Finiteness of Symbolic Derivatives in Lean

Ekaterina Zhuchko
Tallinn University of Technology, Estonia

Hendrik Maarand
Tallinn University of Technology, Estonia

Margus Veanes
Microsoft Research, USA

Gabriel Ebner
Microsoft Research, USA

Abstract

Brzozowski proved that the set of derivatives of any regular expression is finite modulo associativity, idempotence and, notably, commutativity of the union operator. We extend this result to the case of symbolic location based derivatives, for which we prove finiteness of the state space by quotienting only by associativity, deduplication and idempotence (ADI); the fact that we don't use commutativity allows for this result to carry over to the derivative based backtracking (PCRE) match semantics, where the union operator is noncommutative. Furthermore, we consider regular expressions extended with lookarounds, intersection, and negation. We also show that our method for proving finiteness allows us to include certain simplification rules in the derivative operation while preserving finiteness. The finiteness proof is constructive: given an expression R, we construct a finite set that is an overapproximation (modulo ADI) of the set of derivatives of R. We reuse some of the infrastructure provided in previous formalization efforts for regular expressions in Lean 4, showing the flexibility and reusability of the framework.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Lean, regular languages, lookarounds, derivatives, finiteness

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.16

Funding E. Zhuchko and H. Maarand were supported by the Estonian Research Council grant PRG1210 (*Automata in Learning, Interaction and Concurrency*).

1 Introduction

Regular expression derivatives, first introduced by Brzozowski [6], provide a powerful technique to lazily convert regular expressions into finite state automata. A *finite* automaton, by definition, has a finite number of states. Thus, the set of iterated derivatives of an expression (which correspond to states) must be finite. Brzozowski showed that, under a suitable quotienting relation, the number of derivatives of any expression is always finite. Remarkably, to obtain finiteness it is sufficient to quotient by *associativity*, *commutativity* and *idempotence*, or *ACI*, of union (alternation). In this paper, we extend this framework to a broader class of regular expressions that includes *lookarounds*, and where union is not necessarily commutative. This noncommutativity reflects the behavior of many modern regex engines – including those in Python, Java, and JavaScript – which use a *backtracking* matching strategy. In such engines, alternation is evaluated sequentially, meaning that the order affects match selection.

Moreover, we consider *symbolic* regular expressions. Many practical applications need large or even infinite alphabets where predicates of an *effective Boolean algebra* (EBA), rather than individual characters, are used in regular expressions. A typical example is the predicate or character class `\w` which denotes the set of all Unicode word-letters. Furthermore,

the alphabet algebra can even be *undecidable* or *semidecidable*, where one cannot reliably compute *satisfiable* Boolean combinations of predicates, so-called *minterms* [10] – implying that there is no guaranteed method for computing a finite alphabet, in which case the problem does not straightforwardly reduce to the classical case.

The main result of this paper, the proof of finiteness, is needed to formalize termination proofs of various algorithms that incrementally unfold symbolic extended regular expressions into symbolic automata, such as: formalizing and generalizing [24, Theorem 7.1] to the case of semidecidable alphabet theories; formalizing correctness of *MatchEnd* in [19, Theorem 5.3] where alternation is noncommutative; formalizing termination of *LLMatch* [29, Theorem 4]; and formalizing ω-regularity modulo theories of *RLTL* [30, Theorem 6] that uses symbolic derivatives of extended regular expressions modulo alphabet algebras that are not required to be decidable.

Contribution

We develop a *formalized proof in Lean*[1] of finiteness of the set of all iterated derivatives in the class $\mathbf{ERE}_{\leq}$ of regular expressions that allow *intersection*, *complement* and *lookarounds*, modulo any *effective Boolean algebra* (EBA) over characters. A key insight of the proof is that in the equivalence relation of regular expressions we can replace *commutativity* of alternation with *deduplication*, which maintains the order of alternations as required in the backtracking semantics.

We first lift the technique of *transition regexes* from [24] and formalize it in Lean through *symbolic location derivatives* where the symbolic location derivative of an expression is a *transition tree*. This way we abstract away from *concrete* locations and can take advantage of the *symbolic* representation as transition trees. In this context, the set of iterated (concrete) derivatives of R is a subset of the set of leaves of (the transition trees of) all iterated symbolic location derivatives of R. This is because transition trees, due to their symbolic representation, may contain unreachable leaves. Our approach to proving finiteness is to first define an explicit procedure that, given a regular expression R, constructs a (finite) list of regular expressions. We then show that any leaf, reachable or unreachable, of an iterated symbolic location derivative of R is contained in this list (modulo certain equations). We also show how this overapproximation can be used to include certain simplifications in the derivative operation while preserving finiteness.

We use the formalization of the core theory of location derivatives of $\mathbf{ERE}_{\leq}$ in [32] to show that symbolic location derivatives indeed preserve the match semantics. In doing so, we work with alternation up to *associativity*, *deduplication* and *idempotence* (*ADI*) and make use of several standard Lean libraries. Although our ultimate goal is practical efficiency, the finiteness proof presented in this paper prioritizes simplicity and ease of formalization while keeping the set of quotienting operations (relatively) minimal.

Synopsis

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 provides the preliminaries used in the rest of the paper, including the key results and definitions needed from [32]. Section 4 introduces the Lean theory of symbolic location derivatives that builds on transition regexes from [24]. Section 5 develops the Lean formalization of the

[1] See [31] for the complete self-contained Lean formalization.

finiteness result including the main theorem, `finiteness`. Finally, Section 6 discusses future work.

2 Related Work

Location based derivatives were developed and implemented as a new backend in the .NET platform [19]. The finiteness of the state space produced by the rewrite rules in [19] is not directly implied by Brzozowski's original proof, since in this setting the union operator is noncommutative. This paper fills the gap by considering deduplication, which is present in [19] with the key rule AltUni. To remove duplicates, the AltUni rule uses deduplication and the fact that the union is unital with respect to the $\bot$ element. Location based derivatives were further extended in [29] to support intersection and complement operations, while shifting the match semantics from PCRE to POSIX, where commutativity is permissible. The correctness of the matching algorithm was formally verified in Lean in [32]. The finiteness result in this paper applies directly to both works, as our proof – though based on deduplication – naturally extends to settings where union is commutative.

Symbolic Boolean derivatives and their underlying representation of transition regexes were first introduced in [24], and they generalize the definitions of derivatives in [6] and [2]. The formalism of symbolic derivatives enables more efficient decision procedures for solving extended regular expression constraints. This idea was later generalized to the framework of transition terms in [30].

There are several formally-verified decision procedures based on regular expression derivatives. Coquand and Siles [9] implement a decision procedure for regular language equivalence based on Brzozowski derivatives, and highlight how the key technical difficulty in the formalization work is proving the finiteness of the state space using Brzozowski's original proof of finiteness, which is done using Bar induction. Similarly, Traytel and Nipkow [20] present a framework for regular expression equivalence which they instantiate with several procedures including Brzozowski derivatives; the authors prove finiteness of the standard fragment of regular expressions and formalize their results in the Isabelle/HOL proof assistant. Given a regular expression R, both of the above mentioned works produce a finite list of normalized regular expressions which is closed under derivation. Our work similarly computes an explicit overapproximation but without relying on any normalization procedure.

In terms of relations used, authors of [9] prove finiteness up to language equivalence and remark that it seems possible to prove finiteness up to ACI equivalence only with the extra condition that the derivative function respects ACI equivalence of regular expressions. Authors of [20] also refer to this, and show that the derivative function respects equivalence. Interestingly, in our work the derivative function actually violates ACI preservation (see Section 4.3), but it does not affect the finiteness proof. As noted in [9], the finiteness proof does not depend on the alphabet being finite: our work similarly highlights this since our underlying symbolic representation of transition regexes does not rely on concrete characters and supports infinite alphabets. Moreover, our work highlights that the set of predicates from the underlying EBA may also be infinite.

