Lean Formalization of Extended Regular Expression Matching with Lookarounds

Ekaterina Zhuchko*
Tallinn University of Technology
Estonia
ekzhuc@taltech.ee

Margus Veanes
Microsoft Research
USA
margus@microsoft.com

Gabriel Ebner
Microsoft Research
USA
gabrielebner@microsoft.com

Abstract

We present a formalization of a matching algorithm for extended regular expression matching based on locations and symbolic derivatives which supports intersection, complement and lookarounds and whose implementation mirrors an extension of the recent .NET NonBacktracking regular expression engine. The formalization of the algorithm and its semantics uses the Lean 4 proof assistant. The proof of its correctness is with respect to standard matching semantics.

CCS Concepts: • **Theory of computation → Regular languages**; • **Computing methodologies → Boolean algebra algorithms**.

Keywords: regex, derivative, POSIX

ACM Reference Format:
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 '24), January 15–16, 2024, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3636501.3636959

1 Introduction

Regular expressions play a central role in many practical security critical contexts often as a core subset of a domain specific language for detection, search, filtering, and lexing. There are many such applications, including in data security for *credential scanning*, in network *intrusion detection*, in web security to prevent cross-site scripting attacks as rules of *traffic filters* in most firewalls, in *malware scanning*, in email *spam filters*, etc. Regular expressions are also supported in some SMT solvers as an integrated part of their *sequence theory* algorithms, and SMT solvers are widely used by other verification tools.

When regular expressions are deployed in this manner, essentially as an executable *specification* language, the *trust in correctness* of their supporting theory and of the underlying algorithms becomes critical, perhaps even more so than *perfomance*. *Formal* correctness and analysis of *derivative* based theory and algorithms of regular expressions, using *proof assistants*, is a fairly recent and active research area including work in analysis of *lexing* algorithms [3, 8, 24, 26]. One explanation is that proof assistants have matured and become easier to work with, and derivatives admit an algebraic and inductive view of regular expression semantics, that lends itself to formal analysis in a way that direct handling of finite automata is less suited for.

A new *location* based theory of derivatives of regular expressions has recently been developed and implemented as a new backend in the .NET regular expression framework [17], where also *anchors* are supported and *backtracking* (PCRE) semantics is maintained. This work has subsequently been extended in [27] to also support *lookarounds* as well as *intersection* and *complement* while switching the semantics from PCRE to POSIX because the current understanding of what intersection and complement would mean in PCRE is unspecified. Since both the theory as well as the matching algorithm are new, having formal confidence by a proof assistant is very valuable for any further adaptation or development of the theory and optimizations in related algoritms.

In this paper we target the work in [27] and formalize the core aspects of it in Lean. This work also provides a platform for analysis of [17] because the different variations of matching semantics can also be represented faithfully with derivatives and encoded as different rules in Lean, in particular, by omitting *commutativity* of alternation and selecting the standard subset of regular expressions with *lookarounds*. This work has important practical applications as it builds confidence in an existing industrial engine in .NET with a variant that extends it with Boolean operators so far not available in other engines, with much increased expressivity, allowing to use regular expressions as a *full-blown logic*, which is particularly relevant for applications such as CredScan [13].

The main lessons learnt from the formalization are that choosing the right representation and definitions can give a lot of insight into the problem that we are defining. For example, defining and working with locations as zippers

*Work done during Microsoft Research internship.

CPP '24, January 15–16, 2024, London, UK

ACM ISBN 979-8-4007-0488-8/24/01
https://doi.org/10.1145/3636501.3636959

and spans allowed a direct and intuitive definition of the universe of the match semantics. Moreover, we present the first formalization of an Effective Boolean Algebra which can represent an interface to SMT solvers. Another insight is the choice of termination metrics needed in proofs, which filled some of the gaps in induction arguments in [27]. The complete Lean formalization of the work presented here is available in [28].

Outline of the paper. We first introduce basic terminology and notation in Section 2 where we also point out the close relation between *locations* and *list zippers* [11]. In the following sections we develop the formal theory while also providing the "informal" definitions in parallel with the formal ones in Lean. In Section 3 we formally define the class **RE** that includes lookarounds and all Boolean operators and we define their semantics in terms of *spans* that are at some level related to the value type `System.Span<T>` for representing contiguous memory regions in .NET [25]. In Sections 4 and 5 we formally develop the theory of derivatives, prove their correctness with respect to their match semantics, and prove the correctness of the match algorithm **llmatch** introduced in [27] that respects POSIX semantics. In Section 6 we discuss additional rewrites that can be used to simplify derivatives and we also prove a theorem that shows that negative lookarounds can be eliminated, which has potential practical applications for further optimizations that we briefly mention. Section 7 is about related work and Section 8 mentions future work and open problems. Section 9 concludes the paper.

2 Preliminaries

The formalization makes use of concepts like zippers that are not traditionally used in formal languages. To make it easier to follow along, we will provide an informal explanation of the arguments involved. In this section, we give an overview of the basic concepts and how they are represented in Lean. We write $lhs \stackrel{\text{DEF}}{=} rhs$ to let *lhs* be *equal by definition* to *rhs*. Let `Bool` $= \{\textbf{false}, \textbf{true}\}$ denote Boolean values. We write $\langle x_1, \ldots, x_n \rangle$ for tuples of elements for $n \geq 2$ and let $\langle x_1, \ldots, x_n \rangle.i \stackrel{\text{DEF}}{=} x_i$ for $1 \leq i \leq n$.

Words. Let σ be an *alphabet* type, σ may denote an infinite set. *Words* or *strings* over σ are represented by lists of elements of type σ, that is denoted by σ^* and in Lean represented by the type `List` σ. We let ϵ or [] denote the empty word. The length of a word w is denoted by $|w|$. For $0 \leq i < |w|$ let $w_i \in \sigma$ denote the i'th element of w. We denote the *reverse* of w by w^{r}, so that $w^{\mathrm{r}}{}_i = w_{|w|-1-i}$ for $0 \leq i < |w|$. Concatenation of $u \in \sigma^*$ with $v \in \sigma^*$ is formally denoted by $u +\!\!+ v$. We also write $a :: v$ for prepending $a \in \sigma$ to $v \in \sigma^*$ (the cons operation on lists). Informally, we write the juxtaposition uv and av in both cases. The subword of w from position i where $0 \leq i \leq |w|$ of length n where $i + n \leq |w|$ is denoted by $w_{i,n}$, e.g., $\epsilon_{0,0} = \epsilon$.

Locations. A *location* describes a position in a string. We represent a location as pair of words in $\textbf{Loc} \stackrel{\text{DEF}}{=} \sigma^* \times \sigma^*$. A location $\langle \epsilon, u \rangle$ is *initial* and a location $\langle u, \epsilon \rangle$ is *final*. The *reverse* of a location $\langle u, v \rangle$ is the location $\langle u, v \rangle^{\mathrm{r}} \stackrel{\text{DEF}}{=} \langle v, u \rangle$. Let $\langle u, a :: v \rangle + 1 \stackrel{\text{DEF}}{=} \langle a :: u, v \rangle$ define the *next* location from any given nonfinal location and let $\langle u, \epsilon \rangle + 1 \stackrel{\text{DEF}}{=} \langle u, \epsilon \rangle$. Let also $x+n$ be defined analogously for all $x \in \textbf{Loc}$ and $n \geq 0$ where we let $x + 0 \stackrel{\text{DEF}}{=} x$.

Let $w \in \sigma^*$. A *location in* w is a location $\langle u^{\mathrm{r}}, v \rangle$ such that $w = u +\!\!+ v$. Informally, we write $w[i]$ for the location with $|u| = i$, where i, $0 \leq i \leq |w|$, is called the *index* or *position* of the location in w.

A location is a variant of a list *zipper* [11] for $w \in \sigma^*$ that facilitates traversing w forwards and backwards relative to an index which is why the first segment u^{r} above is in reverse.The representation is also convenient in Lean where the underlying type of words is `List` σ and this representation aligns well with standard list operations of directly accessing the head and the tail of a list and simplifying the main operations over locations (and the reversed locations).

In Lean, we represent locations by the following product type and refer to the left projection as `Loc.left` and the right projection as `Loc.right`.

```
def Loc : List σ × List σ
```

Informally, we consider the set of all *nonfinal locations* $\textbf{Loc}^+ \stackrel{\text{DEF}}{=} \sigma^* \times \sigma^+$. A *nonfinal* location $\langle u^{\mathrm{r}}, a::v \rangle$ in $w = uav$ corresponds to the zipper $\langle u^{\mathrm{r}}, a, v \rangle$, but a location in w can also be *final* $\langle w^{\mathrm{r}}, \epsilon \rangle$. Locations in a word w of length $n = |w|$ are illustrated in Figure 1. For example, ϵ has only one location $\epsilon[0] = \langle \epsilon, \epsilon \rangle$ that is both initial and final. Observe that the reverse of a location $w[i]$ in w is the location $w^{\mathrm{r}}[|w|-i]$ in w^{r}. For example, the reverse of the final location in w is the initial location in w^{r}.

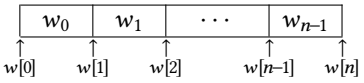

Figure 1. Locations in a word w of length n.

For finite nonempty sets X of locations in some word w, we informally let $\max(X)$ ($\min(X)$) denote the maximum (minimum) location in the set according to the location order defined by $w[i] \leq w[j] \stackrel{\text{DEF}}{=} i \leq j$.

Effective Boolean Algebras. An *Effective Boolean Algebra (EBA) over* σ is a tuple $\mathcal{A} = (\sigma, \alpha, [\![\_]\!], \bot, \top, \sqcup, \sqcap, ^{\mathrm{c}})$ where α is a set of *predicates* such that $\bot, \top \in \alpha$ and α is closed under the Boolean connectives [5]. The function $[\![\_]\!] : \alpha \to 2^\sigma$ is the *denotation* satisfying $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \sigma$, and for all $\varphi, \psi \in \alpha$, $[\![\varphi \sqcup \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \sqcap \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and

$[\![\varphi^c]\!] = \sigma \setminus [\![\varphi]\!]$. In Lean, we model EBAs as a type class on the type α. The element type σ is marked as an out-param, which means that σ is automatically inferred from α.

```
class Denotation (α : Type u) (σ : outParam (Type v)) where
  denote : α → σ → Bool

class EffectiveBooleanAlgebra (α : Type u)
    (σ : outParam (Type v)) extends Denotation α σ,
      Bot α, Top α, Inf α, Sup α, HasCompl α where
  denote_bot : denote ⊥ c = false
  denote_top : denote ⊤ c = true
  denote_compl : denote ϕᶜ c = !denote ϕ c
  denote_inf : denote (ϕ⊓ψ) c = (denote ϕ c && denote ψ c)
  denote_sup : denote (ϕ⊔ψ) c = (denote ϕ c || denote ψ c)
```

For $\varphi \in \alpha$ and $c \in \sigma$ the *membership* relation (`denote` φ c) returns **true** iff $c \in [\![\varphi]\!]$ and is therefore decidable. We have not formalized *satisfiability* of $\varphi \in \alpha$ in Lean here, i.e., deciding $[\![\varphi]\!] \neq \emptyset$, because we do not use it here. Additionally, for every type β with a denotation function $[\![\_]\!] : \beta \to 2^\sigma$ we define a type `BA` β of Boolean combinations over β, which is naturally an EBA.

