Mechanized semantics for ECMAScript regexes

Master Project Defense

Noé De Santo

EPFL

February 15, 2024

Introduction

Regular expressions

Regular expressions as originally defined by Kleene [6] and used in automata theory [3, 4].

Regular expressions

Regular expressions as originally defined by Kleene [6] and used in automata theory [3, 4].

$$a \in \Sigma \qquad r ::= a \quad || \quad \varepsilon \quad || \quad r_1 r_2 \quad || \quad r_1 \mid r_2 \quad || \quad r^*$$

Regular expressions

Regular expressions as originally defined by Kleene [6] and used in automata theory [3, 4].

$$a \in \Sigma \qquad r ::= a \quad || \quad \varepsilon \quad || \quad r_1 r_2 \quad || \quad r_1 \mid r_2 \quad || \quad r^*$$

$$\frac{}{a \vdash \text{“a”}} \qquad \frac{}{\varepsilon \vdash \text{“”}} \qquad \frac{r_1 \vdash v \quad r_2 \vdash w}{r_1 r_2 \vdash vw}$$

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

$$\frac{}{r^* \vdash \text{“”}} \qquad \frac{r \vdash v \quad r^* \vdash w}{r^* \vdash vw}$$

Regexes are not Regular Expressions (anymore)

Regexes as in PCRE2,Perl,JavaScript,Java,.NET,Python,Rust,...

Regexes are not Regular Expressions (anymore)

Regexes as in PCRE2,Perl,JavaScript,Java,.NET,Python,Rust,...

$$
\begin{array}{rcl}
r ::= & & a \\
& || & [a_1 - a_2] \\
& || & \backslash b \quad || \quad \ldots \\
& || & r_1 r_2 \\
& || & r_1 \mid r_2 \\
& || & r^* \quad || \quad r^+ \quad || \quad r^? \quad || \quad r^{\{\mathbb{N},\mathbb{N}\}} \\
& || & r^* \quad || \quad r^{+?} \quad || \quad r^{??} \quad || \quad r^{\{\mathbb{N},\mathbb{N}\}?} \\
& || & (r) \quad || \quad (_{<\text{name}>} r) \quad || \\
& || & \backslash\mathbb{N} \quad || \quad \backslash<\text{name}> \\
& || & (?= r) \quad || \quad (?\neq r) \quad || \quad (?\leq r) \quad || \quad (?\not\leq r)
\end{array}
$$

Regexes are not Regular Expressions (anymore)

$$
\begin{array}{l}
\| \quad \hat{} \quad \| \quad \$ \quad \| \quad \backslash b \quad \| \quad \backslash B \\
\| \quad (?\!> r) \\
\| \quad (_{<\text{name1-name2}>} r) \\
\| \quad (?\mathbb{N}) \quad \| \quad (?+\mathbb{N}) \quad \| \quad (?-\mathbb{N}) \quad \| \\
\| \quad (?C_{\text{name}}) \\
\| \quad \ldots
\end{array}
$$

Differences between regex languages

What about semantics?

Differences between regex languages

What about semantics?

- $a \mid ab$ on "ab".

Differences between regex languages

What about semantics?

- $a \mid ab$ on "ab".
  - Posix (ERE): " a b "
  - JavaScript: " a b "

Differences between regex languages

What about semantics?

- $a \mid ab$ on "ab".
  - Posix (ERE): " $\underbrace{\text{a b}}$ "
  - JavaScript: " $\underbrace{\text{a}}$ b "
- $(?: (a) \mid b)^*$ on "ab".

Differences between regex languages

What about semantics?

- $a \mid ab$ on “ab”.
  - Posix (ERE): “ $\underline{a\ b}$ ”
  - JavaScript: “ $\underline{a}$ b ”

- $(?{:}\ (a) \mid b)^*$ on “ab”.
  - Python,Rust,.Net,...: “ $\underline{\overbrace{a}^{1}\ b}$ ”
  - JavaScript: “ $\underline{a\ b}$ ” (nothing was captured)

A specification of Regexes

- JavaScript has a specification: ECMAScript.

A specification of Regexes

- JavaScript has a specification: ECMAScript.
- ECMAScript specifies the semantics of its regexes.

Goals & benefits

Goal: mechanize the section about regexes of the ECMAScript specification in the Coq proof assistant.

Goals & benefits

Goal: mechanize the section about regexes of the ECMAScript specification in the Coq proof assistant.

Mechanization benefits:

- Foundation for formal reasoning about JavaScript regexes;

Goals & benefits

Goal: mechanize the section about regexes of the ECMAScript specification in the Coq proof assistant.

Mechanization benefits:

- Foundation for formal reasoning about JavaScript regexes;
- Provide a better understanding of implicit invariants of the specification;

Goals & benefits

Goal: mechanize the section about regexes of the ECMAScript specification in the Coq proof assistant.

Mechanization benefits:

- Foundation for formal reasoning about JavaScript regexes;
- Provide a better understanding of implicit invariants of the specification;
- An executable “ground truth”.

State of the art

- The ECMAScript specification has already been mechanized many times [1, 10, 8, 9]...

State of the art

- The ECMAScript specification has already been mechanized many times [1, 10, 8, 9]... but the section on regexes was not.