Several formalizations of Antimirov's partial derivatives have been developed, including those by [18], [14] and [20]. Antimirov's definition simplifies the finiteness proof, as discussed in Section 5.3. However, extending it to support extended operators is nontrivial [7]. In contrast, the Brzozowski derivative approach naturally handles extended operators.

Derivatives have also been used to describe match search algorithms for *lexing* [25, 26] that have subsequently been improved upon and formalized in Isabelle/HOL by [3]. The

match semantics in this setting is POSIX [21] style. The work was recently expanded by [28] to allow bounded loops as well as *character-sets*; the latter extension is captured in our work by the use of *effective Boolean algebras* $\boldsymbol{\alpha}$, which allow us to formalize regular expressions modulo predicates of $\boldsymbol{\alpha}$. Recently, [27] presented a functional recursive variant of the [25] algorithm, also formalized in Isabelle/HOL. A recent derivative based lexing algorithm [11, 12] has been formalized in Coq, which differs from [28] in the way POSIX tokens are processed, as well as how simplifications are applied to derivatives.

Miyazaki and Minamide [17] introduced derivatives for regular expressions with lookaheads. Similarly, Berglund et al. [5] proposed a matching construction using automata with ε-transitions to support lookaheads. However, neither approach trivially extends to lookbehinds due to the inherent asymmetry between forward and backward matching. More recently, several works [15, 4] have proposed linear-time matching algorithms that support both lookaheads and lookbehinds through the use of oracle NFAs.

3 Preliminaries

Words, locations and spans

Let $\boldsymbol{\sigma}$ be an *alphabet* type, $\boldsymbol{\sigma}$ may denote an infinite set. A *word* or *string* over $\boldsymbol{\sigma}$ is represented by a list of elements of type $\boldsymbol{\sigma}$, that is denoted by $\boldsymbol{\sigma}^*$ and represented in Lean as `List` σ. We let ε or [] denote the empty word. Concatenation of $u, v \in \boldsymbol{\sigma}^*$ is formally denoted by $u +\!\!+ v$. We also write $a :: v$ for prepending $a \in \boldsymbol{\sigma}$ to $v \in \boldsymbol{\sigma}^*$ (the cons operation on lists). Informally, we write the juxtaposition uv and av in both cases.

A *location* describes a position in a string and is represented as a pair of words in $\mathbf{Loc} \stackrel{\text{DEF}}{=} \boldsymbol{\sigma}^* \times \boldsymbol{\sigma}^*$. Let $w \in \boldsymbol{\sigma}^*$, a *location in* w is a pair $\langle u^{\mathbf{r}}, v\rangle$, analogous to a list *zipper* [13]. A location $\langle \varepsilon, u\rangle$ is *initial* and a location $\langle u, \varepsilon\rangle$ is *final*. For a non-final location $x = \langle u, a :: v\rangle$, we write $(x.2)_0$ to refer to the current symbol a. 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 $\langle u, \varepsilon\rangle + 1 \stackrel{\text{DEF}}{=} \langle u, \varepsilon\rangle$.

A *span* is defined as $\mathbf{Span} \stackrel{\text{DEF}}{=} \boldsymbol{\sigma}^* \times \boldsymbol{\sigma}^* \times \boldsymbol{\sigma}^*$, i.e., a span is a triple of words that we denote by $\langle s^{\mathbf{r}}, u, v\rangle$ with $s, u, v \in \boldsymbol{\sigma}^*$ and where u represents the matching section of the string, and s and v represent the left and right context, respectively. A span can also be viewed as two locations x and y, each pointing to the beginning and end of the matching section in a string, we refer to this as $span(x, y)$.

Effective Boolean algebras

An *Effective Boolean algebra (EBA)* over an element universe $\boldsymbol{\sigma}$ [10] is a tuple $(\boldsymbol{\sigma}, \boldsymbol{\alpha}, \vDash, \bot, \top, \sqcup, \sqcap, ^{\mathbf{c}})$ where $\boldsymbol{\alpha}$ is a set of *predicates* that is closed under the Boolean connectives and contains $\bot$ and $\top$. For $a \in \boldsymbol{\sigma}$ and $\phi \in \boldsymbol{\alpha}$ the *models* relation $a \vDash \phi$ obeys classical Tarski laws. Let $[\![\phi]\!] \stackrel{\text{DEF}}{=} \{a \in \boldsymbol{\sigma} \mid a \vDash \phi\}$. Then $[\![\bot]\!] = \emptyset$ and $[\![\top]\!] = \boldsymbol{\sigma}$. If $[\![\phi]\!] \neq \emptyset$ then ϕ is *satisfiable*. We require all the connectives to be *computable* and $\vDash$ to be *decidable*. In Lean, we model EBAs as a type class of the type $\boldsymbol{\alpha}$ where $a \vDash \phi$ is represented by `denote` ϕ a. Here we refer to $\boldsymbol{\alpha}$ also as the EBA and say that $\boldsymbol{\alpha}$ is *decidable* if satisfiability in $\boldsymbol{\alpha}$ is decidable.

An alternative perspective is that an EBA consists of two Boolean algebras and a homomorphism between them. There is a Boolean algebra of syntax, corresponding to $\boldsymbol{\alpha}$ in the definition above, for representing predicates (e.g., in regular expressions) and a Boolean algebra of subsets of $\boldsymbol{\sigma}$. The homomorphism $[\![.]\!] : \boldsymbol{\alpha} \to (\boldsymbol{\sigma} \to \mathbb{B})$ interprets each predicate $\psi \in \boldsymbol{\alpha}$ as a subset of $\boldsymbol{\sigma}$ (where $\mathbb{B}$ denotes the set of Booleans).

Regular expressions with lookarounds

Here we formally define extended regular expressions with *lookarounds* modulo an EBA $(\boldsymbol{\sigma}, \boldsymbol{\alpha}, \vDash, \bot, \top, \sqcup, \sqcap, ^{\mathbf{c}})$. The class $\mathbf{ERE}_{\leq}$ of regular expressions, or *regexes*, is here given by the following abstract grammar

$$R \quad ::= \quad \psi \mid \varepsilon \mid R_1 \Cup R_2 \mid R_1 \Cap R_2 \mid R_1 \cdot R_2 \mid R* \mid \sim R \mid \texttt{(?=}R\texttt{)} \mid \texttt{(?<=}R\texttt{)} \mid \texttt{(?!}R\texttt{)} \mid \texttt{(?<!}R\texttt{)}$$

where $\psi \in \boldsymbol{\alpha}$ and $R, R_1, R_2 \in \mathbf{ERE}_{\leq}$.

The operators from *alternation* (or *union*) $\Cup$ to *complement* $\sim$ appear in the order of precedence with complement having the highest precedence. There are four *lookaround* expressions: $\texttt{(?=}R\texttt{)}$ is *lookahead*, $\texttt{(?<=}R\texttt{)}$ is *lookbehind*, $\texttt{(?!}R\texttt{)}$ is *negative lookahead*, and $\texttt{(?<!}R\texttt{)}$ is *negative lookbehind*, where R is the *body* of the lookaround. We refer to these collectively as $\mathbf{LA}$. The remaining operators are the *empty-word regex* ε, *intersection* $\Cap$, *concatenation* $\cdot$, and *Kleene star* $*$. Concatenation $\cdot$ is often implicit by using juxtaposition when this is unambiguous. The regex matching *nothing* is the predicate $\bot$ in $\boldsymbol{\alpha}$.

The *lookaround height* of $R \in \mathbf{ERE}_{\leq}$, $\mathit{lookHeight}(R)$, is defined as the nesting level of lookarounds in R. If R does not contain any lookarounds then $\mathit{lookHeight}(R) \stackrel{\text{DEF}}{=} 0$, otherwise $\mathit{lookHeight}(R) \stackrel{\text{DEF}}{=} \mathit{lookHeight}(L) + 1$ where L is the body of the lookaround in R with the largest height.

