM linear matching algorithm, used in many deployed engines; and a new formal notion of contextual equivalence that allowed us to prove and disprove regex equivalences from the literature and from the optimizer of a popular JavaScript regex manipulation library. Thanks to the verified connection with an audited formalization, formal proofs conducted with our semantics can be trusted to correspond exactly to the behavior specified by ECMAScript. Our work lays the foundation for the development of verified and realistic modern regex engines.

Acknowledgments

We thank Yann Herklotz, Martin Odersky, and Marcin Wojnarowski for feedback on this paper, and Eugène Flesselle for preliminary exploration of regex equivalence. This research was funded in whole or in part by the Swiss National Science Foundation (SNSF), grant number 10003649.

Artifact Availability

Our development is free software and can be accessed online: https://github.com/epfl-systemf/Linden. We have also packaged all the definitions and proofs presented in this paper in an artifact [Barrière et al. 2025a]. The artifact offers both the standalone Rocq files, and a virtual machine image (along with a script to create this image from scratch).

References

Andrea Asperti. 2012. A Compact Proof of Decidability for Regular Expression Equivalence. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7406)*, Lennart Beringer and Amy P. Felty (Eds.). Springer, 283–298. doi:10.1007/978-3-642-32347-8_19

Fahad Ausaf, Roy Dyckhoff, and Christian Urban. 2016. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9807)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer, 69–86. doi:10.1007/978-3-319-43144-4_5

Aurèle Barrière and Clément Pit-Claudel. 2024. Linear Matching of JavaScript Regular Expressions. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1336–1360. doi:10.1145/3656431

Aurèle Barrière, Victor Deng, and Clément Pit-Claudel. 2025a. *Artifact for "Formal Verification for JavaScript Regular Expressions: a Proven Mechanized Semantics and its Applications", at POPL 2026.* doi:10.5281/zenodo.17305393

Aurèle Barrière, Victor Deng, and Clément Pit-Claudel. 2025b. Formal Verification for JavaScript Regular Expressions: a Proven Semantics and its Applications (Extended Version). arXiv:2507.13091 [cs.PL] https://arxiv.org/abs/2507.13091

Martin Berglund and Brink van der Merwe. 2017. On the semantics of regular expression parsing in the wild. *Theor. Comput. Sci.* 679 (2017), 69–82. doi:10.1016/J.TCS.2016.09.006

Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. 2021. Regular Expressions with Lookahead. *J. Univers. Comput. Sci.* 27, 4 (2021), 324–340. doi:10.3897/JUCS.66330

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 87–100. doi:10.1145/2535838.2535876

Thomas Braibant and Damien Pous. 2010. An Efficient Coq Tactic for Deciding Kleene Algebras. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 163–178. doi:10.1007/978-3-642-14052-5_13

Elton Maximo Cardoso, Leonardo Vieira dos Santos Reis, and Rodrigo Geraldo Ribeiro. 2023. A Verified Operational Semantics for Regular Expression Parsing. In *Proceedings of the XXVII Brazilian Symposium on Programming Languages, SBLP 2023, Campo Grande, MS, Brazil, September 25-29, 2023*. ACM, 82–90. doi:10.1145/3624309.3624317

Samuel Chassot and Viktor Kunčak. 2024. Verified invertible lexer using regular expressions and DFAs. arXiv:2412.13581 [cs.PL] https://arxiv.org/abs/2412.13581

Agnishom Chattopadhyay, Wu Angela Li, and Konstantinos Mamouras. 2025. Verified and Efficient Matching of Regular Expressions with Lookaround. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, Kathrin Stark, Amin Timany, Sandrine Blazy, and Nicolas Tabareau (Eds.). ACM, 198–213. doi:10.1145/3703595.3705884

Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving string constraints with Regex-dependent functions through transducers with priorities and variables. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31. doi:10.1145/3498707

Nariyoshi Chida and Tachio Terauchi. 2023. Repairing Regular Expressions for Extraction. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1633–1656. doi:10.1145/3591287

Nariyoshi Chida and Tachio Terauchi. 2024. Repairing Regex-Dependent String Functions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 294–305. doi:10.1145/3691620.3695005

Thierry Coquand and Vincent Siles. 2011. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Certified Programs and Proofs - First International Conference, CPP 2011*. doi:10.1007/978-3-642-25379-9_11

Russ Cox. 2009. Regular Expression Matching: the Virtual Machine Approach. https://swtch.com/~rsc/regexp/regexp2.html.