State of the art

- The ECMAScript specification has already been mechanized many times [1, 10, 8, 9]... but the section on regexes was not.
- Alternative equivalent semantics for regexes are sometimes defined in the literature [7, 2]...

State of the art

- The ECMAScript specification has already been mechanized many times [1, 10, 8, 9]... but the section on regexes was not.
- Alternative equivalent semantics for regexes are sometimes defined in the literature [7, 2]... but these typically get details wrong, e.g.

$$e^? \equiv e \mid \varepsilon$$

does not hold.

Results

A reasonably future-proof, proven-safe, executable mechanization of the ECMAScript regexes.

Results

A reasonably future-proof, proven-safe, executable mechanization of the ECMAScript regexes.

- Mechanizing the ECMAScript regex specification in the Coq proof assistant (section 3);

Results

A reasonably future-proof, proven-safe, executable mechanization of the ECMAScript regexes.

- Mechanizing the ECMAScript regex specification in the Coq proof assistant (section 3);
- Using this mechanized specification: proofs (section 4) that
  - Matching always terminate;
  - No operation ever fails, e.g.
    - Assertions;
    - List indexing.

Results

A reasonably future-proof, proven-safe, executable mechanization of the ECMAScript regexes.

- Mechanizing the ECMAScript regex specification in the Coq proof assistant (section 3);
- Using this mechanized specification: proofs (section 4) that
  - Matching always terminate;
  - No operation ever fails, e.g.
    - Assertions;
    - List indexing.
- Extracting an executable engine in OCaml (section 5).

ECMA Regexes

Features: overview

Feature name	Syntax
Character	a, b, c, $\backslash n$, ...
Character classes	$[abc]$, $[\hat{}A-Z]$, $\cdot$, $\backslash d$, ...
Sequence/concatenation	$r_1 r_2$
Disjunction/union	$r_1 \mid r_2$
Greedy quantifiers	r^*, r^+, $r^?$, $r^{\{\mathbb{N},\mathbb{N}\}}$, ...
Lazy quantifiers	$r^{*?}$, $r^{+?}$, $r^{??}$, $r^{\{\mathbb{N},\mathbb{N}\}?}$, ...
Capturing groups	(r), $(_{<\text{name}>} r)$
Non-capturing groups	$(?{:}\ r)$
Backreferences	$\backslash\mathbb{N}$, $\backslash<\text{name}>$
Lookarounds	$(?{=}\ r)$, $(?{\leq}\ r)$, $(?{\neq}\ r)$, $(?{\not\leq}\ r)$
Anchors	$\hat{}$, \$, \b, \B

Features: close-up view

- **Quantifiers:** Allow to repeat another regex repeatedly, e.g. *, $^+$, and $^?$. ECMAScript also offers *bounded* quantifiers. E.g. $a^{\{4,\}}$ matches 'a' as many times as possible, and at least 4 times.

"aa"	χ
"aaaa"	$\checkmark$
"aaaaaaaaaaa"	$\checkmark$

Features: close-up view

- **Quantifiers:** Allow to repeat another regex repeatedly, e.g. *, $^+$, and $^?$. ECMAScript also offers *bounded* quantifiers.
- **Capturing groups:** Allow to retrieve strings matched by sub-parts of the regex.

  E.g. $((a^*)(b^*))$ on "aaba" yields "$\underbrace{\overbrace{\text{a a}}^{2}\,\overbrace{\text{b}}^{3}}_{1}$ a "

Features: close-up view

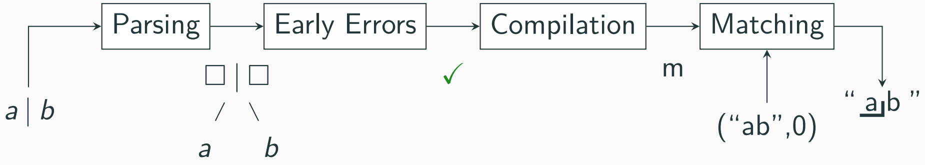
- **Quantifiers:** Allow to repeat another regex repeatedly, e.g. *, $^+$, and $^?$. ECMAScript also offers *bounded* quantifiers.
- **Capturing groups:** Allow to retrieve strings matched by sub-parts of the regex.
- **Non-capturing groups:** Allow to override the operators' precedence. E.g. $(?{:}\, a \mid b)c \quad \not\equiv \quad a \mid bc \quad \equiv \quad a \mid (?{:}\, bc)$

Pipeline

- ECMAScript regexes: ~50 pages; pseudo-code for a backtracking-based matching algorithm.