4 Symbolic location derivatives

The intuition behind using locations and spans is that matching with lookarounds is context dependent. Classically, nullability of a regex R is defined as a static property that holds when the language of R matches the empty word. However, in the presence of lookarounds, nullability is a dynamic property that depends on the location in the string. We will make use of the key definitions of $\mathbf{der}(R, x)$ and $\mathbf{null}(R, x)$ from [32] where x is a location. These definitions are used without modification in our setting. In particular, $\mathbf{null}(R, x)$ plays a central role in defining the semantics of transition regexes.

In this section we formally define the framework of *symbolic location based derivatives*. The definitions build on transition regexes $\mathbf{TR}$ from [24] and use them for the class $\mathbf{ERE}_{\leq}$ of regexes with lookarounds and location based derivatives given in [29]. We highlight two key differences from [24]: the first is that we use *nullability of regexes (with lookaheads)* as conditions in $\mathbf{TR}$, rather than membership in predicates. The second is that we lift all operations on $\mathbf{ERE}_{\leq}$ to $\mathbf{TR}$ and avoid explicit lifting rules as in [24, Section 4.1]; the latter choice makes sense in the SMT context where ITE expressions exist natively.

4.1 Transition regexes revisited

Transition regexes are the underlying representation of symbolic derivatives. The abstract grammar of $f, g \in \mathbf{TR}$ is as follows, for $L, R \in \mathbf{ERE}_{\leq}$:

$$f \quad ::= \quad L \quad \mid \quad (R, f, g)$$

A transition regex is a *binary classification tree*: it is either a *leaf* L or an *ITE* term (R, f, g) with *condition* R, *then-case* f and *else-case* g. The key idea is that the regular expression R in the condition acts as a symbolic predicate: the ITE evaluates to f if R is nullable at the current location, and to g otherwise.

All binary operations $\diamond : \mathbf{ERE}_{\leq} \times \mathbf{ERE}_{\leq} \to \mathbf{ERE}_{\leq}$ and unary operations $\blacklozenge : \mathbf{ERE}_{\leq} \to \mathbf{ERE}_{\leq}$ are lifted to $\mathbf{TR}$ as follows, let $f, g, h \in \mathbf{TR}$ and $L, R \in \mathbf{ERE}_{\leq}$:

$$(R, f, g) \diamond h \overset{\text{DEF}}{=} (R, f \diamond h, g \diamond h) \quad L \diamond (R, f, g) \overset{\text{DEF}}{=} (R, L \diamond f, L \diamond g) \quad \blacklozenge(R, f, g) \overset{\text{DEF}}{=} (R, \blacklozenge f, \blacklozenge g)$$

The *symbolic location derivative* $\delta(R)$ of $R \in \mathbf{ERE}_{\leq}$ is defined as a transition regex in $\mathbf{TR}$ which, intuitively, represents a partial evaluation of $\mathbf{der}(R, x)$ with respect to R whose nullability tests have been postponed. Let $\ell \in \mathbf{LA}$, $\psi \in \boldsymbol{\alpha}$:

$$\begin{aligned}
\delta(\varepsilon) &\overset{\text{DEF}}{=} \bot & \delta(\ell) &\overset{\text{DEF}}{=} \bot \\
\delta(\psi) &\overset{\text{DEF}}{=} (\texttt{(?=}\psi\texttt{)}, \varepsilon, \bot) & \delta(L{\cdot}R) &\overset{\text{DEF}}{=} (L, \delta(L){\cdot}R \Cup \delta(R), \delta(L){\cdot}R) \\
\delta(L \Cap R) &\overset{\text{DEF}}{=} \delta(L) \Cap \delta(R) & \delta(\sim R) &\overset{\text{DEF}}{=} \sim\delta(R) \\
\delta(L \Cup R) &\overset{\text{DEF}}{=} \delta(L) \Cup \delta(R) & \delta(R*) &\overset{\text{DEF}}{=} \delta(R) \cdot R*
\end{aligned}$$

where $\Cup$, $\Cap$, $\cdot$ and $\sim$ are lifted to $\mathbf{TR}$. We simplify $\delta(\bot) \overset{\text{DEF}}{=} \bot$.

4.2 Semantics of transition regexes

A transition regex encodes a function from locations (in the input string) to regular expressions. That is, given $f \in \mathbf{TR}$ and $x \in \mathbf{Loc}$, we define $f[x] \in \mathbf{ERE}_{\leq}$ as the *evaluation of f at x*. Let $L, R \in \mathbf{ERE}_{\leq}$ and $f, g \in \mathbf{TR}$:

$$L[x] \overset{\text{DEF}}{=} L \quad (R, f, g)[x] \overset{\text{DEF}}{=} \begin{cases} f[x], & \textbf{if } \mathbf{null}(R, x); \\ g[x], & \textbf{otherwise}. \end{cases}$$

While the evaluation function of transition regexes is well-defined for all locations, transition regexes are used to represent functions from nonfinal locations $\mathbf{Loc}^+$ to $\mathbf{ERE}_{\leq}$. For example, the derivative $\delta(\psi)$ of a predicate ψ is the transition regex $\texttt{(?=}\psi\texttt{)}, \varepsilon, \bot)$ using the *lookahead* of ψ as its condition. Thus, for all nonfinal locations $x \in \mathbf{Loc}^+$,

$$\delta(\psi)[x] = (\texttt{(?=}\psi\texttt{)}, \varepsilon, \bot)[x] = \begin{cases} \varepsilon, & \textbf{if } (x.2)_0 \vDash \psi; \\ \bot, & \textbf{otherwise}. \end{cases}$$

because, using that x is nonfinal,

$$\begin{aligned}
\mathbf{null}(\texttt{(?=}\psi\texttt{)}, x) &\Leftrightarrow \mathit{span}(x, x) \models \texttt{(?=}\psi\texttt{)} \\
&\Leftrightarrow \exists\, y : \mathit{span}(x, y) \models \psi \;\Leftrightarrow \mathit{span}(x, x+1) \models \psi \;\Leftrightarrow (x.2)_0 \vDash \psi
\end{aligned}$$

The following *lifting lemma* of transition regexes is used frequently in many contexts where $\diamond$ and $\blacklozenge$ are lifted binary and unary operations.

▶ **Lemma 1.** $\forall f, g \in \mathbf{TR}, x \in \mathbf{Loc}$:

1. $(f \diamond g)[x] = f[x] \diamond g[x]$
2. $(\blacklozenge f)[x] = \blacklozenge(f[x])$

The above lemma justifies certain useful transformations. For example, we can propagate complement into the leaves of a transition regex *without affecting its conditions*. This is the justification for the last step in the following equations

$$\delta(\sim\psi) = \sim\delta(\psi) = \sim(\texttt{(?=}\psi\texttt{)}, \varepsilon, \bot) = (\texttt{(?=}\psi\texttt{)}, \sim\varepsilon, \sim\bot)$$

where $\psi \in \boldsymbol{\alpha}$, $\delta(\sim\varepsilon) = \sim\bot$ and $\delta(\sim\bot) = \sim\bot$. In this case the resulting transition terms can directly be viewed as the *Symbolic Finite Automaton* [10] given in Figure 1 where the states $\sim\bot$ and $\sim\psi$ are always nullable, hence unconditionally accepting, and $\sim\varepsilon$ is never nullable,

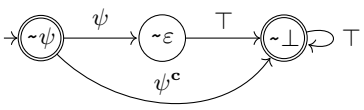

Figure 1 Symbolic Finite Automaton derived from ~ψ.

hence unconditionally non-accepting. The regex ~ψ is, without using ~, equivalent to the regex $\varepsilon \Cup \psi^{\mathbf{c}}{\cdot}\top{*} \Cup \psi{\cdot}\top{+}$ that is also clear from Figure 1. Propagation of ~ into leaves is a *superpower* of transition regexes that *enables lazy propagation of complement* as in [24] by *avoiding eager powerset* constructions for determinization.