The EBA abstraction implies that the formalization presented here is generic with respect to any EBA $\mathcal{A}$, although examples will use $\mathcal{A}$ to represent character classes over the Plane 0 subset of Unicode, which is also of primary interest here because the character type `char` in .NET is based on `Uint16`. More generally, $\mathcal{A}$ can represent an interface to an SMT solver where σ is the infinite universe of all models that the solver can produce, α is the set of all well-formed formulas of the solver, and $[\![\varphi]\!]$ is an effective enumeration of all models that satisfy the formula φ. The only operation needed here is, given $c \in \sigma$ and $\varphi \in \alpha$, to decide if $c \in [\![\varphi]\!]$, that is typically also denoted by $c \vDash \varphi$ in $\mathcal{A}$.

Character classes. In all the examples below we let σ stand for the standard 16-bit character set of Unicode[1] and use the .NET syntax [14] of regular expression character classes. For example, `[A-Z]` stands for all the Latin capital letters, `[0-9]` for all the *Latin numerals*, `\d` for all the *decimal digits*, `\w` for all the *word-letters* and `.` for all characters besides the *newline character* `\n`. Any individual character (as a character class) denotes only that character.

In regular expression examples we use character classes directly as predicates. For example, $\texttt{0} \in [\![\texttt{\textbackslash d}]\!]$ and $\texttt{0} \in [\![\texttt{\textbackslash w}]\!]$ and so $\texttt{0} \notin [\![\texttt{\textbackslash w} \sqcap \texttt{\textbackslash d}^c]\!]$. More accurately, a predicate corresponding to a character class C is represented by some predicate ψ_C in α. So, in Lean (`denote` $\psi_{\texttt{[A-Z]}}$ c) returns **true** iff c is a Latin capital letter, and $\psi_{\texttt{[\textbackslash w-[\textbackslash d]]}}$ is equivalent to $\psi_{\texttt{\textbackslash w}} \sqcap \psi^c_{\texttt{\textbackslash d}}$, i.e., both predicates denote the same elements.

In Lean, we can construct a predicate from a string (denoting the union of all the characters in the string)[2] with the following function:

```
def String.characterClass (s : String) : BA Char :=
 s.toList |>.map .atom |>.foldr .or .bot
```

For example the string "`0123456789`" is converted into a predicate that is equivalent to $\psi_{\texttt{[0-9]}}$. Here the type `BA Char` corresponds to α and `Char` corresponds to σ above.

3 Regexes with Lookarounds

Here we formally define extended regular expressions with *lookarounds* modulo an EBA $\mathcal{A} = (\sigma, \alpha, [\![\_]\!], \bot, \top, \sqcup, \sqcap, ^c)$. In examples we use standard (.NET Regex) character classes, while the actual representation of predicates is irrelevant and σ may even be *infinite*.

3.1 Regexes

The class **RE** of regular expressions, or *regexes* for short, is here defined by the following abstract grammar. Let $\psi \in \alpha$ and R, R' range over **RE**.

$$R \quad ::= \quad \psi \mid \varepsilon \mid R \cup R' \mid R \cap R' \mid R \cdot R' \mid R* \mid \sim R \mid \ell$$

$$\ell \quad ::= \quad (?{=}R) \mid (?{<}{=}R) \mid (?!R) \mid (?{<}!R)$$

where the operators in the first row appear in order of precedence, with *alternation* $\cup$ having lowest and *complement* $\sim$ having highest precedence. The remaining operators in the first row are the *empty-word regex* ε also denoted by $()$, *intersection* $\cap$, *concatenation* $\cdot$, and *Kleene star* $*$. We use the standard abbreviation $R+$ for $R{\cdot}R*$. Concatenation is often implicit by using juxtaposition when this is unambiguous.

The regex matching *nothing* is $\bot \in \alpha$.

The expressions ℓ are called *lookarounds*: $(?{=}R)$ is *lookahead*, $(?{<}{=}R)$ is *lookbehind*, $(?!R)$ is *negative lookahead*, and $(?{<}!R)$ is *negative lookbehind*, where R is the *body* of the lookaround, also denoted by $body(\ell)$. We let **LA** denote the set of all lookarounds.

While *anchors* are not explicitly included in the core definition of **RE**, they can be defined in terms of lookarounds as illustrated in Section 6.2 for the start anchor and end anchor. More generally, all standard anchors can be defined using lookarounds, see [27, Table 1].

In Lean, we represent the syntax of regular expressions as the following inductive datatype `RE`. The parameter α here refers to the EBA, theorems about regular expression then take an `[EffectiveBooleanAlgebra` α σ`]` typeclass argument.

```
inductive RE (α : Type) : Type where
| ε               : RE α
| Pred (e : α)    : RE α
| Alternation       : RE → RE → RE
| Intersection      : RE → RE → RE
| Concatenation     : RE → RE → RE
| Star              : RE → RE
| Negation          : RE → RE
| Lookahead         : RE → RE
| Lookbehind        : RE → RE
| NegLookahead      : RE → RE
| NegLookbehind     : RE → RE
```

[1] Also known as *Plane 0* or the *Basic Multilingual Plane* of Unicode.
[2] The string is not being "parsed" in any way as a character class of .NET.

In order to present the semantics of the Kleene star operator in Section 3.2, we define the *n-times repetition of R* for $n \geq 0$, $R^{(n)}$, as $R^{(0)} \stackrel{\text{DEF}}{=} \varepsilon$ and $R^{(n+1)} \stackrel{\text{DEF}}{=} R \cdot R^{(n)}$. In Lean, we define the operation inductively on the number of repetitions:

```
def repeat_cat (R : RE α) (n : ℕ) : RE α :=
 match n with
 | 0          => ε
 | Nat.succ n => R · (repeat_cat R n)
notation f "(" n ")" => repeat_cat f n
```

The empty-word regex is used as base case and the inductive case essentially represents the expansion law for the Kleene star operator, with concatenation on the left. The `notation` command allows us to define arbitrary mixfix syntax `r ( n )`.

It is crucial in the formalization to ensure that all procedures terminate, which can be done in Lean using the `termination_by` pragma. For this reason, when the inductive step is not trivially terminating, we need to provide for each theorem a custom metric that can be used to ensure that the recursion is indeed well-founded.

The size of a regex R, $|R|$, is defined recursively as the number of operators of R, where $|\psi| = |\varepsilon| \stackrel{\text{DEF}}{=} 0$ and for all unary operators $\blacklozenge$ and binary operators $\diamond$, $|\blacklozenge R| \stackrel{\text{DEF}}{=} 1 + |R|$ and $|L \diamond R| \stackrel{\text{DEF}}{=} 1 + |L| + |R|$. For example, $|(?=\sim(\varepsilon \cdot \bot))| = 3$.

Other crucial examples of metrics used in the theorems are the star height and lookaround height of R, denoted by $\mathit{starHeight}(R)$ and $\mathit{lookHeight}(H)$, respectively. Each metric captures the nesting depth of the star and lookarounds of the regex, and takes the maximum of the nesting depths in the case of binary operators. For example,

$\mathit{starHeight}(R*) \stackrel{\text{DEF}}{=} \mathit{starHeight}(R) + 1$,
$\mathit{starHeight}(L \diamond R) \stackrel{\text{DEF}}{=} \max(\mathit{starHeight}(L), \mathit{starHeight}(R))$,
$\mathit{starHeight}(\ell) \stackrel{\text{DEF}}{=} \mathit{starHeight}(\mathit{body}(\ell))$,

and lookaround height is defined analogously by increasing the nesting only in $\mathit{lookHeight}(\ell) \stackrel{\text{DEF}}{=} \mathit{lookHeight}(\mathit{body}(\ell)) + 1$, for example, $\mathit{lookHeight}((?=\sim(\varepsilon \cdot \top*) \cdot (?!\top))) = 2$.

In Lean, we let `lookaround_height` and `star_height` be defined analogously, both of which will be used as termination metrics later in the development.

3.2 Match Semantics

In this section we introduce the *match semantics* of regexes, which will serve as the specification to prove the correctness of our derivative-based approach. The *match semantics* corresponds essentially to the classical semantics of regexes based on languages, but using spans and locations in order to reason about the matching sections of a word. A *span* is defined as $\textbf{Span} \stackrel{\text{DEF}}{=} \sigma^* \times \sigma^* \times \sigma^*$, i.e., a span is a triple of words that we denote by $\langle s, u, v \rangle$ where $s, u, v \in \sigma^*$. Let $w \in \sigma^*$, a *span in w* is a span $sp = \langle s^{\mathrm{r}}, u, v \rangle$ such that $w = suv$, where, similarly to locations, $sp.1$ is the reverse of s.