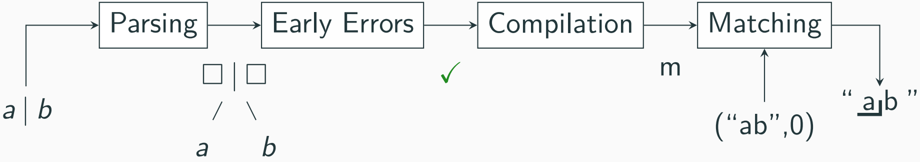
Pipeline

- ECMAScript regexes: ~50 pages; pseudo-code for a backtracking-based matching algorithm.
- Describes the following pipeline:



Pipeline

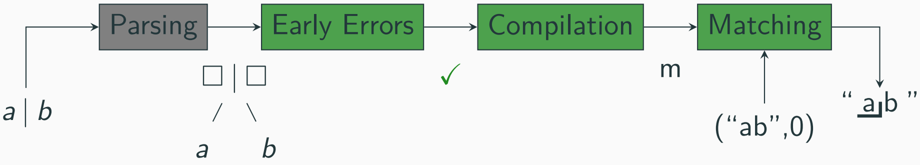
- ECMAScript regexes: ~50 pages; pseudo-code for a backtracking-based matching algorithm.
- Describes the following pipeline:



Early errors: $a^{\{3,2\}}$, $(a)\backslash 2$

Pipeline

- ECMAScript regexes: ~50 pages; pseudo-code for a backtracking-based matching algorithm.
- Describes the following pipeline:



The mechanization does not include Parsing.

In this presentation, we will focus on compilation and matching.

Compilation&matching: state

The specification needs types to represent the (internal) state of the match.

$$\texttt{MatchState} := (\texttt{String} * \texttt{EndIndex} * \texttt{Captures})$$
$$\texttt{MatchResult} := \texttt{MatchState}\ \texttt{or}\ \texttt{mismatch}$$

Compilation&matching: state

The specification needs types to represent the (internal) state of the match.

`MatchState` := (`String` ∗ `EndIndex` ∗ `Captures`)
`MatchResult` := `MatchState` or `mismatch`

E.g. `MatchState`:

“ $\text{a}\ \overbrace{\text{a}\ \text{b}}^{1}\ \text{b}\ \text{b}\ \text{a}\ \text{c}$ ”

Compilation&matching: state

The specification needs types to represent the (internal) state of the match.

$$\begin{aligned}\texttt{MatchState} &:= (\texttt{String} * \texttt{EndIndex} * \texttt{Captures})\\ \texttt{MatchResult} &:= \texttt{MatchState or mismatch}\end{aligned}$$

E.g. `MatchState`:

$$\text{“}\ \underline{\overbrace{\text{a a b}}^{1}}\ \text{b b a c}\ \text{”}$$

is represented as

$(\text{“aabbbac”}, 3, \{\#_1 \mapsto [1, 2], \#_2 \mapsto \texttt{undefined}\})$

Compilation&matching: state

The specification needs types to represent the (internal) state of the match.

$$\texttt{MatchState} \;:=\; (\texttt{String} * \texttt{EndIndex} * \texttt{Captures})$$
$$\texttt{MatchResult} := \;\texttt{MatchState}\ \text{or}\ \texttt{mismatch}$$

E.g. `MatchResult`:

$\texttt{Some}$ ("aabbbac", 3, $\{\#_1 \mapsto [1,2], \#_2 \mapsto \texttt{undefined}\}$)

or

$\texttt{mismatch}$ (a.k.a. $\texttt{None}$)

The compilation phase

The main compilation function: `compileSubpattern: Regex` $\rightarrow$ `Matcher`.

Defined recursively on the regex being compiled.

Compilation&matching: example

Consider (?: $a \mid b)c$. It could be compiled as follows:

Compilation&matching: example

Consider $(?: a \mid b)c$. It could be compiled as follows:

```
(* a | b *)
(* MatchState → MatcherContinuation → MatchResult *)
let m: Matcher := fun (x: MatchState) (k: MatcherContinuation) ⇒
    if next_char x = 'a' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else if next_char x = 'b' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else mismatch
```

Compilation&matching: example

Consider $(?: a \mid b)c$. It could be compiled as follows:

```
(* a | b *)
(* MatchState → MatcherContinuation → MatchResult *)
let m: Matcher := fun (x: MatchState) (k: MatcherContinuation) ⇒
    if next_char x = 'a' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else if next_char x = 'b' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else mismatch
(* c *)
(* MatchState → MatchResult *)
let k: MatcherContinuation := fun (x: MatchState) ⇒
    if next_char x = 'c' then Some (consume_char x)
    else mismatch
```

Compilation&matching: example

Consider $(?: a \mid b)c$. It could be compiled as follows:

```
(* a | b *)
(* MatchState → MatcherContinuation → MatchResult *)
let m: Matcher := fun (x: MatchState) (k: MatcherContinuation) ⇒
    if next_char x = 'a' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else if next_char x = 'b' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else mismatch
(* c *)
(* MatchState → MatchResult *)
let k: MatcherContinuation := fun (x: MatchState) ⇒
    if next_char x = 'c' then Some (consume_char x)
    else mismatch
```

A match would be found by calling, e.g. `m ("abc", 1, NoCaptures) k`.

Compilation&matching: example

Consider $(?: a \mid b)c$. It could be compiled as follows:

```
(* a | b *)
(* MatchState → MatcherContinuation → MatchResult *)
let m: Matcher := fun (x: MatchState) (k: MatcherContinuation) ⇒
    if next_char x = 'a' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else if next_char x = 'b' and k (consume_char x) ≠ mismatch
        then k (consume_char x)
    else mismatch
(* c *)
(* MatchState → MatchResult *)
let k: MatcherContinuation := fun (x: MatchState) ⇒
    if next_char x = 'c' then Some (consume_char x)
    else mismatch
```

A match would be found by calling, e.g. `m ("abc", 1, NoCaptures) k`.