Finally, we prove that the symbolic derivative (evaluated at a location x) is the same as the classical derivative:

▶ **Theorem 1.** $\forall\, x \in \mathbf{Loc}, R \in \mathbf{ERE}_{\leq} : \delta(R)[x] = \mathbf{der}(R, x)$

An immediate consequence of Theorem 1 is that the correctness of $\mathbf{der}(R, x)$ in [32] carries over to the symbolic definition. Observe also that if the lookaround height of L is 0 then nullability of L, $\mathit{Null}(L)$, is independent of locations, and thus

$$(L, f, g) \doteq \begin{cases} f, \textbf{ if } \mathit{Null}(L); \\ g, \textbf{ otherwise}. \end{cases}$$

4.3 Iterated derivatives

Recall that the symbolic location derivative $\delta(R)$ is a transition regex (i.e., a tree). Note that Theorem 1 implies that it is enough in the finiteness proof to care about the leaves of $\delta(R)$ as an overapproximation of $\{\mathbf{der}(R, x) \mid x \in \mathbf{Loc}^+\}$ (i.e., the set of all states reachable in one derivative step from R). To reason about iterated applications of the derivative, we repeatedly collect the leaves from the transition regex tree and apply δ to each of them. This process captures all states reachable through successive derivations and forms the basis of our finiteness argument.

The following function collects all of the leaves of a transition regex:

$$\mathbf{lvs}(r) \stackrel{\text{DEF}}{=} [r] \qquad \mathbf{lvs}((R, g, h)) \stackrel{\text{DEF}}{=} \mathbf{lvs}(g)\text{++}\mathbf{lvs}(h)$$

We lift unary operations $\blacklozenge$ and binary operations $\diamond$ on regexes to lists of regexes X, Y.

$$\blacklozenge X \stackrel{\text{DEF}}{=} [\blacklozenge R \mid R \in X] \qquad X \diamond Y \stackrel{\text{DEF}}{=} [L \diamond R \mid L \in X, R \in Y]$$

Lifting satisfies the following properties for any $f, g \in \mathbf{TR}$:

$$\mathbf{lvs}(f \diamond g) = \mathbf{lvs}(f) \diamond \mathbf{lvs}(g) \qquad \mathbf{lvs}(\blacklozenge f) = \blacklozenge \mathbf{lvs}(f)$$

The unary case is straightforward in Lean, but lifting binary operations requires the helper function `productWith`, which computes the Cartesian product of two lists and applies a given binary function to each pair. This function differs from `zipWith` which instead combines corresponding elements pointwise.

```
def productWith (op : α -> α -> α) (xs ys : List α) : List α :=
  map (uncurry op) (product xs ys)
#eval productWith (· + ·) [1,2] [3,4,5] -- [4, 5, 6, 5, 6, 7]
```

Next, we define a function that computes an overapproximation of the (immediate) derivatives of a regular expression R as $\mathbf{step}(R) \stackrel{\text{DEF}}{=} \mathbf{lvs}(\delta(R))$. The following lemmas describe how the step function is well-behaved with respect to the extended regular expression operations.

▶ **Lemma 2.** *Let $L, R \in \mathbf{ERE}_{\leq}$, then:*

$$\begin{aligned}
\mathbf{step}(L \Cap R) &= \mathbf{step}(L) \Cap \mathbf{step}(R) \\
\mathbf{step}(L \Cup R) &= \mathbf{step}(L) \Cup \mathbf{step}(R) \\
\mathbf{step}(\sim R) &= \sim\mathbf{step}(R) \\
\mathbf{step}(L \cdot R) &= \mathbf{step}(L) \cdot R \Cup \mathbf{step}(R) +\!\!+ \mathbf{step}(L) \cdot R \\
\mathbf{step}(R*) &= \mathbf{step}(R) \cdot R*
\end{aligned}$$

To reason about the set of all derivatives of an expression we need a way to iterate the symbolic location derivative. We define the function `steps` that systematically explores all possible derivatives of the initial regular expression by using a symbolic approach which at each step expands the current leaves.

```
def steps (r : RE α) : ℕ → List (RE α)
  | 0 => [r]
  | Nat.succ n => map step (steps r n) |> flatten
```

Thus `steps r n` is an overapproximation of derivatives of `r` wrt. words of length `n`. Here `|>` is (reverse) function application and `flatten` merges nested lists.

Recall that the symbolic location derivative δ can introduce unreachable leaves when constructing the *ITE* trees. The consequence of these unreachable leaves is that the `step` function does not respect idempotence.

```
theorem step_not_idem : ∃ (r : RE α), ¬ step (r ⋓ r) ⊆[ (· ≅ ·) ] step r
```

The following is a concrete example where the idempotence property is violated. Consider the predicate representing the character a, denoted by

```
def a : RE (BA Char) := Pred (atom 'a')
```

Taking a single step with `a` and `a ⋓ a` produces the following lists of expressions:

```
step a       = [ε, ⊥]
step (a ⋓ a) = [ε ⋓ ε,ε ⋓ ⊥,⊥ ⋓ ε,⊥ ⋓ ⊥]
```

Observe that the expressions `a` and `a ⋓ a` are similar but the lists of expressions that we get are not. In Figure 2, which is the transition regex δ `(a ⋓ a)`, one can clearly see that evaluating it with a concrete location can never reach the leaves $\varepsilon \Cup \bot$ and $\bot \Cup \varepsilon$ as the nullability conditions on those paths are contradictory. Consider the path from the leaf $\varepsilon \Cup \bot$ to the root: it contains one dashed line which represents the false branch of the nullability test and also one solid line which represents the true branch of the nullability test of the same expression so we reach a contradiction. We see that the unreachable leaves of symbolic location derivatives may be expressions that are not reachable even up to similarity. In a similar manner, the deduplication property is also violated.

For the finiteness proof it is sufficient to consider all possible states without actually pruning unreachable ones.

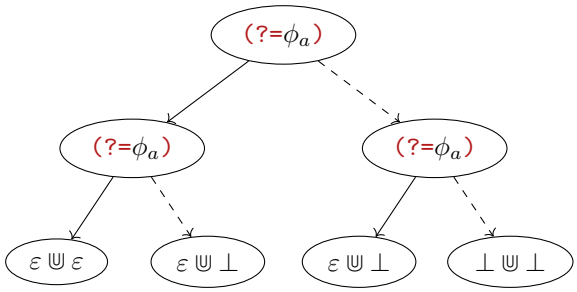

Figure 2 Example of a transition regex with unreachable leaves.

5 Finiteness of Derivation State Space

The main result of this paper is that the set of all possible derivatives up to similarity is finite. Intuitively, we would like to reason about the following set: $\mathcal{D}er(R) = \{der_w(R) \mid w \in \Sigma^*\}/_{\cong}$ for some notion of similarity $\cong$.

As an intermediate result, we associate with any regular expression R a finite set of expressions *pieces(R)*. We show that R itself can be represented as the sum of a subset of *pieces(R)*. Moreover, we show that any iterated derivative of R can be represented in the same way using pieces of R. For example, the pieces of the expression $a \Cup b \cdot c$ are $\{\bot, \varepsilon, a\} \cup \{\bot, \varepsilon, c, \bot \cdot c, \varepsilon \cdot c, b \cdot c\}$. The Brzozowski derivative of $a \Cup b \cdot c$ with respect to a is the expression $\varepsilon \Cup \bot \cdot c$, which can be represented as the sum of a subset of the pieces given above.

5.1 Brzozowski's proof

Let us first review the main idea of Brzozowski's proof of finiteness [6]. As main finiteness statement [6, Theorem 5.2], Brzozowski proves that any regular expression R has a finite number of *dissimilar* derivatives. The notion of *similarity* refers to the associativity, commutativity and idempotence of the union operator. Interestingly, the original proof by Brzozowski does not work without the additional equations $\varepsilon \cdot R \cong R$ and $\bot \cdot R \cong \bot$, as first pointed out by Salomaa [23]. Both Salomaa [23] and Aanderaa [1] relied on Brzozowski's proof of finiteness to establish the completeness of their systems. Later, Antimirov [2] proposed a modification to the original definition of derivatives, ensuring the finiteness result by Brzozowski holds just with the three original equations. This modified approach was subsequently widely adopted as the canonical definition [9, 18, 22].