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018*. ACM, 246–256. doi:10.1145/3236024.3236027

Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024. A Coq Mechanization of JavaScript Regular Expression Semantics. *Proc. ACM Program. Lang.* 8, ICFP (2024), 1003–1031. doi:10.1145/3674666

Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. 2013. A Constructive Theory of Regular Languages in Coq. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, 82–97. doi:10.1007/978-3-319-03545-1_6

Christian Doczkal and Gert Smolka. 2018. Regular Language Representations in the Constructive Type Theory of Coq. *J. Autom. Reason.* 61, 1-4 (2018), 521–553. doi:10.1007/S10817-018-9460-X

Mark Jason Dominus. 2000. Perl Regular Expression Matching is NP-Hard. https://perl.plover.com/NPC/NPC-3SAT.html.

Danny Dubé and Marc Feeley. 2000. Efficiently building a parse tree from a regular expression. *Acta Informatica* 37, 2 (2000), 121–144. doi:10.1007/S002360000037

ECMA. 2023. ECMA-262, 14th edition: ECMAScript® 2023 Language Specification. https://262.ecma-international.org/14.0/

ECMA. 2024. ECMA-262, 15th edition: ECMAScript® 2024 Language Specification. https://262.ecma-international.org/15.0/

ECMA. 2025. ECMA-262, 16th edition: ECMAScript® 2025 Language Specification. https://262.ecma-international.org/16.0/

Derek Egolf, Sam Lasser, and Kathleen Fisher. 2021. Verbatim: A Verified Lexer Generator. In *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. IEEE, 92–100. doi:10.1109/SPW53761.2021.00022

Derek Egolf, Sam Lasser, and Kathleen Fisher. 2022. Verbatim++: verified, optimized, and semantically rich lexing with derivatives. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 27–39. doi:10.1145/3497775.3503694

Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rümmer, and Andrei Sabelfeld. 2023. Black Ostrich: Web Application Scanning with String Solvers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 549–563. doi:10.1145/3576915.3616582

Denis Firsov and Tarmo Uustalu. 2013. Certified Parsing of Regular Languages. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, 98–113. doi:10.1007/978-3-319-03545-1_7

Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory Comput. Syst.* 53, 2 (2013), 159–193. doi:10.1007/S00224-012-9389-0

Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings (Lecture Notes in Computer Science,*

Vol. 3142), Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, 618–629. doi:10.1007/978-3-540-27836-8_53

Hiroya Fujinami and Ichiro Hasuo. 2024. Efficient Matching with Memoization for Regexes with Look-around and Atomic Grouping. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14577)*, Stephanie Weirich (Ed.). Springer, 90–118. doi:10.1007/978-3-031-57267-8_4

Andrew Gallant. 2014. Crate regex: An implementation of regular expressions for Rust. https://docs.rs/regex/latest/regex/.

Go. 2025. GO Regexp package. https://pkg.go.dev/regexp.

Google. 2010. RE2: A fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in PCRE, Perl, and Python. https://github.com/google/re2.

Philip Hazel. 1997. PCRE - Perl Compatible Regular Expressions. https://www.pcre.org/.

Philip Hazel. 2012. PCRE Library Functions Manual. https://pcre.org/pcre.txt.

Iain Ireland. 2020. A New RegExp Engine in SpiderMonkey. https://hacks.mozilla.org/2020/06/a-new-regexp-engine-in-spidermonkey/.

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 397–416. doi:10.1007/978-3-642-28869-2_20

Stefan Kahrs and Colin Runciman. 2022. Simplifying regular expressions further. *J. Symb. Comput.* 109 (2022), 124–143. doi:10.1016/J.JSC.2021.08.003

Ohad Kammar and Katarzyna Marek. 2023. Idris TyRE: a dependently typed regex parser. *CoRR* abs/2305.04480 (2023). doi:10.48550/ARXIV.2305.04480 arXiv:2305.04480

Alexander Krauss and Tobias Nipkow. 2010. Regular Sets and Expressions. *Archive of Formal Proofs* (May 2010). https://isa-afp.org/entries/Regular-Sets.html, Formal proof development.