In Lean, we represent spans by the nested pair of the following type:

```
def Span : List σ × (List σ × List σ)
```

For a span `sp := (s, u, v)`, we refer to `s` as `sp.left`, `u` as `sp.match` and `v` as `sp.right`. We also define the operation to convert a span into a location view.

```
def Span.loc (sp : Span σ) : List σ × List σ :=
⟨sp.left, sp.match ++ sp.right⟩
```

Let $sp \in \textbf{Span}$. The *reverse* of sp is $sp^{\mathrm{r}} \stackrel{\text{DEF}}{=} \langle sp.3, sp.2^{\mathrm{r}}, sp.1 \rangle$. Thus $|sp.2| = |sp^{\mathrm{r}}.2|$. The start location *loc*, the *end* location, the substring *match*, and the *string* of sp are:

$$\begin{aligned} \mathit{loc}(sp) &\stackrel{\text{DEF}}{=} \langle sp.1, sp.2 +\!\!+ sp.3 \rangle \\ \mathit{end}(sp) &\stackrel{\text{DEF}}{=} \langle sp.2^{\mathrm{r}} +\!\!+ sp.1, sp.3 \rangle \\ \mathit{match}(sp) &\stackrel{\text{DEF}}{=} sp.2 \\ \mathit{string}(sp) &\stackrel{\text{DEF}}{=} sp.1^{\mathrm{r}} +\!\!+ sp.2 +\!\!+ sp.3 \end{aligned}$$

Note that $\mathit{end}(sp) = \mathit{loc}(sp^{\mathrm{r}})^{\mathrm{r}}$. For all $R \in \textbf{RE}$ and $sp \in \textbf{Span}$, the *match semantics* $sp \models R$ is defined by induction over the regexes, as follows. Note in particular that spans and locations are used to capture the context conditions of the semantics of lookarounds.

$$\begin{aligned} sp &\models \varepsilon && \stackrel{\text{DEF}}{=} |sp.2| = 0 \\ sp &\models \phi && \stackrel{\text{DEF}}{=} |sp.2| = 1 \wedge (sp.2)_0 \in [\![\phi]\!] \\ sp &\models L \cdot R && \stackrel{\text{DEF}}{=} \exists s_1, s_2 : \mathit{loc}(sp) = \mathit{loc}(s_1) \wedge s_1 \models L \wedge \\ & && \qquad \mathit{end}(sp) = \mathit{end}(s_2) \wedge s_2 \models R \\ sp &\models L \cup R && \stackrel{\text{DEF}}{=} sp \models L \vee sp \models R \\ sp &\models L \cap R && \stackrel{\text{DEF}}{=} sp \models L \wedge sp \models R \\ sp &\models \sim R && \stackrel{\text{DEF}}{=} sp \not\models R \\ sp &\models R* && \stackrel{\text{DEF}}{=} \exists n : sp \models R^{(n)} \\ sp &\models (?=R) && \stackrel{\text{DEF}}{=} |sp.2|{=}0 \wedge \exists s : \mathit{loc}(s){=}\mathit{loc}(sp) \wedge s \models R \\ sp &\models (?!R) && \stackrel{\text{DEF}}{=} |sp.2|{=}0 \wedge \nexists s : \mathit{loc}(s){=}\mathit{loc}(sp) \wedge s \models R \\ sp &\models (?<=R) && \stackrel{\text{DEF}}{=} |sp.2|{=}0 \wedge \exists s : \mathit{end}(s){=}\mathit{end}(sp) \wedge s \models R \\ sp &\models (?<!R) && \stackrel{\text{DEF}}{=} |sp.2|{=}0 \wedge \nexists s : \mathit{end}(s){=}\mathit{end}(sp) \wedge s \models R \end{aligned}$$

Any span $sp = \langle s^{\mathrm{r}}, u, v \rangle$ in w also has the more intuitive informal representation $w[i,n] \stackrel{\text{DEF}}{=} w[i{:}j] \stackrel{\text{DEF}}{=} sp$, as illustrated in Figure 2, where $i = |s|$, $n = |u|$ and $j = i + n$.

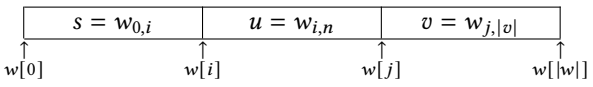

Figure 2. Span $w[i{:}j]$ in a word w.

Example 3.1. Consider a regex $R = R_1 \cap R_2 \cap R_3 \cap R_4$ with $R_1 =$ `(.*[A-Z].*)`, $R_2 =$ `(.*\d.*\d.*)`, $R_3 =$ `(.*[a-z].*)`, and $R_4 =$ `((?<=\W).*(?=\W))`. R is intuitively a specialized "password search pattern" matching a substring containing at least one Latin capital letter, at least two digits, at least one Latin noncapital letter, and is surrounded by non-word-letters. Let $w =$ "`0B:1aD2;e`" be the input string. Then $w[3{,}4] \models R$. Note that, $w[3{,}4] = \langle$"`:B0`", "`1aD2`", "`;e`"$\rangle$ where $\mathit{match}(w[3{,}4]) = w_{3,4} =$ "`1aD2`". ⊠

In Lean, we mirror the *match semantics* by defining the *match semantics function* `models`, which we represent also

```
def models (sp : Span σ) (R : RE α) : Prop :=
 match R with
 | ε      => sp.match.length = 0
 | Pred φ => sp.match.length = 1
             ∧ sp.match_head?.any (denote φ)
 | l · r  =>
   ∃ u₁ u₂, models ⟨sp.left, u₁, u₂ ++ sp.right⟩ l
            ∧ models ⟨u₁ʳ ++ sp.left, u₂, sp.right⟩ r
            ∧ u₁ ++ u₂ = sp.match
 | l ⋓ r  => models sp l ∨ models sp r
 | l ⋒ r  => models sp l ∧ models sp r
 | ~ r    => ¬ models sp r
 | r *    => ∃ (m : ℕ), models sp r ^ m
 | ?= r   =>
   sp.match.length = 0
   ∧ ∃ (s : Span σ), models s r ∧ s.loc = sp.loc
 | ?! r   =>
   sp.match.length = 0
   ∧ ¬ (∃ (s : Span σ), models s r ∧ s.loc = sp.loc)
 | ?<= r  =>
   sp.match.length = 0
   ∧ ∃ (s : Span σ), models sʳ r ∧ s.loc = spʳ.loc
 | ?<! r  =>
   sp.match.length = 0
   ∧ ¬ (∃ (s : Span σ), models sʳ r ∧ s.loc = spʳ.loc)
termination_by models sp R => star_metric R
infix:30 " ⊨ " => models
```

Figure 3. Formal definition of match semantics in Lean.

by the infix operator ⊨ as above. See Figure 3. Note that $end(sp)^{\mathbf{r}} = loc(sp^{\mathbf{r}})$, so in the Lean formalization explicit definition of $end(sp)$ is not needed.

Note that, although defined as function, this effectively represents a *matching relation*. The main reason why the `models` relation ⊨ is not defined as an inductively-defined predicate is due to the negative occurrence of the relation that would arise when defining the `~ r` constructor. Another possible strategy would have been to define the relation using a positive normal form i.e. for each constructor, we state what it means to be a match and what it means *not* to be a match. However, it would be particularly cumbersome to have to introduce all dual definitions and would result in a considerable number of (mainly artificial) operators. At the end, we decided to simply adopt a function constructing a **`Prop`** in order to adhere to the positivity requirement in Lean. Inductively-defined predicates have no termination requirement with respect to the regular expression, and the star case could have been effectively represented with its expansion law $R* = \varepsilon \cup R \cdot R*$. However, due to our definition as a function, we had to come up with an alternative encoding to represent the star case in order to accomodate the termination requirement. We instead define the semantics of the star case with the `repeat_cat` function by stating that there exists a (finite) number of repetitions of the regex which matches.

In order to prove termination of the `models` function, we need a custom metric `star_size_metric` which is effectively defined as the lexicographic order by jointly using the star height and size of a regex metrics. The star height is only needed to make the star constructor case terminate, while the other metric is used in all other cases. In particular, it is crucial for our definition that $starHeight(R^{(n)}) < starHeight(R*)$.

Note that semantics of lookbehind can be stated either in terms of regex reversal or span reversal; for example this is a valid condition for the positive lookbehind case (where regex reversal is defined below):

```
∃ (s : Span σ), models s (rʳ) ∧ s.loc = spʳ.loc
```

However, we decided to use the reversal on spans instead, as using the reversal operation would implicitly rely on the correctness of the more complicated regex reversal. Use of span reversal, as a one-line definition, moreover reflects accurately of what lookbehind means intuitively.

3.3 Reversal

Reversal of regexes plays a key role in derivatives of lookbehinds as well as in the top-level match algorithm. The *reverse* $R^{\mathbf{r}}$ of $R \in \mathbf{RE}$ is defined as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
\psi^{\mathbf{r}} & \overset{\text{DEF}}{=} & \psi & (?{=}R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & (?{<}{=}R^{\mathbf{r}}) \\
\varepsilon^{\mathbf{r}} & \overset{\text{DEF}}{=} & \varepsilon & (?{<}{=}R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & (?{=}R^{\mathbf{r}}) \\
(L \cdot R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & R^{\mathbf{r}} \cdot L^{\mathbf{r}} & (?!R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & (?{<}!R^{\mathbf{r}}) \\
(R*)^{\mathbf{r}} & \overset{\text{DEF}}{=} & (R^{\mathbf{r}})* & (?{<}!R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & (?!R^{\mathbf{r}}) \\
(L \cup R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & L^{\mathbf{r}} \cup R^{\mathbf{r}} & & & \\
(L \cap R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & L^{\mathbf{r}} \cap R^{\mathbf{r}} & & & \\
(\sim R)^{\mathbf{r}} & \overset{\text{DEF}}{=} & \sim(R^{\mathbf{r}}) & & &
\end{array}
$$

Note how in the lookarounds cases we replace each constructor with its dual in the opposite direction. It follows by induction over R that $|R^{\mathbf{r}}| = |R|$ and that $(R^{\mathbf{r}})^{\mathbf{r}} = R$. Theorem 1 is instrumental in linking the derivative based definition (which uses reversal based definition of lookbehinds) with the formal match semantics.

Theorem 1. $\forall R \in \mathbf{RE}, sp \in \mathbf{Span} : sp \models R \Leftrightarrow sp^{\mathbf{r}} \models R^{\mathbf{r}}$

Note that, by the reversal theorem and the fact that reversal of regexes is an involutive operation, we can a posteriori restate the matching semantics of lookbehinds in terms of regex reversal rather than span reversal.

In order to prove Theorem 1 in Lean it was occasionaly useful to reason about the equivalence of regular expressions in an equational way using a setoid reasoning-style approach, as it can be idiomatically done in Agda and Idris and that Lean also supports using the **`calc`** abstraction. In order to achieve this, we define the relation `models_equivalence` and show that it is indeed an equivalence relation and a congruence with respect to concatenation.

```
def models_equivalence : Prop := sp ⊨ r ↔ sp ⊨ q
infixr:30 " ↔ᵣ " => models_equivalence
```

```
theorem equiv_trans (rq : r ↔ᵣ q) (qp : q ↔ᵣ p) : r ↔ᵣ p
theorem equiv_sym (rq : r ↔ᵣ q) : q ↔ᵣ r
theorem equiv_refl : r ↔ᵣ r
theorem equiv_cat_cong (rr : r ↔ᵣ r') (qq : q ↔ᵣ q') :
 r · q ↔ᵣ r' · q'
```

We make use of the $\leftrightarrow_r$ relation in the theorems below which allows us to prove associativity of concatenation, as well as the left and right identity laws. Moreover, we prove two useful properties of the `repeat_cat` function which will be needed for the star case in the proof of reversal. In particular, we use this reasoning to prove intermediate lemmas about `repeat_cat`.

```
theorem equiv_cat_assoc : ((r · q) · w) ↔ᵣ (r · (q · w))
theorem equiv_eps_cat : ε · r ↔ᵣ r
theorem equiv_cat_eps : r · ε ↔ᵣ r
theorem equiv_repeat_cat_cat : r ⟮ m ⟯ · r ↔ᵣ r · r ⟮ m ⟯
theorem equiv_reverse_regexp_repeat_cat {r : RE α} {m : ℕ}:
 (r ⟮ m ⟯) ʳ ↔ᵣ (r ʳ) ⟮ m ⟯
```

The following theorem witnesses the correctness of the reversal operation on regexes, as done in Theorem 1.

```
theorem models_reversal : sp ⊨ R ↔ spʳ ⊨ Rʳ
```

The proof follows by induction on `R`. We found proving this theorem using the semantic definition more intuitive than using the derivative-based matching.