Brzozowski first proves that every expression R has a finite number d_R of "*types*" of derivatives, where *types* are equivalence classes of expressions under language equivalence [6, Theorem 4.3(a)]. The proof is by induction and relies on arguments about the shapes of the resulting derivatives. He also gives a method [6, Theorem 4.3(b)] to construct the set of all d_R types of derivatives of R by lexicographically enumerating words and iteratively computing their derivatives until a fixpoint is reached under language equivalence. Brzozowski then proves that the number of dissimilar derivatives is also finite [6, Theorem 5.2] by showing that the process of constructing the set of all dissimilar derivatives must terminate. This argument relies on a careful analysis of the proof of [6, Theorem 4.3(a)].

As a comparison, we first note that Brzozowski's proof technique relies on the finiteness of the underlying alphabet, our strategy does not. The other key difference is that Brzozowski proves the existence of a bound on the number of derivatives whereas our proof simply

constructs an *overapproximation* of the set of derivatives. However, the underlying analysis when constructing the set of pieces of R is somewhat similar to the arguments in Brzozowski's proof of finiteness.

5.2 Similarity

The set of iterated derivatives is clearly not finite in general; to achieve finiteness, we define a relation which we call *similarity*, which contains the following cases for $L, R, S \in \mathbf{ERE}_{\leq}$:

$$(L \Cup R) \Cup S \cong L \Cup (R \Cup S) \qquad \text{(associativity)}$$
$$L \Cup (R \Cup L) \cong L \Cup R \qquad \text{(right deduplication)}$$
$$R \Cup R \cong R \qquad \text{(idempotence)}$$

In the Lean formalization, we define the similarity relation `Sim` as an inductive predicate that is almost a congruence relation; we do not require a congruence rule for lookarounds and Kleene star, as derivatives do not occur under lookarounds or star. Moreover, the congruence rule for concatenation keeps the second argument fixed, i.e., if $L \cong R$ then $L \cdot S \cong R \cdot S$.

Both deduplication and idempotence are used to ensure duplicate-free expressions. We conjecture that the finiteness proof would still hold without idempotence, at the cost of a larger state space. Instead of assuming commutativity, we adopt a right-biased deduplication rule: when eliminating duplicates in expressions like $L \Cup (R \Cup L)$, we retain the leftmost occurrence of L. This choice aligns with PCRE-style greedy semantics [19], where the first matching alternative has priority. Removing the earlier occurrence would break this behavior, hence the asymmetry. For the same reason, we cannot allow commutativity in the equivalence relation. Ignoring order—e.g., by treating alternation as a set operation—would further reduce the state space, but our finiteness result would still hold. While additional quotienting rules could coarsen the equivalence relation, they are not necessary for proving finiteness.

Since we work modulo an equivalence relation, it is natural for the theorems about regular expressions in this setting not to hold strictly, but *up to equivalence* under similarity. We therefore begin by setting up essential infrastructure for reasoning about lists of regular expressions modulo an equivalence relation. We define weaker "up to" definitions of membership of an element inside a list, inclusion, and equality (of lists as sets, i.e., mutual inclusions) w.r.t. any given relation `R`:

- The notation `x ∈[ R ] ys` expresses that there is an element in `ys` which is related to `x` by `R`;
- The notation `xs ⊆[ R ] ys` expresses that every element of `xs` is an element of `ys` up to equivalence under `R` (i.e., for all `x ∈ xs`, `x ∈[ R ] ys`);
- Finally, `xs =[ R ] ys` expresses that `xs` and `ys` are equal up to equivalence under `R`.

5.3 Pieces of regular expressions

In this section we define the function `pieces` which, given a regular expression R, computes a list of regular expressions. The expressions in this list can then be combined together with the union operator to reconstruct any derivative of R, as discussed in the beginning of Section 5.

To illustrate the underlying idea, we begin with the so-called partial derivatives which were introduced by Antimirov [2]. The main difference with Brzozowski derivatives is that, given an input regular expression and a character, a Brzozowski derivative is simply a single regular expression (naturally leading to a DFA construction) while there is a *set*

of Antimirov derivatives (naturally leading to an NFA construction). An advantage of Antimirov's approach is that the finiteness of partial derivatives is significantly easier to prove than the finiteness of Brzozowski derivatives.

Interestingly, Mirkin [16] and Antimirov [2] independently proposed a similar method for constructing an NFA from a regular expression. Later, Champarnaud and Ziadi [8] proved that these constructions are equivalent, unifying the idea and presenting the notion of a *support* of a regular expression. It is defined as the following function where $c \in \Sigma$, for some alphabet Σ and $\bot$ denotes the empty language:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{support}(\bot) & \stackrel{\text{DEF}}{=} & \emptyset & \mathit{support}(L \Cup R) & \stackrel{\text{DEF}}{=} & \mathit{support}(L) \cup \mathit{support}(R) \\ \mathit{support}(\varepsilon) & \stackrel{\text{DEF}}{=} & \emptyset & \mathit{support}(L{\cdot}R) & \stackrel{\text{DEF}}{=} & \mathit{support}(L) \odot R \cup \mathit{support}(R) \\ \mathit{support}(c) & \stackrel{\text{DEF}}{=} & \{\varepsilon\} & \mathit{support}(R*) & \stackrel{\text{DEF}}{=} & \mathit{support}(R) \odot R* \end{array}$$

In the definition above, the $\odot$ operation is simply concatenation lifted to set of regular expressions on the left and a single expression on the right.

It can be shown that all iterated Antimirov derivatives of R are contained in the following set: $\{R\} \cup \mathit{support}(R)$. It is easy to see that Antimirov derivatives are not of the form $e \Cup f$. In order to show finiteness of the set of Antimirov derivatives of R, it is sufficient to show that the $\mathit{support}(R)$ is finite. Note that it is not necessary to consider any similarity relation for this, as the set representation inherently accounts for ACI. The proof is by induction on R.

Our proof of finiteness follows a similar strategy, where we approximate the set of all iterated derivatives in a similar fashion to the *support* function above. Note that Brzozowski derivatives (and their finiteness) are not directly related to Antimirov derivatives, because additional equations would be needed to relate the two. In particular, one needs to include distributivity of concatenation over union in the similarity relation. For example, the Brzozowski derivative of an expression of the form $(e \Cup f) \cdot g$ leads to expressions of the form $(e' \Cup f') \cdot g \Cup g'$ (assuming that e or f is nullable), where e', f' and g' represent the derivatives of e, f and g respectively. For Antimirov derivatives, this leads to a set of expressions of the form $e' \cdot g$, $f' \cdot g$ or g' (again assuming that e or f is nullable).

Since we are working with Brzozowski-style derivatives, which do not compute sets of expressions, we can take the definition of *support* only as an inspiration. We need to tailor the definition of *pieces* to our setting which has extended regular expressions, deterministic derivatives and similarity consisting of associativity, deduplication and idempotence.

5.3.1 Definitions

We start by defining the function `toSum` which creates a union from a list of regular expressions. This function is used as a building block to combine the pieces of an expression together.

```
def toSum : List (RE α) → RE α
  | []  => Pred ⊥
  | [a] => a
  | a::b::bs => a ⋓ toSum (b::bs)
```

Note that in our setting, we never use `toSum` on empty lists, and we add an extra case for the singleton list which differentiates the one-element case from the empty list case: this is a non-trivial optimization, since we want to keep the number of equivalence laws for regular expressions to a minimum, and unnecessarily introducing $\bot$ would force us to assume that the union of regular expressions is unital (on at least one side) with respect to $\bot$.

