A Coq Mechanization of JavaScript Regular Expression Semantics

Noé De Santo, Aurèle Barrière, Clément Pit-Claudel

SYSTEMF · EPFL

Our Project: Formally Verified Linear Engines for JavaScript Regexes

- Algorithms.
- Specification.
- Proof of correctness.

Our Project: Formally Verified Linear Engines for JavaScript Regexes

- Algorithms. Done : [PLDI24]
- Specification.
- Proof of correctness.

Our Project: Formally Verified Linear Engines for JavaScript Regexes

- Algorithms. Done : [PLDI24]
- Specification. This work
- Proof of correctness. Future work

Our Project: Formally Verified Linear Engines for JavaScript Regexes

We need a mechanized specification for JavaScript Regexes.

Textbook Regex Specification:

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

Extended with JavaScript regex features: [PLDI19, PLDI23]

Our Project: Formally Verified Linear Engines for JavaScript Regexes

We need a mechanized specification for JavaScript Regexes.

Official Specification:

The ECMAScript standard.

Regex Chapter:
33 pages of pseudocode for
a backtracking algorithm.

Textbook Regex Specification:

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

Extended with JavaScript regex features:
[PLDI19, PLDI23]

Our Project: Formally Verified Linear Engines for JavaScript Regexes

We need a mechanized specification for JavaScript Regexes.

Official Specification:

The ECMAScript standard.

Regex Chapter:
33 pages of pseudocode for
a backtracking algorithm.

Hard to audit.

Textbook Regex Specification:

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

Extended with JavaScript regex features:
[PLDI19, PLDI23]: **incomplete** & **incorrect**.

Our Project: Formally Verified Linear Engines for JavaScript Regexes

We need a mechanized specification for JavaScript Regexes.

Official Specification:

The ECMAScript standard.

Regex Chapter:
33 pages of pseudocode for a backtracking algorithm.

Hard to audit.

Textbook Regex Specification:

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

Extended with JavaScript regex features:
[PLDI19, PLDI23]: **incomplete** & **incorrect**.

Easy to audit.

Our Mechanized Specification in Coq

We prioritize **auditability** and **faithfulness**.

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

Features

$r ::=$	a	Characters
	$r_1\ r_2$	Concatenation
	$r_1 \mid r_2$	Disjunction
	r^*	Iteration (Kleene star)

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

New Features

r ::=	a	Characters	
	$r_1\ r_2$	Concatenation	
	$r_1 \mid r_2$	Disjunction	
	r^*	Iteration (Kleene star)	
	(r)	**Capture Group**	$\longrightarrow$ Return the substring last matched by subexpressions in parentheses: `"aab".match(/((a*)b)/) = ("aab", "aa")`.

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

New Features

$r ::=$	a	Characters
	$r_1\ r_2$	Concatenation
	$r_1 \mid r_2$	**Disjunction**
	r^*	Iteration (Kleene star)
	(r)	Capture Group

Disjunction is not commutative!
`"ab".match(/(a|ab)/) = "a"`.

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

New Features

$r ::=$	a	Characters
	$r_1\ r_2$	Concatenation
	$r_1 \mid r_2$	Disjunction
	r^*	Iteration (Kleene star)
	(r)	Capture Group
	$(? <= r)$	**Lookbehind**

$\longrightarrow$ Traverse the string backwards:

`"ICFP24".match(/(?<=ICFP)\d+/) = "24".`

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

New Features

r ::=	a	Characters
	$r_1\ r_2$	Concatenation
	$r_1 \mid r_2$	Disjunction
	r^*	Iteration (Kleene star)
	(r)	Capture Group
	$(? <= r)$	Lookbehind
	$\$$	Anchor
	$(? = r)$	Lookahead
	$\backslash 1$	Backreference
	$r\{n, m\}$	Counted Quantifier
	$r^{*?}$	Lazy Quantifier
	...	...

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

New Features

$r ::=$	a	Characters
	$r_1\ r_2$	Concatenation
	$r_1 \mid r_2$	Disjunction
	r^*	Iteration (Kleene star)
	(r)	Capture Group
	$(? <= r)$	Lookbehind
	$\$$	Anchor
	$(? = r)$	Lookahead
	$\backslash 1$	Backreference
	$r\{n, m\}$	Counted Quantifier
	$r^{*?}$	Lazy Quantifier
	...	...