Mechanization

Literate mechanization

```
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).
```

Literate mechanization

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*).

```
| Disjunction r1 r2 ⇒
  let! m1 =≪ compileSubPattern r1 (Disjunction_left r2 :: ctx) rer direction in
  let! m2 =≪ compileSubPattern r2 (Disjunction_right r1 :: ctx) rer direction in
  (fun (x: MatchState) (c: MatcherContinuation) ⇒
    let! r =≪ m1 x c in
    if r is not failure then
      r
    else
    m2 x c): Matcher
```

Literate mechanization

```
| Disjunction r1 r2 ⇒
  (*>> 1. Let m1 be CompileSubpattern of Alternative with arguments rer and direction. <<*)
  let! m1 =<< compileSubPattern r1 (Disjunction_left r2 :: ctx) rer direction in
  (*>> 2. Let m2 be CompileSubpattern of Disjunction with arguments rer and direction. <<*)
  let! m2 =<< compileSubPattern r2 (Disjunction_right r1 :: ctx) rer direction in
  (*>> 3. Return a new Matcher with parameters (x, c) that captures m1 and m2 and performs
     the following steps when called: <<*)
  (fun (x: MatchState) (c: MatcherContinuation) ⇒
    (*>> a. Assert: x is a MatchState. <<*)
    (*>> b. Assert: c is a MatcherContinuation. <<*)
    (*>> c. Let r be m1(x, c). <<*)
    let! r =<< m1 x c in
    (*>> d. If r is not failure, return r. <<*)
    if r is not failure then
      r
    else
    (*>> e. Return m2(x, c). <<*)
    m2 x c): Matcher
```

Literate mechanization

```
| Disjunction r1 r2 ⇒
  (*>> 1. Let m1 be CompileSubpattern of Alternative with arguments rer and direction. <<*)
  let! m1 =<< compileSubPattern r1 (Disjunction_left r2 :: ctx) rer direction in
  (*>> 2. Let m2 be CompileSubpattern of Disjunction with arguments rer and direction. <<*)
  let! m2 =<< compileSubPattern r2 (Disjunction_right r1 :: ctx) rer direction in
  (*>> 3. Return a new Matcher with parameters (x, c) that captures m1 and m2 and performs
     the following steps when called: <<*)
  (fun (x: MatchState) (c: MatcherContinuation) ⇒
    (*>> a. Assert: x is a MatchState. <<*)
    (*>> b. Assert: c is a MatcherContinuation. <<*)
    (*>> c. Let r be m1(x, c). <<*)
    let! r =<< m1 x c in
    (*>> d. If r is not failure, return r. <<*)
    if r is not failure then
      r
    else
    (*>> e. Return m2(x, c). <<*)
    m2 x c): Matcher
```

But of course not everything can be directly translated.

Problem 1: the specification uses fallible operations

Atom :: **(** *GroupSpecifier*$_{opt}$ *Disjunction* **)**

1. Let *m* be CompileSubpattern of *Disjunction* with arguments *rer* and *direction*.
2. Let *parenIndex* be CountLeftCapturingParensBefore(*Atom*).
3. Return a new Matcher with parameters (*x*, *c*) that captures *direction*, *m*, and *parenIndex* and performs the following steps when called:
   a. Assert: *x* is a MatchState.
   b. Assert: *c* is a MatcherContinuation.
   c. Let *d* be a new MatcherContinuation with parameters (*y*) that captures *x*, *c*, *direction*, and *parenIndex* and performs the following steps when called:
      i. Assert: *y* is a MatchState.
      ii. Let *cap* be a copy of *y*'s *captures* List.
      iii. Let *Input* be *x*'s *input*.
      iv. Let *xe* be *x*'s *endIndex*.
      v. Let *ye* be *y*'s *endIndex*.
      vi. If *direction* is forward, then
         1. Assert: *xe* ≤ *ye*.
         2. Let *r* be the CaptureRange (*xe*, *ye*).
      vii. Else,
         1. Assert: *direction* is backward.
         2. Assert: *ye* ≤ *xe*.
         3. Let *r* be the CaptureRange (*ye*, *xe*).
      viii. Set *cap*[*parenIndex* + 1] to *r*.
      ix. Let *z* be the MatchState (*Input*, *ye*, *cap*).
      x. Return *c*(*z*).
   d. Return *m*(*x*, *d*).