We define the `neSubsets` function which computes all permutations of non-empty sublists:

```
def neSubsets (xs : List α) : List (List α) :=
  neSublists xs |> map permutations' |> join
```

The final ingredient necessary to define *pieces* is the following composition:

```
def toSumSubsets (xs : List (RE α)) : List (RE α) :=
  xs |> neSubsets |> map toSum
```

We split the definitions of `neSubsets` and `toSumSubsets` in order to modularly prove correctness and completeness of the first function, which does not fundamentally rely on properties of regular expressions. We introduce the notation $\oplus$ to represent `toSumSubsets`. The function $\oplus$`xs` computes all permutations of non-empty sublists of `xs`, where permutations are represented as sums of expressions. For example, $\oplus([a, b]) = [a, a \cup b, b \cup a, b]$. The need to consider all permutations of a list (of regular expressions) arises from the fact that we eliminate commutativity of union as part of the similarity relation. Otherwise, we could simply take all sublists without considering permutations e.g., $\oplus([a, b]) = [a, a \cup b, b]$. Crucially, we prove that the `toSumSubsets` function preserves the property of inclusion up-to similarity:

```
theorem toSumSubsets_monotone [DecidableEq α] {xs : List (RE α)} :
  xs ⊆[ (· ≅ ·) ] ys → ⊕xs ⊆[ (· ≅ ·) ] ⊕ys
```

The type class `DecidableEq` is required in some proofs involving the $\oplus$ operator as it can lead to simpler proofs in certain cases.

A characterizing property for the above function is the fact that any element of `toSumSubsets` can be represented as a non-empty subset of the original list.

```
theorem toSumSubsets_to_neSubset {xs : List (RE α)} :
  x ∈ ⊕xs → ∃ zs, zs ≠ [] ∧ x = toSum zs ∧ zs ⊆ xs
```

We now have all the ingredients to define the `pieces` function:

```
def pieces : RE α → List (RE α)
  | ε          => [ε, Pred ⊥]
  | ?= r       => [?= r, ε, Pred ⊥]
  | ?! r       => [?! r, ε, Pred ⊥]
  | ?<= r      => [?<= r, ε, Pred ⊥]
  | ?<! r      => [?<! r, ε, Pred ⊥]
  | Pred φ     => [Pred φ, ε, Pred ⊥]
  | l ⋓ r      => pieces l ++ pieces r
  | l ⋒ r      => productWith (· ⋒ ·) ⊕(pieces l) ⊕(pieces r)
  | ~ r        => map (~ ·) ⊕(pieces r)
  | l · r      => map (· · r) ⊕(pieces l) ++ pieces r
  | r *        => r* :: map (· · r*) ⊕(pieces r)
```

It turns out that for any expression `r`, `pieces r` contains the parts necessary to represent *any iterated derivative of* `r`. We highlight the case for concatenation; for example, the derivative of `(e ⋓ f) · g` takes one of two forms: `(e' ⋓ f') · g`, when `e` or `f` is nullable, or `g'` (which corresponds to an element in `pieces g`). We will prove later that an expression like `e' ⋓ f'` can be built from the pieces of `e ⋓ f`, by which we mean that `e' ⋓ f'` is equivalent to an element in $\oplus$`(pieces(e ⋓ f))`. Consequently, this ensures that `(e' ⋓ f') · g` can be built from the pieces of `(e ⋓ f) · g`.

As with the *support* function defined earlier, the union operator does not appear as the top-most operator in any individual pieces:

```
theorem topmost_not_union {r x y : RE α} : ¬ (x ⩏ y ∈ pieces r)
```

Using the definition of `pieces`, we can finally define our overapproximation of the derivatives of a regular expression `r`: `⊕(pieces r)`. This list contains all permutations of all non-empty subsets of pieces of `r` (where each subset is joined together by `toSum`). Next, we describe properties of `pieces` and ⊕ that are used for proving finiteness.

The `pieces` function satisfies the following reflexivity-like property:

```
theorem pieces_refl {r : RE α} :
  ∃ xs, xs ∈ neSublists (pieces r) ∧ toSum xs ≅ r
```

The following theorem establishes that similar expressions have the same pieces up-to similarity:

```
theorem pieces_sim [DecidableEq α] {f f' : RE α} (sim : f ≅ f') :
  pieces f =[ (· ≅ ·) ] pieces f'
```

The `pieces` function also satisfies the following transitivity-like property, in the sense that all pieces of a piece of `r` are already contained in `pieces r` (up-to similarity):

```
theorem pieces_trans [DecidableEq α] {e f g : RE α}
  e ∈ pieces f →
  f ∈ pieces g →
  e ∈[ (· ≅ ·) ] pieces g
```

To prove this form of transitivity, we make use of the properties `toSumSubsets_monotone` and `toSumSubset_to_neSubset`.

So far we have only considered *pieces* (i.e., elements of `pieces r`) but the set of all iterated derivatives of `r` is approximated by `⊕(pieces r)`. The reflexivity and transitivity properties also extend to `⊕(pieces r)`:

```
theorem toSumSubsets_pieces_refl {r : RE α} :
  r ∈[ (· ≅ ·) ] ⊕(pieces r)
theorem toSumSubsets_pieces_trans [DecidableEq α] {e f g : RE α} :
  e ∈[ (· ≅ ·) ] ⊕(pieces f) →
  f ∈[ (· ≅ ·) ] ⊕(pieces g) →
  e ∈[ (· ≅ ·) ] ⊕(pieces g)
```

5.4 Finiteness

From the previous section, we know that for every expression `r` there is a finite set `⊕(pieces r)`. Now we will show that every derivative of `r` is represented by an element in this set.

We first prove that every one-step derivative of `f` (given by the `step` function) is similar to a sum of pieces of `f`. The following theorem captures precisely this property:

```
theorem step_to_pieces {e f : RE α} (in_step : e ∈ step f) :
  ∃ xs, toSum xs ≅ e ∧ xs ∈ neSubsets (pieces f)
```

It follows that the set of one-step derivatives is contained in the overapproximation. This fact is captured by the following theorem:

```
theorem step_to_toSumSubsets_pieces {r : RE α} :
  step r ⊆[ (· ≅ ·) ] ⨁(pieces r)
```

To show that all derivatives are contained in the overapproximation `⨁(pieces r)`, it suffices to prove that any `n`-step derivative is in the overapproximation (up-to similarity).

```
theorem steps_to_toSumSubsets_pieces [DecidableEq α] {r : RE α} :
  steps r n ⊆[ (· ≅ ·) ] ⨁(pieces r)
```

The proof proceeds by induction on the number of derivative steps `n`. The base case follows from `toSumSubsets_pieces_refl`. For the inductive step, let `e` be any `(n - 1)`-step derivative. By the inductive hypothesis, we have that `e` is in the overapproximation of `r`, `⨁(pieces r)`. Then, we have that any single-step derivative `f` (of `e`) is in the overapproximation of `e`, `⨁(pieces e)`, by lemma `step_to_toSumSubsets_pieces`. The theorem is concluded by applying `toSumSubsets_pieces_trans` to show that `f` is contained in the overapproximation of `r`.

The main theorem asserts the existence of a list of regular expressions that contains the set of all iterated derivatives of a given regular expression `r` up-to similarity. We choose `⨁(pieces r)` as the overapproximation.

```
theorem finiteness [DecidableEq α] {r : RE α} :
    ∃ (xs : List (RE α)), ∀ {n : ℕ}, steps r n ⊆[ (· ≅ ·) ] xs :=
  ⟨⨁(pieces r), steps_to_toSumSubsets_pieces⟩
```

In our formalization, the state space captured by the `steps` function only accounts for the parts of the expression at lookaround depth 0 (i.e., not under a lookaround). This is because the derivative of a lookaround is always $\bot$ and the actual complexity of lookarounds lies in the nullability check (which is only done at matching time). A transition regex $\delta(R)$ encodes all possible transitions from R with respect to any location. For a location x, the concrete transition is into $\delta(R)[x]$ and the computation of this target state may involve nullability checks. If the nullability check is for a subexpression of R that is a lookaround, then this spawns an automaton for the lookaround body at location x.