4 Location Based Derivatives

Here we formally define the framework of *location based derivatives* and introduce its Lean representation. The definitions are based on [27] that builds on [17] and is a generalization of [4].

We start with some intuition behind the notion of location based derivatives. Classically, a derivative of a regex R is taken for a given character and *nullability* of R is a static property of R that holds iff R matches the empty word.

The derivative of a regex $R \in \mathbf{RE}$ for a location x, denoted by $\mathbf{der}(R,x) \in \mathbf{RE}$, mimics the classical definition, except that if R is a concatenation $R_1{\cdot}R_2$ then nullability of R_1 depends on x and all lookarounds ℓ are treated the same as ε ($\mathbf{der}(\ell,x) = \bot$). In all other aspects, $\mathbf{der}(R,x)$ computes a derivative in the style of [4] that is then used to continue matching from the next location.

When lookarounds are used, the notion of nullability becomes *location dependent*. Nullability of R in a location $w[i]$, denoted by $\mathbf{null}(R, w[i])$, means in terms of the formal semantics that $w[i,0] \models R$, i.e., R matches the empty word $w_{i,0}$ in position i, which is always true when R is either ε or a Kleene star loop, but may in general depend on i. For example, `(?=\n)` matches an internal end of line, e.g., if $w =$ "a\n\nbcde" then $w[i,0] \models$ `(?=\n)` only for $i = 1, 2$.

In order to capture the semantics of lookarounds, nullability test of a lookaround recursively invokes existence of a match that also uses derivatives. To explain this more clearly, we define, for $n \geq 0$, the *n'th derivative* of R from location x, $\mathbf{der}^n_x(R)$, as follows:

$$\mathbf{der}^0_x(R) \stackrel{\text{DEF}}{=} R, \quad \mathbf{der}^{n+1}_x(R) \stackrel{\text{DEF}}{=} \mathbf{der}^n_{x+1}(\mathbf{der}(R,x))$$

In the case of a lookahead, nullability $\mathbf{null}(\texttt{(?=}R\texttt{)}, x)$ tests existence of a match of R starting from location x, meaning that there exists $n \geq 0$ such that $\mathbf{der}^n_x(R)$ is nullable in the location $x{+}n$. For lookbehind, $\mathbf{null}(\texttt{(?<=}R\texttt{)}, x)$ uses reversal to test $\mathbf{null}(\texttt{(?=}R^{\mathbf{r}}\texttt{)}, x^{\mathbf{r}})$.

Consider `(?<=\n)` as an example and let $x = w[3]$ with w as above. In this case $\mathbf{null}(\texttt{(?<=\n)}, x)$ tests if there exists a derivation of `\n` starting from $x^{\mathbf{r}} =$ "edcb\n\na"[4]. The derivative $\mathbf{der}(\texttt{\n}, x^{\mathbf{r}}) = \varepsilon$ because $w^{\mathbf{r}}_4 =$ `\n` and ε is nullable in all locations, so $\mathbf{null}(\texttt{(?<=\n)}, x) = \mathbf{true}$.

The main intuition behind derivative based matching of a regex $\texttt{(?<=}L\texttt{)}{\cdot}R$ from a location $x = w[i]$ is illustrated in Figure 4 as a state machine with conditional branching, whose states are regexes with $\texttt{(?<=}L\texttt{)}{\cdot}R$ as the initial state and the state $\mathbf{der}^n_x(R)$ is conditionally accepting in location $x + n$ iff it is nullable in $x + n$.

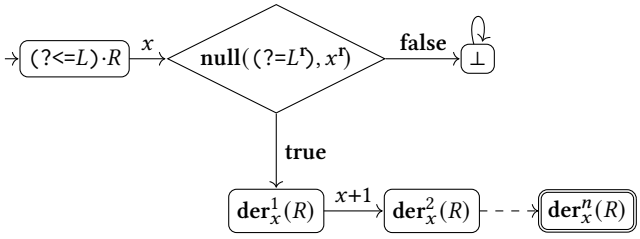

Figure 4. Derivative based matching of $\texttt{(?<=}L\texttt{)}{\cdot}R$.

The relationship to the formal semantics follows from Section 5, namely that $w[i,n] \models \texttt{(?<=}L\texttt{)}{\cdot}R$. The figure is simplified by treating $\bot$ as the zero element of $\cdot$ and as the unit element of $\cup$, and where $\mathbf{der}(\texttt{(?<=}L\texttt{)}, x) = \bot$.

4.1 Main Derivative Function

The *location derivative* $\mathbf{der}(R,x)$ of $R \in \mathbf{RE}$ for a nonfinal location $x \in \mathbf{Loc}^+$ is defined as follows. Let $\ell \in \mathbf{LA}$, $\phi \in \alpha$:

$$\begin{aligned}
\mathbf{der}(\varepsilon, x) &\stackrel{\text{DEF}}{=} \bot \\
\mathbf{der}(\ell, x) &\stackrel{\text{DEF}}{=} \bot \\
\mathbf{der}(\phi, x) &\stackrel{\text{DEF}}{=} \begin{cases} \varepsilon, & \textbf{if } (x.2)_0 \in [\![\phi]\!]; \\ \bot, & \textbf{otherwise}. \end{cases} \\
\mathbf{der}(L{\cdot}R, x) &\stackrel{\text{DEF}}{=} \begin{cases} \mathbf{der}(L,x){\cdot}R \cup \mathbf{der}(R,x), & \textbf{if } \mathbf{null}(L,x); \\ \mathbf{der}(L,x){\cdot}R, & \textbf{otherwise}. \end{cases} \\
\mathbf{der}(L \cup R, x) &\stackrel{\text{DEF}}{=} \mathbf{der}(L,x) \cup \mathbf{der}(R,x) \\
\mathbf{der}(L \cap R, x) &\stackrel{\text{DEF}}{=} \mathbf{der}(L,x) \cap \mathbf{der}(R,x) \\
\mathbf{der}(R*, x) &\stackrel{\text{DEF}}{=} \mathbf{der}(R,x) \cdot R* \\
\mathbf{der}(\sim R, x) &\stackrel{\text{DEF}}{=} \sim\mathbf{der}(R,x)
\end{cases}
\end{aligned}$$

In Lean, we totalize the location derivative so that it is defined for final locations as well. The only case we need to adapt is the one for predicates: we set $\mathbf{der}(\phi, \langle u, \epsilon\rangle) \stackrel{\text{DEF}}{=} \bot$. The Boolean function $\mathbf{null}(R,x)$ checks if $R \in \mathbf{RE}$ is *nullable*

at the location $x \in \mathbf{Loc}$, is defined by induction over **RE**:

$$\begin{aligned}
\mathbf{null}(\varepsilon, x) &\stackrel{\text{DEF}}{=} \mathbf{true} \\
\mathbf{null}(\phi, x) &\stackrel{\text{DEF}}{=} \mathbf{false} \\
\mathbf{null}(R*, x) &\stackrel{\text{DEF}}{=} \mathbf{true} \\
\mathbf{null}(L \Cup R, x) &\stackrel{\text{DEF}}{=} \mathbf{null}(L, x) \vee \mathbf{null}(R, x) \\
\mathbf{null}(L \Cap R, x) &\stackrel{\text{DEF}}{=} \mathbf{null}(L, x) \wedge \mathbf{null}(R, x) \\
\mathbf{null}(L \cdot R, x) &\stackrel{\text{DEF}}{=} \mathbf{null}(L, x) \wedge \mathbf{null}(R, x) \\
\mathbf{null}(\sim R, x) &\stackrel{\text{DEF}}{=} \neg\, \mathbf{null}(R, x) \\
\mathbf{null}((?{=}R), x) &\stackrel{\text{DEF}}{=} \mathbf{existsMatch}(R, x) \\
\mathbf{null}((?{<}{=}R), x) &\stackrel{\text{DEF}}{=} \mathbf{existsMatch}(R^{\mathbf{r}}, x^{\mathbf{r}}) \\
\mathbf{null}((?!R), x) &\stackrel{\text{DEF}}{=} \neg\, \mathbf{existsMatch}(R, x) \\
\mathbf{null}((?{<}!R), x) &\stackrel{\text{DEF}}{=} \neg\, \mathbf{existsMatch}(R^{\mathbf{r}}, x^{\mathbf{r}})
\end{aligned}$$

Existence of a match in $R \in \mathbf{RE}$ from location $x \in \mathbf{Loc}$ is then defined as follows:

$$\begin{aligned}
&\mathbf{existsMatch}(R, x) \stackrel{\text{DEF}}{=} \\
&\quad \mathbf{null}(R, x) \vee (x.2 \neq \epsilon \wedge \mathbf{existsMatch}(\mathbf{der}(R, x), x + 1))
\end{aligned}$$

These are defined in Lean by a mutually recursive definition of `null`, `existsMatch` and `der`. See Figure 5. The mutual dependence arises from the presence of lookarounds. More specifically, we have to check if there exists a valid match when performing nullability checks for lookarounds.

Given the interdependence of the definitions due to lookarounds, these mutually inductive definitions need a specific metric to show that the recursion is indeed terminating. The metric is defined as follows, by carefully combining lexicographically `lookaround_height`, size of the regex, remaining length of the string to be read, and a flag to prioritize the checking of `existsMatch` with respect to the other functions.

```
def der_termination_metric (r : RE α) (x : Loc σ) (n : ℕ) :
  ℕ × ℕ × ℕ × ℕ :=
  (lookaround_height r, sizeOf x.right, sizeOf_RE r, n)
```

Another crucial aspect for termination that has been omitted from the definition above is the output type of the `der` function, which is given as the following subtype:

```
def der (R : RE α) (x : Loc σ) :
  {r : RE α // lookaround_height r ≤ lookaround_height R}
```

The main idea is to encapsulate the output regex with a proof that it is structurally smaller than the input one with respect to one of the metrics used. This trick is used to simultaneously provide a definition for `der` and show that the mutual calls are indeed terminating; the property is specifically necessary in the `der` call in `existsMatch` to show that the whole induction terminates.