Flags

i	Case Insentivity
u	Unicode Mode
m	Multiline Mode
g	Global
d	Return Group Indices
y	Sticky Matching
s	Dot Matches All

Regexes are Natively Supported in JavaScript

30% of npm packages use regexes [FSE18].

```
> /a*b/.test("aaab")
true
> /(a|ab)c/.test("ad")
false
```

New Features

r ::=	a	Characters
	$r_1\ r_2$	Concatenation
	$r_1 \mid r_2$	Disjunction
	r^*	Iteration (Kleene star)
	(r)	Capture Group
	$(? <= r)$	Lookbehind
	$\$$	Anchor
	$(? = r)$	Lookahead
	\1	Backreference
	$r\{n, m\}$	Counted Quantifier
	$r^{*?}$	Lazy Quantifier
	...	...

Flags

i	Case Insentivity
u	Unicode Mode
m	Multiline Mode
g	Global
d	Return Group Indices
y	Sticky Matching
s	Dot Matches All

It's hard to build a JavaScript regex engine

- Firefox uses Chrome's [Mozilla20].
- We previously found bugs in Chrome's linear-time engine [PLDI24].

The ECMAScript Regex Chapter

- **Parsing**. Parse the regex.
- **Early errors**. Check that the regex is well-formed (no duplicate groups, undefined backreferences...).
- **Compilation**. Turn the regex into a backtracking *matcher* pseudocode function.

22.2.2.3 Runtime Semantics: CompileSubpattern

The syntax-directed operation CompileSubpattern takes arguments *rer* (a RegExp Record) and *direction* (forward or backward) and returns a Matcher.

Disjunction :: *Alternative* | *Disjunction*

1. Let *m1* be CompileSubpattern of *Alternative* with arguments *rer* and *direction*.
2. Let *m2* be CompileSubpattern of *Disjunction* with arguments *rer* and *direction*.
3. Return a new Matcher with parameters (*x*, *c*) that captures *m1* and *m2* and performs the following steps when called:
   a. Assert: *x* is a MatchState.
   b. Assert: *c* is a MatcherContinuation.
   c. Let *r* be *m1*(*x*, *c*).
   d. If *r* is not failure, return *r*.
   e. Return *m2*(*x*, *c*).

33 pages of specification

22.2.2.4.1 IsWordChar (rer, Input, e)

The abstract operation IsWordChar takes arguments rer (a RegExp Record), Input (a List of characters), and e (an integer) and returns a Boolean.
It performs the following steps when called:

1. Let InputLength be the number of elements in Input.
2. If e = -1 or e = InputLength, return false.
3. Let c be the character Input[e].
4. If WordCharacters(rer) contains c, return true.
5. Return false.

```
(** >>
   22.2.2.4.1 IsWordChar ( rer, Input, e )

   The abstract operation IsWordChar takes arguments rer (a RegExp Record), Input
   (a List of characters), and e (an integer) and returns a Boolean.
   It performs the following steps when called:
<<*)

  (*>> 1. Let InputLength be the number of elements in Input. <<*)

  (*>> 2. If e = -1 or e = InputLength, return false. <<*)


  (*>> 3. Let c be the character Input[ e ]. <<*)

  (*>> 4. If WordCharacters(rer) contains c, return true. <<*)



  (*>> 5. Return false. <<*)
```

```
(** >>
   22.2.2.4.1 IsWordChar ( rer, Input, e )

   The abstract operation IsWordChar takes arguments rer (a RegExp Record), Input
   (a List of characters), and e (an integer) and returns a Boolean.
   It performs the following steps when called:
<<*)
Definition isWordChar (rer: RegExpRecord) (Input: list Character) (e: integer): Result bool :=
  (*>> 1. Let InputLength be the number of elements in Input. <<*)
  let InputLength := List.length Input in
  (*>> 2. If e = -1 or e = InputLength, return false. <<*)
  if (e =? -1)%Z || (e =? InputLength)%Z then false
  else
  (*>> 3. Let c be the character Input[ e ]. <<*)
  let! c =<< Input[e] in
  (*>> 4. If WordCharacters(rer) contains c, return true. <<*)
  let! wc =<< wordCharacters rer in
  if CharSet.contains wc c then true
  else
  (*>> 5. Return false. <<*)
  false.
```

Mechanization Challenges

Failing Operations Examples

None of these are expected to fail.