Problem 1: the specification uses fallible operations

Atom :: **(** *GroupSpecifier*$_{opt}$ *Disjunction* **)**

1. Let *m* be CompileSubpattern of *Disjunction* with arguments *rer* and *direction*.
2. Let *parenIndex* be CountLeftCapturingParensBefore(*Atom*).
3. Return a new Matcher with parameters (*x*, *c*) that captures *direction*, *m*, and *parenIndex* and performs the following steps when called:
   a. Assert: *x* is a MatchState.
   b. Assert: *c* is a MatcherContinuation.
   c. Let *d* be a new MatcherContinuation with parameters (*y*) that captures *x*, *c*, *direction*, and *parenIndex* and performs the following steps when called:
      i. Assert: *y* is a MatchState.
      ii. Let *cap* be a copy of *y*'s *captures* List.
      iii. Let *Input* be *x*'s *input*.
      iv. Let *xe* be *x*'s *endIndex*.
      v. Let *ye* be *y*'s *endIndex*.
      vi. If *direction* is forward, then
         1. Assert: *xe* ≤ *ye*.
         2. Let *r* be the CaptureRange (*xe*, *ye*).
      vii. Else,
         1. Assert: *direction* is backward.
         2. Assert: *ye* ≤ *xe*.
         3. Let *r* be the CaptureRange (*ye*, *xe*).
      viii. Set *cap*[*parenIndex* + 1] to *r*.
      ix. Let *z* be the MatchState (*Input*, *ye*, *cap*).
      x. Return *c*(*z*).
   d. Return *m*(*x*, *d*).

Solution 1: encoding failures — the error monad

Wrap the result of computations which can fail in the error monad [11].

```
Inductive Result (S F: Type) :=
| Success (s: S)
| Failure (f: F).

Definition bind {R S F: Type} (r: Result R F) (f: R → Result S F) :=
match r with
| Success s ⇒ f s
| Failure f ⇒ Failure f
end.
```

Solution 1: encoding failures — mechanizing

```
(*>> Assert: xe ⩽ ye. ... <<*)
if (xe ⩽ ye)
then ...
else Failure AssertionFailed

(*>> Let ch be the character Input[index]. ... <<*)
bind (List.nth_get input index) (fun ch ⇒ ...)

(*>> Set cap[parenIndex + 1] to r. ... <<*)
bind (List.nth_update cap (parenIndex + 1) r) (fun cap ⇒ ..)
```

Solution 1: encoding failures — notations

```
Notation "'let!' r '=<< ' y 'in' z" := (bind y (fun r => z)).

Notation "'assert!' b ';' z" := (if (negb b) then assertion_failed else z).

Notation "'destruct!' r '<- ' y 'in' z" := (match y with
| r => z
| _ => assertion_failed
end).
```

Solution 1: encoding failures — end result

```
(*>> Assert: xe ⩽ ye. ... <<*)
assert! (xe ⩽ ye) ; ...

(*>> Let ch be the character Input[index]. <<*)
let! ch =≪ List.nth_get input index in ...
(* or *)
let! ch =≪ input[index] in ...

(*>> Set cap[parenIndex + 1] to r. ... <<*)
let! cap: list _ =≪ List.nth_update cap (parenIndex + 1) r in ...
(* or *)
set cap[parenIndex + 1] := r in ...
```

Problem 2: unbounded recursion

General recursion is needed to implement quantifiers, e.g. $e^{\{\mathtt{min},\}}$

$\implies$ not structurally recursive.

Problem 2: unbounded recursion

General recursion is needed to implement quantifiers, e.g. $e^{\{\mathtt{min},\}}$

$\implies$ not structurally recursive.

```
Definition 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 min2 := if min = 0 then 0 else min - 1
    in
    RepeatMatcher m min2 y c
  in
  if min ≠ 0 then m x d
  else
  let z := m x d in
  if z is not mismatch then z
  else c x.
```

Solution 2: encoding (non-)termination — fuel

```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
    (c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    let d := fun (y: MatchState) ⇒
      ...
      RepeatMatcher m min2 y c fuel'
    in
    if min ≠ 0 then m x d
    else
    let z := m x d in
    if z is not mismatch then z
    else c x.
```

Solution 2: encoding (non-)termination — fuel

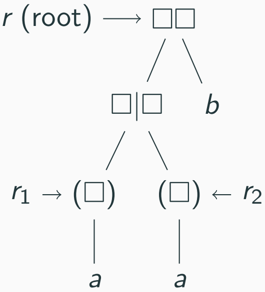
```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
    (c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    let d := fun (y: MatchState) ⇒
      ...
      RepeatMatcher m min2 y c fuel'
    in
    if min ≠ 0 then m x d
    else
    let z := m x d in
    if z is not mismatch then z
    else c x.
```

The (original) function terminates

$$\iff$$

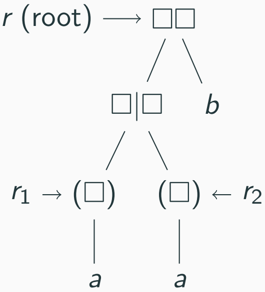
$\exists$ fuel, RepeatMatcher m min x c fuel $\neq$ Failure OutOfFuel.

Problem 3: missing arguments



How to compute the group index of `r2`?

Problem 3: missing arguments



How to compute the group index of `r2`?

`countLeftParenthesesBefore r2 = 1`

Use a zipper [5] to represent a regex and its context.

`RegexNode := (Regex * list RegexContext)`

Solution 3: encoding the context — zipper

Use a zipper [5] to represent a regex and its context.

```
RegexNode := (Regex * list RegexContext)
```

```
Fixpoint compileSubPattern (r: Regex) (ctx: list RegexContext) :=
    match r with
    | Disjunction r1 r2 ⇒
        let! m1 =<< compileSubPattern r1 (Disjunction_left r2 :: ctx) in
        ...
    end.
```

Proofs

Goals

We want to prove two things:

- The matching process always terminate;
- No operation ever fails during matching.