Alexander Krauss and Tobias Nipkow. 2012. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *J. Autom. Reason.* 49, 1 (2012), 95–106. doi:10.1007/S10817-011-9223-4

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 425–438. doi:10.1145/3314221.3314645

Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proc. ACM Program. Lang.* 8, POPL (2024), 2761–2791. doi:10.1145/3632934

Nelma Moreira, David Pereira, and Simão Melo de Sousa. 2012. Deciding Regular Expressions (In-)Equivalence in Coq. In *Relational and Algebraic Methods in Computer Science - 13th International Conference, RAMiCS 2012, Cambridge, UK, September 17-20, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7560)*, Wolfram Kahl and Timothy G. Griffin (Eds.). Springer, 98–113. doi:10.1007/978-3-642-33314-9_7

Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1026–1049. doi:10.1145/3591262

Lasse Nielsen and Fritz Henglein. 2011. Bit-coded Regular Expression Parsing. In *Language and Automata Theory and Applications - 5th International Conference, LATA 2011, Tarragona, Spain, May 26-31, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6638)*, Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide (Eds.). Springer, 402–413. doi:10.1007/978-3-642-21254-3_32

Tobias Nipkow and Dmitriy Traytel. 2014. Unified Decision Procedures for Regular Expression Equivalence. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8558)*, Gerwin Klein and Ruben Gamboa (Eds.). Springer, 450–466. doi:10.1007/978-3-319-08970-6_29

Taisei Nogami and Tachio Terauchi. 2023. On the Expressive Power of Regular Expressions with Backreferences. In *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France (LIPIcs, Vol. 272)*, Jérôme Leroux, Sylvain Lombardy, and David Peleg (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 71:1–71:15. doi:10.4230/LIPICS.MFCS.2023.71

Wendlasida Ouedraogo, Gabriel Scherer, and Lutz Straßburger. 2024. Coqlex: Generating Formally Verified Lexers. *Art Sci. Eng. Program.* 8, 1 (2024). doi:10.22152/PROGRAMMING-JOURNAL.ORG/2024/8/3

Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *Journal of Functional Programming* 19, 2 (2009), 173–190.

Scott Owens and Konrad Slind. 2008. Adapting functional programs to higher order logic. *High. Order Symb. Comput.* 21, 4 (2008), 377–409. doi:10.1007/S10990-008-9038-0

Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 647–658. doi:10.1145/3324884.3416632

Rob Pike. 1987. The Text Editor sam. *Softw. Pract. Exp.* 17, 11 (1987), 813–845. doi:10.1002/SPE.4380171105

Gabriel Radanne. 2019. Typed parsing and unparsing for untyped regular expression engines. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 35–46. doi:10.1145/3294032.3294082

Rodrigo Geraldo Ribeiro and André Rauber Du Bois. 2017. Certified Bit-Coded Regular Expression Parsing. In *Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017, Fortaleza, CE, Brazil, September 21-22, 2017*, Fabio Mascarenhas (Ed.). ACM, 4:1–4:8. doi:10.1145/3125374.3125381

Dmitry Soshnikov. 2025. `regexp-tree`. https://github.com/DmitrySoshnikov/regexp-tree.

Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium, USENIX Security 2018*. USENIX Association, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

Peter Thiemann. 2016. Derivatives for Enhanced Regular Expressions. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9705)*, Yo-Sub Han and Kai Salomaa (Eds.). Springer, 285–297. doi:10.1007/978-3-319-40946-7_24

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

Christian Urban. 2023. POSIX Lexing with Derivatives of Regular Expressions. *J. Autom. Reason.* 67, 3 (2023), 24. doi:10.1007/S10817-023-09667-1

V8. 2021. An Additional Non-backtracking RegExp Engine. https://v8.dev/blog/non-backtracking-regexp.

V8. 2023. Mismatch between the Experimental engine and the backtracking engine on empty repetitions. https://issues.chromium.org/issues/42204037.

V8. 2025. RegExp: Inconsistent branch priority in look-behind assertions. https://issues.chromium.org/issues/388290816.

Ian Erik Varatalu, Margus Veanes, and Juhan-Peep Ernits. 2023. Derivative Based Extended Regular Expression Matching Supporting Intersection, Complement and Lookarounds. *CoRR* abs/2309.14401 (2023). doi:10.48550/ARXIV.2309.14401 arXiv:2309.14401

Ian Erik Varatalu, Margus Veanes, and Juhan P. Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proc. ACM Program. Lang.* 9, POPL (2025), 1–32. doi:10.1145/3704837

Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. 2024. Lean Formalization of Extended Regular Expression Matching with Lookarounds. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 118–131. doi:10.1145/3636501.3636959

Received 2025-07-10; accepted 2025-11-06