4.2 Derivation Relation

The *derivation relation* $x \xrightarrow{R} y$ for $R \in \mathbf{RE}$ and $span(x, y) \in \mathbf{Span}$ defines reachability from location x to location y via derivatives of R, where $span(x, y)$ denotes a span sp such that $x = loc(sp)$ and $y = end(sp)$:

$$\begin{aligned}
x \xrightarrow{R} y \quad &\stackrel{\text{DEF}}{=} \quad (\mathbf{null}(R, x) \wedge x = y) \\
&\vee \quad (x.2 \neq \epsilon \wedge x{+}1 \xrightarrow{\mathbf{der}(R,x)} y)
\end{aligned}$$

```
mutual
def der (R : RE α) (x : Loc σ) : RE α :=
  match R with
  | ε            => Pred ⊥
  | ?= R         => Pred ⊥
  | ?<= R        => Pred ⊥
  | ?! R         => Pred ⊥
  | ?<! R        => Pred ⊥
  | Pred φ       =>
    match x with
    | (_ , [])   => Pred ⊥
    | (_ , c::_) => if denote φ c then ε else Pred ⊥
  | L · R        => if null L x then
                      der L x · R ⋓ der R x
                    else
                      der L x · R
  | L ⋓ R        => der L x ⋓ der R x
  | L ⋒ R        => der L x ⋒ der R x
  | R *          => der R x · R *
  | ~ R          => ~ der R x
def null (R : RE α) (x : Loc σ) : Bool :=
  match R with
  | ε      => true
  | Pred φ => false
  | L · R  => null L x && null R x
  | L ⋓ R  => null L x || null R x
  | L ⋒ R  => null L x && null R x
  | R *    => true
  | ~ R    => ¬ null R x
  | ?= R   => existsMatch R x
  | ?<= R  => existsMatch R^r x^r
  | ?! R   => ¬ existsMatch R x
  | ?<! R  => ¬ existsMatch R^r x^r
def existsMatch (R : RE α) (x : Loc σ) : Bool :=
  match x with
  | (u, [])    =>
    null R (u, [])
  | (u, a::v) =>
    null R (u, a::v)
    || existsMatch (der R (u, a::v)) (a::u, v)
end
termination_by
  null R x        => der_termination_metric R x 0
  existsMatch R x => der_termination_metric R x 1
  der R x         => der_termination_metric R x 0
```

Figure 5. Formal definition of derivatives in Lean.

Lemma 1 is an adjustment of [27, Theorem 1] to the formalism and notations used here and also treats loops slightly differently (but equivalently). More concretely, [27] uses finite loops (counters) for the semantics of Kleene star while we use repetitions here (which simulate counters by iterated concatenation) to simplify the RE type. Note how each lemma effectively mirrors the matching semantics given in Section 3.2.

Lemma 1. $\forall L, R \in \mathbf{RE}, \psi \in \alpha, span(x, y) \in \mathbf{Span}$:

1. $x \xrightarrow{\varepsilon} y \Leftrightarrow x = y$
2. $x \xrightarrow{\Downarrow} y \Leftrightarrow (x.2)_0 \in [\![\psi]\!] \wedge y = x + 1$
3. a. $x \xrightarrow{(?=R)} y \Leftrightarrow x = y \wedge \exists z : x \xrightarrow{R} z$

b. $x \xrightarrow{(?!R)} y \Leftrightarrow x = y \wedge \nexists z : x \xrightarrow{R} z$
4. *a.* $x \xrightarrow{(?<=R)} y \Leftrightarrow x = y \wedge \exists z : x^{\mathrm{r}} \xrightarrow{R^{\mathrm{r}}} z$
b. $x \xrightarrow{(?<!R)} y \Leftrightarrow x = y \wedge \nexists z : x^{\mathrm{r}} \xrightarrow{R^{\mathrm{r}}} z$
5. $x \xrightarrow{L \cup R} y \Leftrightarrow x \xrightarrow{L} y \vee x \xrightarrow{R} y$
6. $x \xrightarrow{L \cap R} y \Leftrightarrow x \xrightarrow{L} y \wedge x \xrightarrow{R} y$
7. $x \xrightarrow{\sim R} y \Leftrightarrow \neg(x \xrightarrow{R} y)$
8. $x \xrightarrow{L \cdot R} y \Leftrightarrow \exists z : x \xrightarrow{L} z \xrightarrow{R} y$
9. $x \xrightarrow{R*} y \Leftrightarrow \exists n : x \xrightarrow{R^{(n)}} y$

In Lean, the derivation relation is encoded by the function `derives sp R` for which we introduce the infix notation `sp ⊢ R`. If `sp.match` is empty, we simply check that `R` is also nullable at the current span `sp`. Otherwise, we continue moving forward in the string and check that the derivative of `R` (with respect to `sp.loc`) matches.

```
def derives (sp : Span σ) (R : RE α) : Bool :=
 match sp with
 | ⟨s, [], v⟩    => null R sp.loc
 | ⟨s, c::u, v⟩ => derives ⟨c::s, u, v⟩ (der R sp.loc)
infix:40 " ⊢ " => derives
```

Note that the `derives` function has the advantage of effectively being a decidable relation, whereas the `models` relation only defines a specification in **Prop**. Thanks to the equivalence theorem that we will present in Section 5, this effectively also provides a decision procedure for the classical match semantics for any given span and regex.

Notationally, given a span *sp* we use the following functions to refer to the position (i.e., the length of the left part) of the matching start location *loc*(*sp*) and matching end location *end*(*sp*):

```
def Span.i (sp : Span σ) : ℕ := sp.left.length
def Span.j (sp : Span σ) : ℕ := sp.i + sp.match.length
```

Here, `sp.i` is the start position and `sp.j` is the end position of `sp.match`.

Our implementation choice for span is to use a zipper-like data-stucture. The `derives` function could have been alternatively defined on a string with two indices or on two locations, with one encoding the start of the match and the second encoding the end of the match. We believe that we chose the more natural span representation because it intrinsically preserves both the invariant that `sp.i ≤ sp.j` (i.e., the two locations cannot be arbitrarily swapped) and the fact that the two locations indeed refer to the same string (i.e., the concatenation of the left and right components of the two locations coincide). Moreover, the span representation directly captures the matching string and the match length (`sp.j - sp.i`), which allows us to apply induction on the length of the match. In practice this is reflected in the test for nullability, when `sp.i = sp.j` and there is a match at the current position, or alternatively advance `sp.i` to the next position and take the derivative of the regex (match later).

5 Equivalence

Our main correctness theorem shows the equivalence between the derivative relation and the match semantics relation. Let $sp \vdash R \overset{\text{DEF}}{=} loc(sp) \xrightarrow{R} end(sp)$.

Theorem 2. $\forall R \in \mathbf{RE}, sp \in \mathbf{Span} : sp \vdash R \Leftrightarrow sp \models R$

In Lean, we state the theorem accordingly:

```
theorem correctness : sp ⊢ R ↔ sp ⊨ R
```

There are two general approaches to proving this theorem. The first approach is to prove the main correctness by induction on the regular expression `R` and show that, for each regex case, the definition of `⊨` induces a property that is also satisfied by the derivation relation with `⊢`. For example, the theorem statement of `derives_Cat` looks identical to the concatenation case in `models`, but using `derives` instead of the matching relation.

```
theorem derives_Cat :
 sp ⊢ (l · r) ↔
 ∃ u₁ u₂,
    ⟨sp.left, u₁, u₂ ++ sp.right⟩ ⊢ l
  ∧ ⟨u₁ʳ ++ sp.left, u₂, sp.right⟩ ⊢ r
  ∧ u₁ ++ u₂ = sp.match
```

This is the approach we follow in the formalization, as it is more modular and allows the main properties of the matching relation (e.g. `derives` on a disjunction is indeed equivalent to the disjunction of the individual `derives` calls) to be directly applied to other theorems, which mainly use the derivation relation.

One alternative way of proving correctness could have been by induction on the match length, and then show that both in the nullable and the inductive case the `null` and `der` functions are indeed the correct definitions with respect to the match semantics relation. The main correctness theorem then follows directly by induction on the match length and unfolding the definition of `derives`. In other words, the correctness would rely on these lemmas:

```
theorem models_to_derives_null :
 ⟨s, [], v⟩ ⊨ R ↔ null R ⟨s, v⟩

theorem models_to_derives_der :
 ⟨s, c::u, v⟩ ⊨ R ↔ ⟨c::s, u, v⟩ ⊨ (der R ⟨s, (c::u) ++ v⟩)
```

We simply prove these properties using the correctness obtained with the first approach. Moreover, the following properties follow from Theorem 2, and Theorem 1 in 5.1(3).

Corollary 5.1. *For all* $R \in \mathbf{RE}$ *and* $span(x, y) \in \mathbf{Span}$*:*

1. $\mathbf{null}(R, x) \Leftrightarrow span(x, x) \models R$
2. $\mathbf{existsMatch}(R, x) \Leftrightarrow \exists y : span(x, y) \models R$
3. $x \xrightarrow{R} y \Leftrightarrow y^{\mathrm{r}} \xrightarrow{R^{\mathrm{r}}} x^{\mathrm{r}}$

5.1 Match Algorithm

After having proven the correctness of the `derives` relation with respect to the classical match semantics `models`, we show how a concrete match finding algorithm based on derivatives can be extracted and proven correct with respect

to the relations previously introduced. One of the main advantage of the algorithm with respect to other approaches for matching is that it is linear with respect to the string (assuming bounded length lookarounds). The top-level match algorithm **llmatch**(R, w) takes a word $w \in \sigma^*$ and a regex $R \in$ **RE**, and either returns none if there exists no match for R or a concrete span $w[i{:}j] \models R$ such that i is minimal and j is maximal for the given i, i.e., the match returned is at the leftmost location with the longest length possible.

We first introduce some auxiliary definitions before presenting the matching algorithm directly in Lean. The function `null?` tests nullability of a given location with respect to a regex, either returning the current location (as a span), or failing otherwise.

```
def null? (r : RE α) (x : Loc σ) : Option (Span σ) :=
 if null r x then
   some x.as_span
 else
   none
```

Moreover, the function `increase_match_left` takes a span `sp` and increases the `sp.match` portion of the span to include the previous character before the start of the match. This function is used in inductive cases to increase the match given by a recursive call, and, thanks to the zipper-like definition of a span, it is a constant-time operation.

```
def Span.increase_match_left (sp : Span σ) : Span σ :=
  match sp with
  | ⟨[], u, v⟩ => ⟨[], u, v⟩
  | ⟨c::s, u, v⟩ => ⟨s, c::u, v⟩
```

The central part of the matching algorithm is the function **maxMatchEnd**(x, R), which finds the *maximal* (*rightmost*) match end location relative to a given start location x. We define it as follows, using the `Option` type in Lean to represent the case in which there is no match at the given start location.

```
def maxMatchEnd (r : RE α) (x : Loc σ) : Option (Span σ):=
 match x with
 | ⟨_,[]⟩ => null? r x
 | ⟨u,c::v⟩ =>
  match maxMatchEnd (der r x) ⟨c::u,v⟩ with
  | none => null? r x
  | some res => some res.increase_match_left
termination_by
  maxMatchEnd r x => x.right
```

The reason why we alternate between locations and spans in the definition of `maxMatchEnd` is because the correctness of `maxMatchEnd` in Theorem 3 and the top-level algorithm is stated with respected to `derives`, which is defined on spans.