In ECMAScript

```
Let ch be the character Input[index]
Assert: i<=j
```

Failing Operations Examples

None of these are expected to fail.

In ECMAScript

```
Let ch be the character Input[index]
Assert: i<=j
```

Encoding Failure with the Error Monad

- Scales well to the entire regex chapter.
- No dependent types!
- Easy to audit.

Failing Operations Examples

None of these are expected to fail.

In ECMAScript

```
Let ch be the character Input[index]
Assert: i<=j
```

In Coq

```
let! ch =<< Input[ index ] in
assert! (i <=? j)
```

Encoding Failure with the Error Monad

- Scales well to the entire regex chapter.
- No dependent types!
- Easy to audit.

```
(* Simplified: used to implement quantifiers such as the star *)
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) :=
 let d := fun (y: MatchState) ⇒
  if min = 0 and endIndex(y) = endIndex(x) then mismatch
  else
  let nextmin := if min = 0 then 0 else min - 1 in
   RepeatMatcher m nextmin y c
 in ...
```

Some functions use non-structural recursion

```
(* Simplified: used to implement quantifiers such as the star *)
Fixpoint RepeatMatcherFuel (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) (fuel: nat) :=
 match fuel with
 | 0 ⇒ OutOfFuel
 | S fuel' ⇒
  let d := fun (y: MatchState) ⇒
   if min = 0 and endIndex(y) = endIndex(x) then mismatch
   else
   let nextmin := if min = 0 then 0 else min - 1 in
   RepeatMatcher m nextmin y c fuel'
  in ...

Definition RepeatMatcher m min x c :=
 RepeatMatcherFuel m min x c (compute_fuel min x).
```

Some functions use non-structural recursion

We add a fuel argument to these functions.
We can compute an initial amount of fuel to provide to the function.

CountLeftCapturingParensBefore (***node*** **)**

1. Assert: *node* is an instance of a production in the RegExp Pattern grammar.
2. Let *pattern* be the *Pattern* containing *node*.
3. Return the number of *Atom* :: `(` *GroupSpecifier*$_{\texttt{opt}}$ *Disjunction* `)` Parse Nodes contained within *pattern* that either occur before *node* or contain *node*.

We need to remember the original regex (Pattern) and the position of the current node within it.

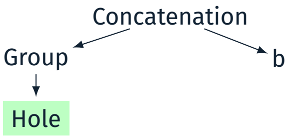
CountLeftCapturingParensBefore (*node*)

1. Assert: *node* is an instance of a production in the RegExp Pattern grammar.
2. Let *pattern* be the *Pattern* containing *node*.
3. Return the number of *Atom* :: (*GroupSpecifier*$_{opt}$ *Disjunction*) Parse Nodes contained within *pattern* that either occur before *node* or contain *node*.

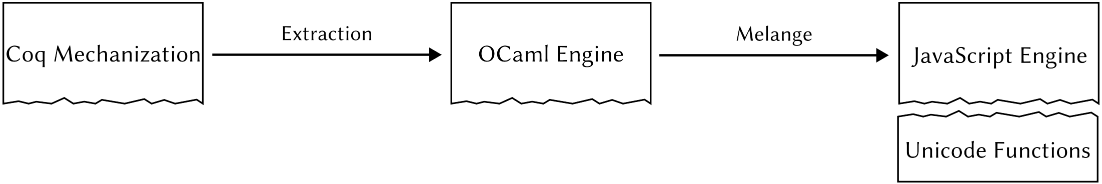
We need to remember the original regex (Pattern) and the position of the current node within it.

We add a Zipper Context

Missing argument: the original regex AST with a hole.