Mechanized invariant

The matcher invariant will be instrumental to proving both termination and the absence of failures.

```
Definition matcher_invariant (m: Matcher) :=
    (* For all valid state x and continuation c *)
    forall x c, Valid x →
        (* then either there is no match *)
        (m x c = mismatch) ∨
        (* or m produced a new valid state y which *)
        (exists y, Valid y ∧
            (* is a progress with respect to x *)
            x ≤ y ∧
            (* was passed to c to complete the match. *)
            m x c = c y).
```

where $\mathtt{x} \leqslant \mathtt{y} \iff \mathtt{EndIndex}\ \mathtt{x} \leqslant \mathtt{EndIndex}\ \mathtt{y}$.

Proving the invariant

```
Theorem compiledSubPattern_matcher_invariant:
    (* For all matcher m compiled from regex r *)
    forall m r, compileSubPattern r = Success m ->
        (* the matcher invariant holds. *)
        matcher_invariant m.
```

By induction on `r`.

Proving the invariant: RepeatMatcher

```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    let d := fun (y: MatchState) ⇒
      if min = 0 and endIndex(y) = endIndex(x)
      then mismatch
      else
        let min2 := if min = 0 then 0 else min - 1 in
        RepeatMatcher m min2 y c fuel'
    in
    if min ≠ 0 then m x d
    else
      let z := m x d in
      if z is not mismatch
      then z
      else c x.
```

Proving the invariant: RepeatMatcher

```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    let d := fun (y: MatchState) ⇒
      if min = 0 and endIndex(y) = endIndex(x)
      then mismatch
      else
        let min2 := if min = 0 then 0 else min - 1 in
        RepeatMatcher m min2 y c fuel'
    in
    if ??? then c x
    else m x d (* m satisfies the matcher invariant by IH *).
```

Proving the invariant: RepeatMatcher

```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    let d := fun (y: MatchState) ⇒
      if min = 0 and endIndex(y) = endIndex(x)
      then mismatch
      else
        let min2 := if min = 0 then 0 else min - 1 in
        RepeatMatcher m min2 y c fuel'
    in
    if ??? then c x
    else if ??? then mismatch
    else
      let y := ???
      (* y is valid *)
      (* x ⩽ y holds *)
      d y.
```

Proving the invariant: RepeatMatcher

```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    if ??? then c x
    else if ??? then mismatch
    else
      let y := ???
      (* y is valid *)
      (* x ⩽ y holds *)
      if min = 0 and endIndex(y) = endIndex(x)
      then mismatch
      else
        let min2 := if min = 0 then 0 else min - 1 in
        RepeatMatcher m min2 y c fuel'.
```

Proving the invariant: RepeatMatcher

```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    if ??? then c x
    else if ??? then mismatch
    else
      let y := ???
      (* y is valid *)
      (* x ⩽ y holds *)
      if min = 0 and endIndex(y) = endIndex(x)
      then mismatch
      else
        (* if min = 0, then we must have EndIndex x < EndIndex y *)
        let min2 := if min = 0 then 0 else min - 1 in
        RepeatMatcher m min2 y c fuel'.
```

Proving the invariant: RepeatMatcher

```
Fixpoint RepeatMatcher (m: Matcher) (min: nat) (x: MatchState)
(c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ Failure OutOfFuel
  | S fuel' ⇒
    if ??? then c x
    else if ??? then mismatch
    else
      let y := ???
      (* y is valid *)
      (* x ⩽ y holds *)
      if min = 0 and endIndex(y) = endIndex(x)
      then mismatch
      else
        (* if min = 0, then we must have EndIndex x < EndIndex y *)
        let min2 := if min = 0 then 0 else min - 1 in
        (* Not OutOfFuel if min + remainingChars(x) + 1 ⩽ fuel *)
        RepeatMatcher m min2 y c fuel'.
```

Proving termination (for free)

```
Theorem termination :
    (* For all matcher m satisfying the invariant *)
    forall m, matcher_invariant m →
    (* and valid state x *)
    forall x, Valid x →
    (* the matcher cannot run out of fuel. *)
    m x (fun z ⇒ Success z) ≠ Failure OutOfFuel.
```

Proving termination (for free)

```
Theorem termination :
    (* For all matcher m satisfying the invariant *)
    forall m, matcher_invariant m →
    (* and valid state x *)
    forall x, Valid x →
    (* the matcher cannot run out of fuel. *)
    m x (fun z ⇒ Success z) ≠ Failure OutOfFuel.
```

Direct if `m x (fun z ⇒ Success z) = mismatch ≠ Failure _`. Otherwise, by the matcher invariant, for some `y`

```
m x (fun z ⇒ Success z)  = (fun z ⇒ Success z) y
                         = Success y
                         ≠ Failure _
```

□

Generalizing the invariant: backward progress

Some regexes go through the string backward, e.g. lookbehinds.

Generalizing the invariant: backward progress

Some regexes go through the string backward, e.g. lookbehinds.