Observe that the structure of `maxMatchEnd` (which produces a concrete match with maximal ending location) is almost identical to that of `existsMatch` (which simply says whether a match exists with the given starting location) and `derives` (which only tests whether a given span is indeed a match).

Dually, we also provide a definition of `minMatchStart` which, given an end location, finds the minimum starting location that matches the regex. This is simply obtained by reversing the inputs (both spans and regexes) and the output of `maxMatchEnd`.

```
def minMatchStart (r : RE α) (x : Loc σ) : Option(Span σ):=
  Option.map Span.reverse (maxMatchEnd r^r x^r)
```

Despite it being defined in terms of `maxMatchEnd`, we abstract away the implementation details of `minMatchStart` by proving its properties as if it was a separate function, and we will use it in the top-level algorithm `llmatch` to find the leftmost location.

We identify the following correctness and completeness properties with respect to the `maxMatchEnd`, which can be proven by induction on the structure of the input location. The correctness of `minMatchStart` can be defined similarly by dualizing appropriately.

Theorem 3. *For all* $R \in$ **RE** *and* $x \in$ **Loc***:*

$$\textbf{maxMatchEnd}(x, R) = \begin{cases} \text{none}, & \textbf{if } \nexists y : x \xrightarrow{R} y \\ \text{some } \max\{y \mid x \xrightarrow{R} y\}, & \textbf{otherwise}. \end{cases}$$

Note that although we stated correctness in terms of locations, our implementation returns a span, thus encoding both a start and final location. This is justified by a crucial invariant of the two algorithms, which is that the initial (resp., final) matching location of the match returned by `maxMatchEnd` (resp., `minMatchEnd`) is at the same location of the one provided as input. This invariant is crucial to prove the correctness of the top-level algorithm later introduced, and we state it as follows by first introducing the intuitive notion of a location `loc` being the starting match location of a span `sp`:

```
def derivesStartLocation (loc : Loc σ) (sp2 : Span σ):Prop
 loc.pos = sp2.i ∧ loc.string = sp2.string

theorem maxMatchEnd_derivesStartLocation
 (matching : maxMatchEnd r x = some sp_out) :
 derivesStartLocation x sp_out
```

The correctness and completeness can then be defined with the following statements, with similarly dualized definitions in the case of `minMatchStart`. The fact that this theorem follows directly by induction on the match length is what justifies the introduction of the derivation relation and its equivalence with the match semantics.

```
theorem maxMatchEnd_max
 (matching : maxMatchEnd r x = some sp_out) :
 (∀ sp, derivesStartLocation x sp
      → sp ⊢ r
      → sp.j ≤ sp_out.j)

theorem maxMatchEnd_correct
 (matching : maxMatchEnd r x = some sp_out) : sp_out ⊢ r

theorem maxMatchEnd_no_match
 (matching : maxMatchEnd r x = none) :
 (∀ sp, derivesStartLocation x sp → ¬(sp ⊢ r))
```

5.2 Top-Level Matching Algorithm

Finally, the top-level matching algorithm `llmatch` takes a regex `R` and a word `w`, and returns the leftmost longest match in `w`, that is Lean is defined as follows:

```
def llmatch (R : RE α) (w : List σ) : Option (Span σ) := do
  let sp ← minMatchStart (R · (Pred ⊤)*) w.as_end_location
  maxMatchEnd R sp.loc
```

The algorithm is written in monadic style using **do**-notation in the `Option` monad, which effectively means that the algorithm returns `none` when either the calls to `maxMatchEnd` or the call to `minMatchStart` fails. Informally the algorithm can equivalently be described as follows:

$$\textbf{llmatch}(R, w) \stackrel{\text{DEF}}{=} \textbf{let}\ x = \textbf{maxMatchEnd}(w^{\mathrm{r}}[0], \top{*}{\cdot}R^{\mathrm{r}})\ \textbf{in} \begin{cases} \mathsf{none}, & \textbf{if}\ x = \mathsf{none}; \\ \mathsf{some}\ \mathit{span}(x^{\mathrm{r}}, \textbf{maxMatchEnd}(x^{\mathrm{r}}, R)), & \textbf{otherwise}. \end{cases}$$

We now describe the general idea of the algorithm more in detail and outline the idea behind its correctness, which we state in Theorem 4. The idea is to call the auxiliary function `maxMatchEnd` twice. First, we determine the leftmost location by calling `minMatchStart` (and thus, indirectly via reversal, `maxMatchEnd`). Then, we supply the leftmost location we found as a start location to `maxMatchEnd`, which provides the longest match found from that location.

More in detail, in the first call to `minMatchStart` we supply as end location the end location of the entire string. In practice, this is simply obtained by reversing the string `w`, and then having no characters to read on the right:

```
def List.as_end_location (w : List σ) : Loc σ := ⟨w^r, []⟩
```

The use of `R · (Pred ⊤)*` is then crucial to ensure that the match length does not play a role in finding the leftmost location, while at the same time ensuring the that the left position matches the regex `R`. Indeed, using the boundary invariants previously mentioned, the span returned by `minMatchStart` is such that the end match position is always going to be the input one, i.e., the end of the string. Thus, it is effectively ignored, and this is reflected by the fact that we simply supply `sp.loc` to the second call.

The role of the second call is conceptually simpler, and is used to find the end of the longest match from the leftmost location. This follows directly by the correctness of the `maxMatchEnd` stated in Section 5.1. Using again the boundary invariant, the left location of the span returned is still going to be the leftmost location.

We formally state the correctness and completeness of the algorithm and subsequently show the theorems as we formalized them in Lean. The following is the correctness theorem of **llmatch**.

Theorem 4. *$\forall w \in \sigma^*, R \in \mathbf{RE}$:*

1. $\textbf{llmatch}(R, w) = \mathsf{none} \Leftrightarrow \nexists i, j : w[i{:}j] \models R$;
2. $\textbf{llmatch}(R, w) = w[i{:}j] \Rightarrow$
   $i = \min\{\imath \mid \exists \jmath : w[\imath{:}\jmath] \models R\} \wedge$
   $j = \max\{\jmath \mid w[i{:}\jmath] \models R\}$.

Proof. Theorem 2 is used implicitly. We have that for all $z \in \mathbf{Loc}$ and all $L \in \mathbf{RE}$, $\textbf{maxMatchEnd}(z, L) = \max\{y \mid z \xrightarrow{L} y\}$. Thus, if $x = \max\{y \mid w^{\mathrm{r}}[0] \xrightarrow{\top{*}{\cdot}R^{\mathrm{r}}} y\} \neq \mathsf{none}$ then $x^{\mathrm{r}} = \min\{y \mid y \xrightarrow{R{\cdot}\top{*}} w[|w|]\}$ by using basic properties of reversal and Theorem 1. So $x^{\mathrm{r}} = w[i]$ is the leftmost (minimal) location in w because $\top{*}$ matches all spans. Then $\textbf{maxMatchEnd}(w[i], R) = w[j]$ is such that $j{-}i$ is longest (maximal). Statement (2) follows.

Statement (1) follows if $x = \mathsf{none}$ because then no match end location for $\top{*}{\cdot}R^{\mathrm{r}}$ exists in w^{r} since $\top{*}$ matches all prefixes of w^{r} and thus no start location for $R{\cdot}\top{*}$ exists in w by using Theorem 1, and thus no match for R exists in w. □

We now show the main theorems on the top-level algorithm as they are proven in Lean. First, we prove that the match returned by the algorithm is indeed a match with respect to the derivation relation. This follows directly from the correctness of `maxMatchEnd`. Note that for simplicity, we use the derivation relation to prove the properties about `llmatch` even though it is equivalent to the match semantics.

```
theorem llmatch_matches
 (matching : llmatch r x = some sp_out) : sp_out ⊢ r
```

When a match is found, we show that it is indeed the one with minimum left position with respect to any other span which recognizes the same string `w`.

```
theorem llmatch_leftmost
 (m : llmatch r w = some sp_out) :
 (∀ sp, sp.string = w
      → sp ⊢ r
      → sp_out.i ≤ sp.i)
```

Similarly, the match given as output is the longest with respect to all those spans which recognize the same string and that start at the same leftmost location.

```
theorem llmatch_longest
 (m : llmatch r w = some sp_out) :
 (∀ sp, sp.string = w
      → sp.i = sp_out.i
      → sp ⊢ r
      → sp_out.match.length ≥ sp.match.length)
```

Finally, we have the completeness property, ensuring that a match is found whenever there is one:

```
theorem llmatch_no_match
 (m : llmatch r w = none) :
 (∀ sp, sp.string = w
      → ¬(sp ⊢ r))
```

5.3 Executing Matching in Lean

All the definitions in our Lean formalization are computable, and, except for the match semantics, the predicates take values in `Bool` instead of **`Prop`**. These definitions can therefore be directly evaluated in Lean [6] as executable code. We take Example 3.1 and show a test run for it. The character classes for upper and lower case letters, and digits are approximated as `uc`, `lc` and `d`, respectively.

```
def uc := "ABCDEFGHIJKLMOPQRSTUVWXYZ".characterClass
def lc := "abcdefghijklmopqrstuvwxyz".characterClass
def d  := "0123456789".characterClass
def us := "_".characterClass
def w  := uc ⊔ lc ⊔ d ⊔ us

def Ts [EffectiveBooleanAlgebra α σ] : RE α := (Pred ⊤) *
def W : RE (BA Char) := Pred w^c
def R₁ : RE (BA Char) := Ts · Pred uc · Ts
def R₂ : RE (BA Char) := Ts · Pred d · Ts · Pred d · Ts
def R₃ : RE (BA Char) := Ts · Pred lc · Ts
def R₄ : RE (BA Char) := (?<= W) · Ts · (?= W)
def R  : RE (BA Char) := R₁ ⋒ R₂ ⋒ R₃ ⋒ R₄

#eval llmatch R "0B:1aD2;e".toList
 -- some ([':','B','0'], ['1','a','D','2'], [';','e'])
```

Note that in the example above, we defined `W` as a singleton regex matching any *non-word-letter*. In order to illustrate the difference between `^c` and `~`, we define the `W'`, `R₄'` variants which use the regex complement `~` rather than `^c` of the EBA.

```
def W'  : RE (BA Char) := ~ Pred w
def R₄' : RE (BA Char) := (?<= W') · Ts · (?= W')
def R'  : RE (BA Char) := R₁ ⋒ R₂ ⋒ R₃ ⋒ R₄'
```

The effect of this change becomes apparent when we execute `llmatch` with the modified `R'` and the same input string, which matches the same string and, somewhat counterintuitively, has different semantics.

```
#eval llmatch R' "0B:1aD2;e".toList
-- some ([], ['0','B',':','1','a','D','2',';','e'], [])
```

The key observation is that the negation inside `W` is still an atomic predicate, which in particular means that the matching length is always going to be exactly one character. However, complementing a singleton regex allows matches of any length different from one, hence the different semantics displayed in the two cases. In particular, we have, for example, that `⊤` matches any singelton string, so the empty string, as well as any string of two or more characters, is a match for `~⊤`, i.e., `~⊤` is equivalent to `ε ⋓ ⊤·⊤+`.