Let us now informally consider the finiteness of the *global* state space for R, which includes the states induced by nullability checks of *lookarounds*. Finiteness follows by induction on lookaround height. For the base case $\mathit{lookHeight}(R) = 0$, i.e., when R contains no lookarounds, the global state space is the set of derivatives that we approximate with `⨁(pieces r)` (which is finite). For the inductive case $\mathit{lookHeight}(R) = n + 1$, we assume that the global state space is finite at lookaround height n. The set of top-level derivatives of R is finite by the `finiteness` theorem. If a top-level state is an expression that has a lookaround as a subexpression, then computing its successor may spawn an automaton for the body of that lookaround (which has lookaround height at most n). By inductive hypothesis, the automaton corresponding to an expression with lookaround height n has a finite *global* state space.

5.5 `pieces` as a closure operator

An alternative perspective on `pieces` is to view it as a closure operator when lifted to lists (sets) of expressions. This lifted function, `piecesS`, satisfies the standard properties of a closure operator: it is extensive, monotonic, and idempotent. In Lean, we define it as follows, where $\circ$ denotes function composition:

```
def piecesS (xs : List (RE α)) : List (RE α) :=
  map ((⊕ ·) ∘ (pieces ·)) xs |> flatten
```

```
theorem piecesS_extensive [DecidableEq α] {xs : List (RE α)} :
  xs ⊆[(· ≅ ·)] piecesS xs
theorem piecesS_monotone [DecidableEq α] {xs ys : List (RE α)} :
  xs ⊆[(· ≅ ·)] ys → piecesS xs ⊆[(· ≅ ·)] piecesS ys
theorem piecesS_idem {xs : List (RE α)} [DecidableEq α] :
  piecesS (piecesS xs) ⊆[(· ≅ ·)] piecesS xs
```

In order to prove finiteness in this setting, we show that

```
steps e n ⊆[(· ≅ ·)] piecesS [e]
```

This proceeds by induction on the number of steps `n`. The base case follows from extensivity, since `steps e 0` is just the singleton `[e]`. For the inductive case, we assume that

```
steps e n ⊆[(· ≅ ·)] piecesS [e]
```

From the theorem `step_to_pieces`, it follows that

```
steps e (n + 1) ⊆[(· ≅ ·)] piecesS (steps e n)
```

By applying monotonicity and idempotence on the inductive hypothesis, we get that

```
piecesS (steps e n) ⊆[(· ≅ ·)] piecesS [e]
```

Finally, by transitivity we can conclude that

```
steps e (n + 1) ⊆[(· ≅ ·)] piecesS [e]
```

In the reasoning above, exactly the three properties of `piecesS` being a closure operator are needed. We can now prove the main `finiteness` theorem with `piecesS [e]` as the overapproximation since `piecesS [e]` contains all derivatives of `e`.

5.6 On the computational complexity of the approximation

We briefly comment on the complexity aspects of our approximation for the derivatives of a regular expression. First of all, in no way is `pieces r` meant to be a (reasonably) precise approximation of the set of derivatives of `r`. It is just an upper bound of this set. The main feature is that `pieces` is a total function and thus `pieces r` is finite for any `r`.

Our definition for the overapproximation focuses on being easy to conceptualize, and is defined in terms of the composition of simpler components in order to make it more amenable to formal verification. In particular, the overapproximation is given in terms of the $\oplus$ function, which computes all non-empty subsets of permutations of a list, and the `pieces` function to actually traverse the regular expression. It can be shown that the computational complexity of $\oplus$ is already $\mathcal{O}(n!)$ in the size of the input list. The worst case of the `pieces` function materializes in the case for intersection, where one takes Cartesian products of the approximations of the two subexpressions.

5.7 Simplifications

We have shown that, for every `n`, every `n`-step derivative `r'` of `r` is made of pieces of `r`. By this we mean that every element `r'` of `steps n r` is ADI-equivalent to an element of $\oplus$`(pieces r)` (recall that $\oplus$`(pieces r)` contains every permutation of every subset of `pieces r` as a

sum). Finiteness of the set of derivatives of `r` (modulo ADI) follows from the fact that the set `pieces r` is finite. We now describe how to integrate simplifications into the derivative computation without sacrificing finiteness. The key idea is to ensure that simplifications do not introduce any new pieces.

Our first observation is that $\oplus$ is monotone: if `xs` $\subseteq$ `ys`, then $\oplus$`xs` $\subseteq$ $\oplus$`ys`. More specifically, if an expression `p'` is ADI-equivalent to an element of $\oplus$`(pieces p)`, then it is also ADI-equivalent to an element of $\oplus$`(pieces r)` provided that `pieces p` $\subseteq$ `pieces r`. This is the condition that determines allowed simplifications: `r` can be simplified to `p` if `p` does not introduce any new pieces (which is necessary for preserving finiteness). For us, a simplification is a function `RE` α `→ RE` α that is applied after every derivative step.

We incorporate such a simplification `f` by defining a modified `step` function which applies `f` to every element in the result of the original `step`.

```
def step' (f : RE α → RE α) (r : RE α) : List (RE α) := map f (step r)
```

We define allowed simplifications `f` to be those that satisfy the following predicate.

```
def NonIncreasing (f : RE α → RE α) : Prop := ∀ r, pieces (f r) ⊆ pieces r
```

Finally, we prove the following property which suffices to keep the rest of the finiteness proof almost the same as it was with just the function `step`. Every one-step derivative of `r`, after simplifications with a suitable `f`, is ADI-equivalent to an element of $\oplus$`(pieces r)`.

```
∀ f r, NonIncreasing f → step' f r ⊆[ (· ≅ ·) ] ⊕(pieces r)
```

Here we write `r` $\rightsquigarrow$ `s` to say that `r` simplifies to `s`. For example, it is easy to see from the definition of `pieces` that the simplification rule `r ⋓ s` $\rightsquigarrow$ `r` is nonincreasing since `pieces r` $\subseteq$ `pieces (r ⋓ s)`. It is not a valid simplification rule in the sense that it does not preserve the language. However, we can restrict ourselves to the cases where it does: `r ⋓ ⊥` $\rightsquigarrow$ `r` and `r ⋓ ~⊥` $\rightsquigarrow$ `~⊥`. It is also easy to see that associativity, deduplication and idempotence can also be made into allowed simplification rules.

The simplification rule `r · s` $\rightsquigarrow$ `s` is also nonincreasing and thus is allowed. Again, this is not a valid rule but we can restrict to the cases where it is: ε `· s` $\rightsquigarrow$ `s` and `r · ⊥` $\rightsquigarrow$ `⊥`. Note, however, that we are *not* allowed to simplify `r · s` to `r` as that would not be nonincreasing. The following simplification rules are allowed.

`⊥ ⋓ s` $\rightsquigarrow$ `s` `~⊥ ⋓ s` $\rightsquigarrow$ `~⊥` ε `· s` $\rightsquigarrow$ `s` `(r ⋓ s) · t` $\rightsquigarrow$ `r · t ⋓ s · t`
`r ⋓ ⊥` $\rightsquigarrow$ `r` `r ⋓ ~⊥` $\rightsquigarrow$ `~⊥` `r · ⊥` $\rightsquigarrow$ `⊥`

We can also compose any sequence of nonincreasing simplifications: the identity function is nonincreasing; if f and g are nonincreasing, then so is $g \circ f$. Also, nonincreasing simplifications satisfy certain congruences. If `f` is nonincreasing, then so are the following rules:

`r ⋓ s` $\rightsquigarrow$ `f r ⋓ f s` `r ⋒ s` $\rightsquigarrow$ `f r ⋒ f s` `~r` $\rightsquigarrow$ `~f r` `r·s` $\rightsquigarrow$ `f r·s`

Note how the last rule keeps `s` fixed.