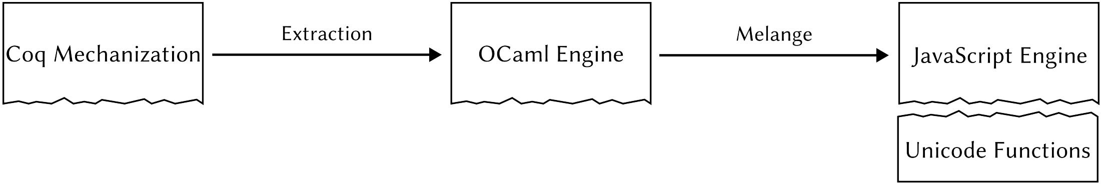



```
Definition countLeftCapturingParensBefore (node) (ctx: RegexContext):=
```



Unicode Parameterization

Our mechanization is parameterized by a character type and character manipulation functions.



Unicode Parameterization

Our mechanization is parameterized by a character type and character manipulation functions.

Checking our mechanization

We ran the Test262 official JavaScript conformance test suite.
495/498 tests passed. 3 timeouts.

Mechanized proofs about the ECMAScript semantics

Matching never fails

Every assertion holds:

```
Theorem no_failure :
  (* For regex r and string s *)
  ∀ r m s, compileSubPattern r = Success m →
  earlyErrors r = OK →
  (* the matcher cannot fail an assertion. *)
  m (init_state s) (identity_cont) ≠ Failure.
```

Matching never fails

Every assertion holds:

```
Theorem no_failure :
  (* For regex r and string s *)
  ∀ r m s, compileSubPattern r = Success m →
  earlyErrors r = OK →
  (* the matcher cannot fail an assertion. *)
  m (init_state s) (identity_cont) ≠ Failure.
```

Matching always terminates

The initial fuel we provide is always enough:

```
Theorem termination :
  (* For all regex r and string s *)
  ∀ r m s, compileSubPattern r = Success m →
  earlyErrors r = OK →
  (* the matcher cannot run out of fuel. *)
  m (init_state s) (identity_cont) ≠ OutOfFuel.
```

Strictly Nullable Regex

A regex that cannot match any character (made with ϵ, lookarounds, anchors).

A V8 optimization

When r is strictly-nullable, r^* can be replaced by ϵ.

Strictly Nullable Regex

A regex that cannot match any character (made with ϵ, lookarounds, anchors).

A V8 optimization

When r is strictly-nullable, r^* can be replaced by ϵ.

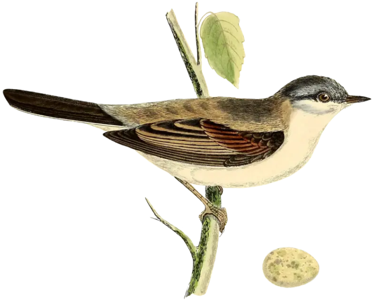
A Coq Proof

```
Theorem strictly_nullable_same_matcher:
  ∀ (r:Regex) (mstar: Matcher) (mepsilon: Matcher),
    strictly_nullable r = true →
    compileSubPattern (Star r) = Success mstar →
    compileSubPattern Epsilon = Success mepsilon →
      mstar = mepsilon.
```

Warblre, a Coq Mechanization of JavaScript regexes

`https://github.com/epfl-systemf/Warblre`

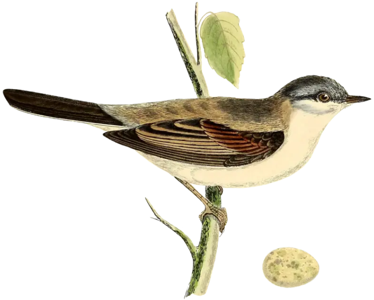
- **Auditable** : line-for-line correspondence with ECMAScript.
- **Faithful** : passes the relevant tests of Test262.
- **Executable** : OCaml and JavaScript extracted engines.
- **Proven-Safe** : proofs of non-failure and termination.