The true power of `~` comes into play when we want to express a property such as "the match must not contain two or more digits in a row", e.g., using the regex `S`:

```
def S : RE (BA Char) := ~ (Ts · Pred d · Pred d · Ts)
```

Then, for example,

```
#eval llmatch R ⋒ S "0B:1aD23;e".toList
-- none
```

while the original input string still contains a match

```
#eval llmatch  R ⋒ S "0B:1aD2;e".toList
 -- some ([':','B','0'], ['1','a','D','2'], [';','e'])
```

and the modified input string contains a match for `R` alone

```
#eval llmatch  R "0B:1aD23;e".toList
 -- some ([':','B','0'], ['1','a','D','2','3'], [';','e'])
```

6 Rewrites

There are several kinds of optimizations that can be applied to derivatives. Rewrites can be applied to regexes while preserving their correctness according to Theorem 2. A further property that we prove below is that negative lookarounds can be rewritten to positive lookarounds through *anchoring*.

6.1 Inlined Simplifications

Our current formalization of derivatives in Lean does not inline rewrites at the level of regular expressions. However, such rules, analogous to the ones in [19], by building on Theorem 2, can be added as a separate optimization layer, in particular for eliminating trivial cases involving ε as the unit of $\cdot$ and $\bot$ as the zero of $\cdot$ and the unit of $\curlyvee$. For example, if $\ell \in \mathbf{LA}$ then the following simplified version of the derivation rule for concatenation can be inlined directly into the definition, which was used earlier implicitly in Figure 4:

$$\mathbf{der}(\ell{\cdot}R, x) = \mathbf{if}\ \mathbf{null}(\ell, x)\ \mathbf{then}\ \mathbf{der}(R, x)\ \mathbf{else}\ \bot$$

However, such inlining of simplifications in the general definition of derivatives needs care, because we also intend to support variants of $\curlyvee$ (and $\curlywedge$) that are potentially non-commutative (see Section 8). For example, in the case of backtracking semantics, $(L \curlyvee R) \cdot S$ is *backtrack-equivalent* to $L \cdot S \curlyvee R \cdot S$ but $S \cdot (L \curlyvee R)$ is *not backtrack-equivalent* $S \cdot L \curlyvee S \cdot R$, i.e., *the distributivity law of concatenation over alternation applies from right to left but does not apply from left to right* in order to preserve backtracking semantics.

6.2 Elimination of General Negative Lookarounds

We define the following regexes, where `\A` is called the *start anchor* and `\z` is called the *end anchor*:

$$\texttt{\textbackslash A} \overset{\text{DEF}}{=} \texttt{(?<!⊤)} \qquad \texttt{\textbackslash z} \overset{\text{DEF}}{=} \texttt{(?!⊤)}$$

In the case of `\A` there cannot exist a location $w[i{-}1]$ immediately before $w[i]$ (or else $w_{i-1} \in [\![\top]\!] = \sigma$), and in the case of `\z` there cannot exist a location $w[i{+}1]$ immediately after $w[i]$ (or else $w_i \in [\![\top]\!] = \sigma$). It follows that $w[i{:}j] \models$ `\A` iff $i = j = 0$, and $w[i{:}j] \models$ `\z` iff $i = j = |w|$. The following basic property follows from the definition of match semantics.

Theorem 5. $\forall R \in \mathbf{RE}$:

1. $\texttt{(?=~(}R\texttt{·⊤*)·\z)} \equiv \texttt{(?!}R\texttt{)}$
2. $\texttt{(?<=\A·~(⊤*·}R\texttt{))} \equiv \texttt{(?<!}R\texttt{)}$.

Proof. We prove (1). Let $R \in \mathbf{RE}$ and $span(x, y) \in \mathbf{Span}$. The proof steps are easier to view by using Theorem 2 although

it follows from the match semantics directly.

$$\begin{aligned} x \xrightarrow{(?=\sim(R\cdot\top*)\cdot\backslash z)} y &\Leftrightarrow x{=}y \wedge \exists z : x \xrightarrow{\sim(R\cdot\top*)\cdot\backslash z} z \\ &\Leftrightarrow x{=}y \wedge \exists z, z' : x \xrightarrow{\sim(R\cdot\top*)} z' \xrightarrow{\backslash z} z \\ &\Leftrightarrow x{=}y \wedge x \xrightarrow{\sim(R\cdot\top*)} w[|w|] \\ &\Leftrightarrow x{=}y \wedge \neg(x \xrightarrow{R\cdot\top*} w[|w|]) \\ &\Leftrightarrow x{=}y \wedge \nexists z : x \xrightarrow{R} z \xrightarrow{\top*} w[|w|] \\ &\Leftrightarrow x{=}y \wedge \nexists z : x \xrightarrow{R} z \\ &\Leftrightarrow x \xrightarrow{(?!R)} y \end{aligned}$$

where we used that $z \xrightarrow{\top*} w[|w|]$ is always true because $\forall sp \in \textbf{Span} : sp \models \top*$. Statement (2) follows by applying Theorem 1 to (1). □

Theorem 5 implies that if we add \A and \z as primitive regexes with the semantics that

$$\langle u, v, s\rangle \models \backslash A \Leftrightarrow u = v = \epsilon \qquad \langle u, v, s\rangle \models \backslash z \Leftrightarrow v = s = \epsilon$$

then negative lookarounds are not needed, but can be represented with ~ and anchors using positive lookarounds. We then also define $\backslash A^{\mathbf{r}} \stackrel{\text{DEF}}{=} \backslash z$ and $\backslash z^{\mathbf{r}} \stackrel{\text{DEF}}{=} \backslash A$ and since anchors are primitive, their lookaround height is zero.

There is a practical application of Theorem 5 that already applies to standard regexes with lookarounds. For example, a concatenation of any two consecutive lookaheads as in $(?{=}L)\cdot(?{=}R)$ can be combined into an equivalent one lookahead $(?{=}L\cdot\top*\Cap R\cdot\top*)$, which means that $(?{=}L)\cdot(?!R)$ can also be combined into one lookahead by using Theorem 5. A single lookahead evaluation is more economical than two separate evaluations, and the regex $L\cdot\top* \Cap R\cdot\top*$ can potentially be further simplified, depending on L and R.

7 Related Work

Brzozowski derivatives were originally studied for *derives* by [19] where it is shown experimentally (in Table 1) that, for the *standard* fragment of regexes, good rewrite rules often provide *minimal* DFAs. An intuitive introduction to derivatives with a matching algorithm developed in Haskell also appears in [9].

The first *industrial* use of a derivative based regex matcher, as far as we know, is SRM [21], that has been deployed in the Microsoft *credential scanning* tool [13]. The work on SRM has subsequently been extended and modified to support *anchors* and integrated into .NET under the NonBacktracking flag [17] of RegexOptions, where, unlike SRM, the matching semantics is compatible with PCRE, i.e., compatible with the other *backtracking* based backends Compiled and None. The theory in [17] introduces *location* based derivatives for supporting *anchors*, and *lookarounds* are introduced informally as a generalization of anchors. *Lookaheads* have also been formalized by [16] with derivatives, but using a different underlying semantic domain based on commutative concatenation.

The recent work in [27] builds on [17] by extending the location based derivative theory with both lookarounds as well as intersection and complement, as denoted here by the class **RE**. This work validates experimentally that **RE** remains feasible to implement and offers succinctness that is also theoretically proved by [10], and practical expressivity, that is currently not supported in any state-of-the-art regular expression engine but could potentially be supported in future versions of such. The work in [27] abandons backtracking semantics in favor of *leftmost-longest* semantics that allows commutativity of $\Cup$ and $\Cap$, and implies that **RE** becomes an *effective Boolean algebra*. While the formalization of [27] was the primary focus here, our work here also provides a platform for the formalization of [17] (see Section 8).

The original work by [22, 23], introduced a match search algorithm for *lexing* via *partial derivatives* [1] by using the POSIX [20] semantics. This algorithm was subsequently generalized to Brzozowski derivatives, improved upon by [2, 3] and formalized in Isabelle/HOL. The work was recently expanded by [26] to allow bounded (finite) loops as well as character-sets, and uses an alternative definition of POSIX values together with an equivalence proof with the definition by [18]. The formalization in [3, 26] elegantly demonstrates the power of proof assistants by discovering several nontrivial gaps in the original correctness argument of [22]. Recently, [24] presented a functional recursive variant of the [22] algorithm formalized in Isabelle/HOL. A recent derivative based lexing algorithm called *Verbatim* (Verbatim++) [7, 8] has been formalized in Coq. Verbatim is related to but differs from [26] in the way POSIX tokens are processed and how simplifications are applied to derivatives.

Recall that the *leftmost-longest* match semantics guaranteed by $\textbf{llmatch}(R, w)$, as it is formulated in Theorem 4, is the following. This is our formal interpretation of POSIX semantics in terms of *spans*, when applied to the class **RE**:

$$\textbf{llmatch}(R, w) = w[i{:}j] \Rightarrow \begin{cases} i = \min\{\imath \mid \exists \jmath : w[\imath{:}\jmath] \models R\} \wedge \\ j = \max\{\jmath \mid w[i{:}\jmath] \models R\} \end{cases}$$

The fundamental difference between $\textbf{llmatch}(R, w)$ and the lexing algorithms mentioned above is that in the *first pass* $\textbf{llmatch}(R, w)$ traverses w *backwards* and only in the *second pass* forwards. This is not desired for lexing or even applicable to streaming input. Although the work in [27] discusses informally some implementation techniques used to swap the passes, those aspects remain unclear to us and fall outside our current formalization.

When comparing the formalization of $\textbf{llmatch}(R, w)$ with the formalization in [26], what stands out is that the ordering semantics of POSIX match values is much more intricate in [26, Fig.2]. We believe that one reason behind this is that detection of match begin location is more difficult when the first pass goes forward and the second pass goes backward from a nullable position as is the case in [26, Fig. 1] – where

reversal is not used, and is not directly applicable in the second pass according to [26, p.22]. Due to the extended class **RE**, and use of *spans*, a deeper analysis requires more research.