- Parametrize progress, the matcher invariant, etc. on the direction of the regex;

```
Definition matcher_invariant (m: Matcher) (dir: Direction) :=
  (* For all valid state x and continuation c *)
  forall x c, Valid x →
    (* then either there is no match *)
    (m x c = mismatch) ∨
    (* or m produced a new valid state y which *)
    (exists y, Valid y ∧
      (* is a progress with respect to x *)
      x ≤dir y ∧
      (* was passed to c to complete the match. *)
      m x c = c y).
```

Generalizing the invariant: backward progress

Some regexes go through the string backward, e.g. lookbehinds.

- Parametrize progress, the matcher invariant, etc. on the direction of the regex;

```
Definition matcher_invariant (m: Matcher) (dir: Direction) :=
  (* For all valid state x and continuation c *)
  forall x c, Valid x →
    (* then either there is no match *)
    (m x c = mismatch) ∨
    (* or m produced a new valid state y which *)
    (exists y, Valid y ∧
      (* is a progress with respect to x *)
      x ≤_dir y ∧
      (* was passed to c to complete the match. *)
      m x c = c y).
```

- Going backward or forward is irrelevant: monotony is the key!

Conclusion

Extracting and executing

- Extracted to OCaml;

Extracting and executing

- Extracted to OCaml;
- Tested;

```
let%expect_test "sequence" =
  test_regex
    ((char 'a') -- (char 'b') -- (char 'b'))
    "abbb"
    0 ();
  [%expect {|
    Matched 3 characters ([0-3]) in 'abbb' (length=4)
  |} ]
```

Extracting and executing

- Extracted to OCaml;
- Tested;
- Cross-validated with V8 using a differential fuzzer implemented by Aurèle.

Extending the specification

The specification is still evolving: our mechanization will have to do the same.

As a matter of facts, the latest draft[1]:

[1] `https://tc39.es/ecma262/`

Extending the specification

The specification is still evolving: our mechanization will have to do the same.

As a matter of facts, the latest draft[1]:

- Does some refactoring.

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*).

[1] https://tc39.es/ecma262/

Extending the specification

The specification is still evolving: our mechanization will have to do the same.

As a matter of facts, the latest draft[1]:

- Does some refactoring.

  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 MatchTwoAlternatives(*m1*, *m2*).

[1] `https://tc39.es/ecma262/`

Extending the specification

The specification is still evolving: our mechanization will have to do the same.

As a matter of facts, the latest draft[1]:

- Does some refactoring.
- Introduces some additional support for unicode (`v` flag).

[1] `https://tc39.es/ecma262/`

Short-term future work

In the near future, we would like to take a look at:

Short-term future work

In the near future, we would like to take a look at:

- Unicode support;

Short-term future work

In the near future, we would like to take a look at:

- Unicode support;
- Integrating with test262;

Short-term future work

In the near future, we would like to take a look at:

- Unicode support;
- Integrating with test262;
- Additional proofs about the semantics:

$$
\begin{aligned}
e^{??} &\not\equiv \varepsilon \mid e \\
(?{:}\ e^{*})^{*} &\equiv e^{*} \\
e^{*} &\equiv \varepsilon \text{ where } e \text{ only ever matches the empty string.}
\end{aligned}
$$

Long-term future work

- Improve the extraction to get a more efficient engine.
- Develop a tool to check our comments against the specification.
- Prove correct an efficient engine for ECMAScript regexes.
- Prove equivalent some alternative semantics more suited for formal reasoning.

References

[1] Martin Bodin et al. **“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*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 87–100. DOI: `10.1145/2535838.2535876`.

[2] Nariyoshi Chida and Tachio Terauchi. **“Repairing DoS Vulnerability of Real-World Regexes”.** In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 2060–2077. DOI: `10.1109/SP46214.2022.9833597`.

[3] Romain Edelmann. **“Efficient Parsing with Derivatives and Zippers”.** In: (2021), p. 246. DOI: `https://doi.org/10.5075/epfl-thesis-7357`. URL: `http://infoscience.epfl.ch/record/287059`.

[4] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. **"Automata theory, languages, and computation".** In: *International Edition* 24.2 (2006), pp. 171–183.

[5] Gérard Huet. **"The zipper".** In: *Journal of functional programming* 7.5 (1997), pp. 549–554.

[6] SC Kleene. **"Representation of events in nerve nets and finite automata".** In: *Automata Studies: Annals of Mathematics Studies. Number 34* 34 (1956), p. 3.

[7] Blake Loring, Duncan Mitchell, and Johannes Kinder. **"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*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 425–438. DOI: 10.1145/3314221.3314645.

[8] Sergio Maffeis, John C. Mitchell, and Ankur Taly. **“An Operational Semantics for JavaScript”.** In: *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*. Ed. by G. Ramalingam. Vol. 5356. Lecture Notes in Computer Science. Springer, 2008, pp. 307–325. DOI: `10.1007/978-3-540-89330-1_22`.

[9] Daejun Park, Andrei Stefanescu, and Grigore Rosu. **“KJS: a complete formal semantics of JavaScript”.** In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 346–356. DOI: `10.1145/2737924.2737991`.