Warblre, a Coq Mechanization of JavaScript regexes

`https://github.com/epfl-systemf/Warblre`

- **Auditable** : line-for-line correspondence with ECMAScript.
- **Faithful** : passes the relevant tests of Test262.
- **Executable** : OCaml and JavaScript extracted engines.
- **Proven-Safe** : proofs of non-failure and termination.



Future Work

- Done by Martin Crettol. Check that the Coq comments exactly correspond to the regex chapter, using SpecMerger: `https://github.com/epfl-systemf/SpecMerger`.
- June 2024 edition: new unicode v flag with unicode string properties.
- Prove it equivalent to some textbook style semantics.
- Formally verified regex engines.

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

+ substring match

$$\frac{r_1;\ v \mathbin{@} n \vdash m}{r_1 \mid r_2;\ v \mathbin{@} n \vdash m} \qquad \frac{r_2;\ v \mathbin{@} n \vdash m}{r_1 \mid r_2;\ v \mathbin{@} n \vdash m}$$

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

+ substring match + match priority

$$\frac{r_1;\ v\ @\ n \vdash m}{r_1 \mid r_2;\ v\ @\ n \vdash m} \qquad \frac{r_1;\ v\ @\ n \vdash \bot \quad r_2;\ v\ @\ n \vdash m}{r_1 \mid r_2;\ v\ @\ n \vdash m}$$

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

+ substring match + match priority + capturing groups and backreferences

$$\frac{r_1;\ \Sigma;\ v @ n \vdash \Sigma';\ m}{r_1 \mid r_2;\ \Sigma;\ v @ n \vdash \Sigma';\ m} \qquad \frac{r_1;\ \Sigma;\ v @ n \vdash \bot \quad r_2;\ \Sigma;\ v @ n \vdash \Sigma';\ m}{r_1 \mid r_2;\ \Sigma;\ v @ n \vdash \Sigma';\ m}$$

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

+ substring match + match priority + capturing groups and backreferences

$$\frac{r_1;\ \Sigma;\ v \mathbin{@} n \vdash \Sigma';\ m}{r_1 \mid r_2;\ \Sigma;\ v \mathbin{@} n \vdash \Sigma';\ m} \qquad \frac{r_1;\ \Sigma;\ v \mathbin{@} n \vdash \bot \quad r_2;\ \Sigma;\ v \mathbin{@} n \vdash \Sigma';\ m}{r_1 \mid r_2;\ \Sigma;\ v \mathbin{@} n \vdash \Sigma';\ m}$$

Can we be sure this corresponds to the ECMAScript definitions?

- This is far from the ECMAScript pseudocode.
- This is difficult to mechanize (non-strict positivity).
- [PLDI19, PLDI23] have tried that. Both **incomplete** and **incorrect**.

$$\frac{r_1 \vdash v}{r_1 \mid r_2 \vdash v} \qquad \frac{r_2 \vdash v}{r_1 \mid r_2 \vdash v}$$

+ substring match + match priority + capturing groups and backreferences

$$\frac{r_1;\ \Sigma;\ v\ @\ n \vdash \Sigma';\ m}{r_1 \mid r_2;\ \Sigma;\ v\ @\ n \vdash \Sigma';\ m} \qquad \frac{r_1;\ \Sigma;\ v\ @\ n \vdash \bot \quad r_2;\ \Sigma;\ v\ @\ n \vdash \Sigma';\ m}{r_1 \mid r_2;\ \Sigma;\ v\ @\ n \vdash \Sigma';\ m}$$

Can we be sure this corresponds to the ECMAScript definitions?

- This is far from the ECMAScript pseudocode.
- This is difficult to mechanize (non-strict positivity).
- [PLDI19, PLDI23] have tried that. Both **incomplete** and **incorrect**.

Incorrect rule			Counter-example
$r?$	$\equiv$	$r \mid \epsilon$	()? on the empty string
$r??$	$\equiv$	$\epsilon \mid r$	$(? = (a))??ab\backslash 1c$ on string "abac"

References I

PLDI24	Linear Matching of JavaScript Regular Expressions.
PLDI19	Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript.
PLDI23	Repairing Regular Expressions for Extraction.
FSE18	The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale.
Mozilla20	A New RegExp Engine in SpiderMonkey.