In prior works, to the best of our knowledge, regex matching with anchors and/or lookarounds have not been formalized in proof assistants. Intersection and complement have also been of moderate interest because matching engines at large have so far not supported them, despite the fact that their derivatives are a very natural extension of standard regexes [4]. Brzozowski derivatives of regular expressions extended with intersection and complement have recently also been formalized in Coq, where handling of complement required some additional encoding effort, increasing specification verbosity for defining negative propositions dually to the corresponding positive ones.[3]

8 Future Work

We discuss some ongoing and future work items where the current Lean formalization is used as the starting platform.

A direct application of the Lean formalization is to validate the correctness of the implementation in [27] through *fuzzing*, by using the fact that the definition of `llmatch` in Lean is *executable*, which enables comparing match results for a variety of different classes of regexes and inputs generated for those regexes.

The formalization of derivatives in Lean can also be extended to support backtracking semantics, where a core aspect is that alternation is *non-commutative*. The following brief summary is based on [17, Sec.4.2]. The derivative of a concatenation is there defined as follows

$$\mathbf{der}(L{\cdot}R, x) \overset{\text{DEF}}{=} \begin{cases} \mathbf{der}(L, x){\cdot}R, & \textbf{if } \neg\mathbf{null}(L, x); \\ \mathbf{der}(R, x) \cup \mathbf{der}(L, x){\cdot}R, & \textbf{else if } \mathbf{null!}(L, x); \\ \mathbf{der}(L, x){\cdot}R \cup \mathbf{der}(R, x), & \textbf{otherwise}. \end{cases}$$

where R has higher priority than L in the second case and where **null!** prioritizes nullability of the *left* alternative over the right one. E.g., while both $\varepsilon \cup \psi$ and $\psi \cup \varepsilon$ are always nullable, $\mathbf{null!}(\varepsilon \cup \psi, x) = \mathbf{true}$ but $\mathbf{null!}(\psi \cup \varepsilon, x) = \mathbf{false}$.

Intuitively, $\mathbf{null!}(L, x)$ means that, *in the location* x, L is equivalent with $(\varepsilon \cup L)$ which implies, by *left-distributivity* of concatenation over alternation in PCRE, that $(\varepsilon \cup L){\cdot}R$ is equivalent to $R \cup L{\cdot}R$, from which the second case of the definition above follows.

Formally $\mathbf{null!}(l \cup r, x) \overset{\text{DEF}}{=} \mathbf{null!}(l, x)$ and $\mathbf{null!}(\varepsilon, x) \overset{\text{DEF}}{=} \mathbf{true}$ and, for $\ell \in \mathcal{A}$, $\mathbf{null!}(\ell, x) \overset{\text{DEF}}{=} \mathbf{null}(\ell, x)$. Observe also that this extension has *no semantic impact* when $\cup$ is *commutative*.

Related *pruning* rules also need to be included into derivative processing that mimic how backtracking chooses a path by eliminating alternatives that backtracking would forget.

[3] Personal communication with Pit-Claudel Clément.

> *An outstanding semantic challenge is to combine these rules in a meaningful way with intersection and complement in the context of the full* **RE** *class with non-commutative alternation.*

We are also working on a *finiteness* proof in Lean of the state space $Q_{/\simeq}$ under a weak equivalence relation $\simeq$, with Q as the set of all reachable regexes from a given initial regex $R \in Q$ as the initial state, such that, for all $q \in Q$ and $x \in \mathbf{Loc}$, $\mathbf{der}(q, x) \in Q$. We believe that the only necessary laws for $\simeq$ are *associativity*, *idempotence*, and *deduplication* of $\cup$, the latter is in [17, Sec.5.3] implicitly present in the key rewrite rule AltUni. In other words, alternations are maintained in a canonical right-associative form and without any duplicate elements. This would formally validate finiteness of the state space required in the input-linear complexity result [17, Theorem 5.3], which does currently not follow directly from [4] due to alternation being non-commutative. A potential follow-up of finiteness of the state space would be to prove *decidability* of *nonemptiness* of $R \in \mathbf{RE}$ modulo $\mathcal{A}$, i.e., decidability of $\exists\, sp \in \mathbf{Span} : sp \models R$. Observe that, for all $\psi \in \alpha$, $\exists\, sp \in \mathbf{Span} : sp \models \psi \Leftrightarrow \exists\, c \in \sigma : c \in [\![\psi]\!] \Leftrightarrow [\![\psi]\!] \neq \emptyset$ so decidablility of $\mathcal{A}$ is a prerequisite in this case, while not needed in the formalization here, where membership $c \in [\![\psi]\!]$ is a much simpler decision problem.

Support for *captures* in the .NET nonbacktracking engine is perhaps the most difficult and complex aspect of the whole engine. This part is also not explained or formalized in any way in [17], other than providing a hint at a relationship to tagged automata [12]. Formalizing even the core aspect of the capture semantics and algorithms of the nonbacktracking engine in Lean, based on its open-source implementation [15], is therefore a major undertaking with many challenges.

9 Conclusions

We have formalized the core of a state-of-the-art industrial-strength regular expression engine supporting intersection, showing that the derivative-based implementation captures precisely the natural semantics of the extended regular expressions. The formalization was completed by an inexperienced Lean user in two months, resulting in around 1kLOC of Lean code. This shows that a formalization of the full engine including backtracking semantics is realistically within reach. The full formalization will not just increase our confidence in the correctness of the existing engine implementation. It will also enable ambitious optimizations that are too risky to deploy in a production engine without formal guarantees.

References

[1] Valentin Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science* 155 (1996), 291–319. https://doi.org/10.1007/3-540-59042-0_96

[2] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. 2016. POSIX Lexing with Derivatives of Regular Expressions. In *Archive of Formal Proofs*. http://www.isa-afp.org/entries/Posix-Lexing.shtml

[3] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. 2016. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Interactive Theorem Proving (LNCS, Vol. 9807)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer, 69–86. https://doi.org/10.1007/978-3-319-43144-4_5

[4] Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *JACM* 11 (1964), 481–494. https://doi.org/10.1145/321239.321249

[5] Loris D'Antoni and Margus Veanes. 2021. Automata Modulo Theories. *Commun. ACM* 64, 5 (May 2021), 86–95. https://doi.org/10.1145/3419404

[6] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 34:1–34:29. https://doi.org/10.1145/3110278

[7] Derek Egolf, Sam Lasser, and Kathleen Fisher. 2021. Verbatim: A Verified Lexer Generator. In *2021 IEEE Security and Privacy Workshops (SPW)*. 92–100. https://doi.org/10.1109/SPW53761.2021.00022

[8] Derek Egolf, Sam Lasser, and Kathleen Fisher. 2022. Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022)*. ACM, 27–39. https://doi.org/10.1145/3497775.3503694

[9] Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A Play on Regular Expressions: Functional Pearl. *SIGPLAN Not.* 45, 9 (2010), 357–368. https://doi.org/10.1145/1863543.1863594

[10] Wouter Gelade and Frank Neven. 2012. Succinctness of the Complement and Intersection of Regular Expressions. *ACM Trans. Comput. Logic* 13, 1, Article 4 (jan 2012), 19 pages. https://doi.org/10.1145/2071368.2071372

[11] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554. https://doi.org/10.1017/S0956796897002864

[12] V. Laurikari. 2000. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *7th International Symposium on String Processing and Information Retrieval*. 181–187. https://doi.org/10.1109/SPIRE.2000.878194

[13] Microsoft. 2021. CredScan. https://secdevtools.azurewebsites.net/helpcredscan.html.

[14] Microsoft. 2021. Regular Expression Language - Quick Reference. https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference.

[15] Microsoft. 2022. .NET Regular Expressions. https://github.com/dotnet/runtime/tree/main/src/libraries/System.Text.RegularExpressions.

[16] Takayuki Miyazaki and Yasuhiko Minamide. 2019. Derivatives of Regular Expressions with Lookahead. *J. Inf. Process.* 27 (2019), 422–430. https://doi.org/10.2197/ipsjjip.27.422

[17] 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. In *PLDI '23: 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation, Florida, USA, June 17-21, 2023*, Nate Foster et al. (Ed.). ACM, 1–2. https://doi.org/10.1145/3591262

[18] Satoshi Okui and Taro Suzuki. 2010. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. In *Implementation and Application of Automata (LNCS, Vol. 6482)*, Michael Domaratzki and Kai Salomaa (Eds.). Springer, 231–240. https://doi.org/10.1007/978-3-642-18098-9_25

[19] Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression Derivatives Re-examined. *J. Funct. Program.* 19, 2 (2009), 173–190. https://doi.org/10.1017/S0956796808007090

[20] POSIX. 2008. IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (2008), 1–3874. https://doi.org/10.1109/IEEESTD.2008.4694976

[21] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS, Vol. 11427)*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer, 372–378. https://doi.org/10.1007/978-3-030-17462-0_24

[22] Martin Sulzmann and Kenny Zhuo Ming Lu. 2012. Regular Expression Sub-Matching Using Partial Derivatives. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP'12)*. ACM, New York, NY, USA, 79–90. https://doi.org/10.1145/2370776.2370788

[23] Martin Sulzmann and Kenny Zhuo Ming Lu. 2014. POSIX regular expression parsing with derivatives. In *FLOPS 2014 (LNCS, Vol. 8475)*, M. Codish and E. Sumii (Eds.). Springer, 203–220. https://doi.org/10.1007/978-3-319-43144-4_5

[24] Chengsong Tan and Christian Urban. 2023. POSIX Lexing with Bitcoded Derivatives. In *14th International Conference on Interactive Theorem Proving (LIPICS, 26)*, A. Naumowicz and R. Thiemann (Eds.). Dagstuhl Publishing, 26:1–26:18. https://doi.org/10.4230/LIPIcs.ITP.2023.27

[25] Stephen Toub. 2018. All About Span: Exploring a New .NET Mainstay. *MSDN Magazine* 33, 1 (2018). https://learn.microsoft.com/en-us/archive/msdn-magazine/2018/january/csharp-all-about-span-exploring-a-new-net-mainstay

[26] Christian Urban. 2023. POSIX Lexing with Derivatives of Regular Expressions. *Journal of Automated Reasoning* 67 (July 2023), 1–24. https://doi.org/10.1007/s10817-023-09667-1

[27] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2023. Derivative Based Extended Regular Expression Matching Supporting Intersection, Complement and Lookarounds. In *arXiv*. https://doi.org/10.48550/arXiv.2309.14401

[28] Ekaterina Zhuchko. 2023. Lean sources for this paper. https://github.com/ezhuchko/extended-regexes

Received 2023-09-19; accepted 2023-11-25