[10] J. G. Politz et al. ***Mechanized LambdaJS.*** https://blog.brownplt.org/2012/06/04/lambdajs-coq.html. 2012.

[11] Philip Wadler. **“Monads for functional programming”.** In: *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*. Springer. 1995, pp. 24–52.

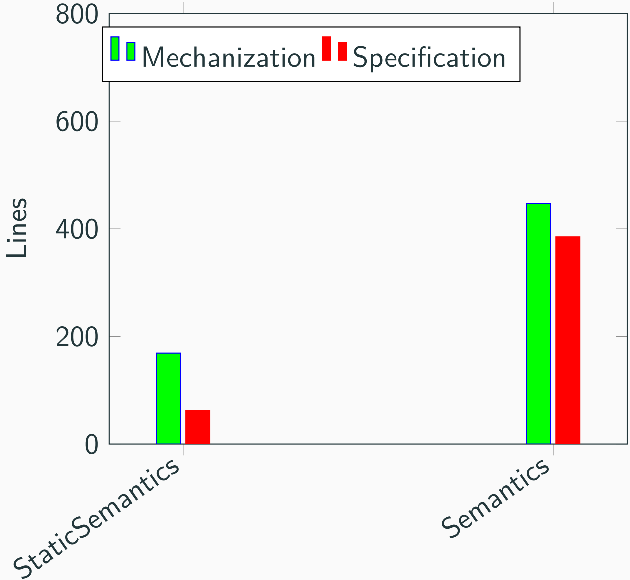
Appendix

Getting rid of mutability

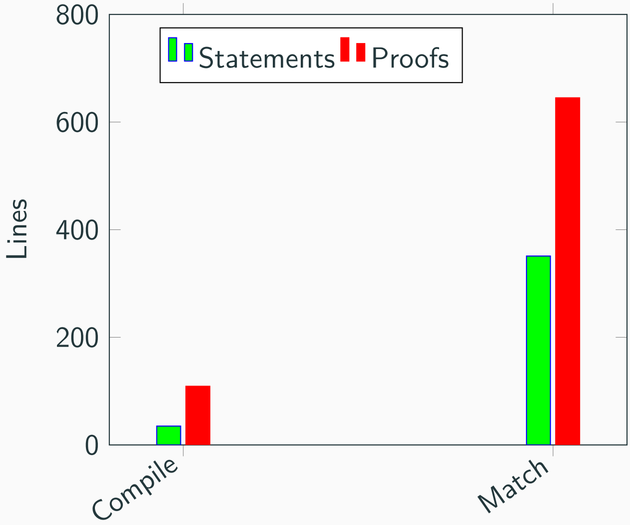
ii. Let *cap* be a copy of *y*'s *captures* List.
iii. Let *Input* be *x*'s *input*.
iv. Let *xe* be *x*'s *endIndex*.
v. Let *ye* be *y*'s *endIndex*.
vi. If *direction* is forward, then
  1. Assert: $xe \leq ye$.
  2. Let *r* be the CaptureRange (*xe*, *ye*).
vii. Else,
  1. Assert: *direction* is backward.
  2. Assert: $ye \leq xe$.
  3. Let *r* be the CaptureRange (*ye*, *xe*).
viii. Set *cap*[*parenIndex* + 1] to *r*.
ix. Let *z* be the MatchState (*Input*, *ye*, *cap*).
x. Return *c*(*z*).

3. $\boxed{\text{Let } cap \text{ be a copy of } x\text{'s } captures \text{ List.}}$
4. For each integer *k* in the inclusive interval from *parenIndex* + 1 to *parenIndex* + *parenCount*, $\boxed{\text{set } cap[k] \text{ to } \textbf{undefined}.}$
5. Let *Input* be *x*'s *input*.
6. Let *e* be *x*'s *endIndex*.
7. Let *xr* be the MatchState (*Input*, *e*, $\boxed{cap)}$.
8. If *min* ≠ 0, return *m*(*xr*, *d*).
9. If *greedy* is **false**, then
   a. Let *z* be *c*(*x*).
   b. If *z* is not failure, return *z*.
   c. Return *m*(*xr*, *d*).
10. Let *z* be *m*(*xr*, *d*).
11. If *z* is not failure, return *z*.
12. Return *c*(*x*).

Statistics about the mechanization



Statistics about the proofs



Raw statistics

spec	proof	comments	
35	109	1	props/Compile.v
11	39	2	props/Definitions.v
133	207	0	props/EarlyErrors.v
7	22	28	props/Incorrectness.v
351	645	122	props/Match.v
81	0	3	spec/base/Characters.v
24	0	5	spec/base/Coercions.v
26	0	2	spec/base/Errors.v
68	22	8	spec/base/Numeric.v
33	0	3	spec/Base.v
27	10	0	spec/ClutterFree.v
109	356	236	spec/Frontend.v
59	0	34	spec/Notation.v
225	183	45	spec/Patterns.v
447	0	385	spec/Semantics.v
169	0	62	spec/StaticSemantics.v
205	0	5	tactics/Focus.v
14	0	0	tactics/Specialize.v
142	28	2	tactics/Tactics.v
247	365	5	utils/List.v
6	0	0	utils/Option.v
61	3	1	utils/Result.v
2480	1989	949	total