Furthermore, as the allowed simplifications are determined by `pieces`, we could allow more simplifications by extending the set of pieces of an expression. For example, if every expression had $\bot$ as a piece, then $\bot$ `· s` $\rightsquigarrow$ $\bot$ would be allowed (or, we could ensure that `pieces r` is a subset of `pieces (r · s)` which would also allow `r ·` ε $\rightsquigarrow$ `r`). Similarly, more simplifications would be allowed if `pieces r` and `pieces s` were subsets of `pieces`

`(r ⫛ s)`. We suspect it would be relatively easy to extend `pieces` in this way so that the following simplifications are also allowed:

$$\texttt{\sim}\bot \mathbin{⫛} \texttt{s} \rightsquigarrow \texttt{s} \quad \bot \mathbin{⫛} \texttt{s} \rightsquigarrow \bot \quad \texttt{r} \cdot \varepsilon \rightsquigarrow \texttt{r} \quad \texttt{\sim\sim r} \rightsquigarrow \texttt{r}$$
$$\texttt{r} \mathbin{⫛} \texttt{\sim}\bot \rightsquigarrow \texttt{r} \quad \texttt{r} \mathbin{⫛} \bot \rightsquigarrow \bot \quad \bot \cdot \texttt{s} \rightsquigarrow \bot$$

6 Future work

A follow-up to the problem of proving finiteness of the state space is to formally show *decidability* of *nonemptiness* of $R \in \mathbf{ERE}_{\leq}$ modulo $\boldsymbol{\alpha}$, i.e., decidability of $\exists\, sp : sp \models R$. Observe that, for all $\psi \in \boldsymbol{\alpha}$, $[\![\psi]\!] \neq \emptyset \Leftrightarrow \exists\, sp : sp \models \psi$, so decidability of $\boldsymbol{\alpha}$ is a prerequisite in this case, while it is not needed in the formalization of finiteness of the state space. A related question is to formally study the size complexity of the state space with respect to the size of R under various restrictions, such as disallowing intersection and/or complement. The finiteness of the state space is also fundamental in being able to formally reason about complexity results of matching R, either for PCRE semantics, where union is noncommutative, or POSIX semantics.

We have described how the function `pieces` determines certain simplification rules (on regular expressions) that preserve finiteness of the set of derivatives. A question for further investigations is how far can this be pushed in the sense that what simplifications can be allowed and which cannot by varying the definition of `pieces`. A further distinction could be made by considering PCRE semantics instead where $X(Y \Cup Z)$ is not equivalent to $XY \Cup XZ$.

Finally, it would be interesting to see if it is possible to define a one-way derivative operation (that processes the input from left to right) for the class of expressions considered in this paper.

References

1 Stål Aanderaa. On the algebra of regular expressions. *Technical Report, Applied Mathematics, Harvard University*, 1965.

2 Valentin Antimirov. Partial derivatives of regular expressions and finite automata constructions. *Theoretical Computer Science*, 155:291–319, 1996. `doi:10.1007/3-540-59042-0\_96`.

3 Fahad Ausaf, Roy Dyckhoff, and Christian Urban. POSIX lexing with derivatives of regular expressions. In *Archive of Formal Proofs*, 2016. URL: `http://www.isa-afp.org/entries/Posix-Lexing.shtml`.

4 Aurèle Barrière and Clément Pit-Claudel. Linear matching of javascript regular expressions. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. `doi:10.1145/3656431`.

5 Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. Regular expressions with lookahead. *Journal of Universal Computer Science*, 27(4):324–340, 2021. `doi:10.3897/jucs.66330`.

6 Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. `doi:10.1145/321239.321249`.

7 Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. Partial derivatives of an extended regular expression. In *Language and Automata Theory and Applications, LATA 2011*, volume 6638 of *LNCS*, pages 179–191. Springer, 2011.

8 Jean-Marc Champarnaud and Djelloul Ziadi. From Mirkin's prebases to Antimirov's word partial derivatives. *Fundam. Inf.*, 45(3):195–205, January 2001.

9 Thierry Coquand and Vincent Siles. A decision procedure for regular expression equivalence in type theory. In *Certified Programs and Proofs*, pages 119–134. Springer Berlin Heidelberg, 2011.

10 Loris D'Antoni and Margus Veanes. Automata modulo theories. *Communications of the ACM*, 64(5):86–95, May 2021.

11 Derek Egolf, Sam Lasser, and Kathleen Fisher. Verbatim: A verified lexer generator. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 92–100, 2021. doi:10.1109/SPW53761.2021.00022.

12 Derek Egolf, Sam Lasser, and Kathleen Fisher. 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)*, page 27–39. ACM, 2022. doi:10.1145/3497775.3503694.

13 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. doi:10.1017/S0956796897002864.

14 Vladimir Komendantsky. Reflexive toolbox for regular expression matching: verification of functional programs in Coq+Ssreflect. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, page 61–70, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2103776.2103784.

15 Konstantinos Mamouras and Agnishom Chattopadhyay. Efficient matching of regular expressions with lookaround assertions. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632934.

16 B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Journal of Symbolic Logic*, 36(4):694–694, 1971. doi:10.2307/2272532.

17 Takayuki Miyazaki and Yasuhiko Minamide. Derivatives of regular expressions with lookahead. *J. Inf. Process.*, 27:422–430, 2019. doi:10.2197/ipsjjip.27.422.

18 Nelma Moreira, David Pereira, and Simão Melo de Sousa. Deciding regular expressions (in-)equivalence in Coq. In Wolfram Kahl and Timothy G. Griffin, editors, *Relational and Algebraic Methods in Computer Science*, pages 98–113, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

19 Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. Derivative based nonbacktracking real-world regex matching with backtracking semantics. In Nate Foster et al., editor, *PLDI '23: 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation, Florida, USA, June 17-21, 2023*, pages 1–2. ACM, 2023. doi:10.1145/3591262.

20 Tobias Nipkow and Dmitriy Traytel. Unified decision procedures for regular expression equivalence. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 450–466, Cham, 2014. Springer International Publishing.

21 POSIX. IEEE standard for information technology - portable operating system interface (POSIX(R)). *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages 1–3874, 2008. doi:10.1109/IEEESTD.2008.4694976.

22 Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. Symbolic regex matcher. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 11427 of *LNCS*, pages 372–378. Springer, 2019. doi:10.1007/978-3-030-17462-0_24.

23 Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, January 1966. doi:10.1145/321312.321326.

24 Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In *PLDI'21*, pages 620–635. ACM, 2021. doi:10.1145/3453483.3454066.

25 Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP'12)*, pages 79–90, New York, NY, USA, 2012. ACM. doi:10.1145/2370776.2370788.

26 Martin Sulzmann and Kenny Zhuo Ming Lu. Posix regular expression parsing with derivatives. In M. Codish and E. Sumii, editors, *FLOPS 2014*, volume 8475 of *LNCS*, pages 203–220. Springer, 2014. doi:10.1007/978-3-319-43144-4_5.

27 Chengsong Tan and Christian Urban. POSIX lexing with bitcoded derivatives. In A. Naumowicz and R. Thiemann, editors, *14th International Conference on Interactive Theorem Proving*, number 26 in LIPICS, pages 26:1–26:18. Dagstuhl Publishing, 2023.

28 Christian Urban. POSIX lexing with derivatives of regular expressions. *Journal of Automated Reasoning*, 67:1–24, July 2023. doi:10.1007/s10817-023-09667-1.

29 Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. RE#: High performance derivative-based regex matching with intersection, complement, and restricted lookarounds. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704837.

30 Margus Veanes, Thomas Ball, Gabriel Ebner, and Ekaterina Zhuchko. Symbolic automata: Omega-regularity modulo theories. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704838.

31 Ekaterina Zhuchko. Lean sources for this paper, 2025. URL: https://github.com/ezhuchko/finiteness-derivatives.

32 Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. 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)*, pages 279–292. ACM, 2024. doi:10.1145/3636501.3636959.