Regex Decision Procedures in Extended RE#

Ian Erik Varatalu[1], Margus Veanes[2(✉)], Ekaterina Zhuchko[1], and Juhan Ernits[1]

[1] Tallinn University of Technology, Tallinn, Estonia
{ian.varatalu,ekzhuc,juhan.ernits}@taltech.ee
[2] Microsoft Research, Redmond, USA
margus@microsoft.com

Abstract. We develop decision procedures for *extended regular expressions* in the new **ERE#** framework that uses *span semantics*, utilizing the power of *symbolic derivatives*. We prove a normal form theorem in Lean for **ERE#** that is closed under all Boolean operations and provides the basis for the given decision procedures. The tool is evaluated on existing SMT benchmarks for regexes that shows it to be the fastest solver to date – *often orders of magnitude faster than state-of-the-art* – albeit specialized for the *single-variable fragment* of string theory.

1 Introduction

The new and currently *fastest nonbacktracking* regex matcher **RE#** [46] supports match search with regexes extended with *intersection* (`&`), *complement* (`~`), and restricted *lookarounds*. Extended operators in **RE#** provide a *separation of concerns* making it possible to express typical search patterns more naturally and compactly, as discussed at length in [46]. In this work, we develop decision procedures for the *Boolean closure* **ERE#** of **RE#** that enables various practical applications for *debugging*, *compiler optimizations*, and *verification* tasks.

Support for `&` and `~` implies a need to decide *nonemptiness* of regexes as a basic *debugging* feature. For example, the regex `.*\d.*&~(.*\w.*)` must match at least one digit but may not contain any word-letters – thus, it *matches nothing* because all digits are word-letters. Character classes can also be easily misunderstood due to subtle semantic differences between platforms or feature interactions in combination with common regex options, such as *case insensitivity* (typically inlined with `(?i:...)`) where *equivalence* checking is essential for ensuring correctness and consistency of the intended behavior. It is possible that $R_1 \equiv R_2$ but $\texttt{(?i:}R_1\texttt{)} \not\equiv \texttt{(?i:}R_2\texttt{)}$, e.g., for R_1 = `[^D]` and R_2 = `[\u0000-CE-\uFFFF]`.

During derivative based compilation of regexes into DFAs, it is common to encounter unions of the form $R_1|R_2$ as target states of transitions. In order to reduce memory footprint, it is crucial to decide if R_i *subsumes* R_j, and if so, to reuse R_i as the sole target state, instead of introducing a new state for the union.

R. Piskac and Z. Rakamarić (Eds.): CAV 2025, LNCS 15933, pp. 106–129, 2025.
https://doi.org/10.1007/978-3-031-98682-6_7

Several basic rewrites necessary were already pointed out in [40], but *bounded loops* in particular are highly problematic [44]. E.g., the derivative of a regex R = `.*a?{`k`}` for `a` is R`|a?{`k`-1}` that can be simplified to R because R subsumes `a?{`$k-1$`}`, where `a?` abbreviates `a|`ε. Omitting such rewrites can quickly lead to a state space explosion up to size 2^k – *observe that* $k = O(2^{|R|})$ *here*! For DFA generation, such subsumption checking must be fast in practice.

Dealing with differences between PCRE and POSIX semantics for a regex is a common problem.[1] In PCRE, also known as *backtracking* or *leftmost-eager* semantics, the union operator `|` is *noncommutative* where in a regex R_1`|`R_2, a match for R_2 is only sought when R_1 fails to match. Note that this difference is relevant for *span* semantics that has recently been formalized in Lean [50], but irrelevant for *language* semantics (`IsMatch`) that is identical under both PCRE and POSIX. A technique to decide if a union R_1`|`R_2 in PCRE has the same span semantics under POSIX, where union is *commutative*, is to decide match equivalence between R_1`|`R_2 and R_1`|`R_2`&~(`R_1`._*)`.

If the span semantics of a regex *differs* between PCRE and POSIX then the regex may contain unreachable cases under PCRE. E.g., the BurntSushi/rebar benchmarking tool [24] – *widely used for industrial PCRE matchers* – uses a dictionary benchmark containing unions such as `may|mayo`. The regex `may|mayo` will never match `"mayo"` for any input under PCRE semantics and has the exact same behavior as the regex `may`. Most industrial regex matchers use PCRE semantics, resulting in different behavior to what is intuitively assumed, as `|` is not union of languages in PCRE. It is a common programming error to define a regex with unintended behavior this way. For example, by using the above technique, `may|mayo&~(may_*)` effectively deletes the alternative `mayo` from `may|mayo` while `mayo|may` would remain intact because `may&~(mayo_*)` ≡ `may`.

The logic **ERE#**, that is introduced in Sect. 4, is a novel contribution and fundamental for many decision problems. In particular, it lifts **RE#** to the status of an *Effective Boolean Algebra* over spans. **RE#** is closed only under *intersection* and each regex admits a *linear* translation to a core regex of the form `(?<=`R_1`)`R_2`(?=`R_3`)` where no R_i contains lookarounds. The *span semantics* $(u_1, u_2, u_3) \models$ `(?<=`R_1`)`R_2`(?=`R_3`)` intuitively means that $u_1u_2u_3$ is a word such that $u_i \in \mathcal{L}(R_i)$ where u_2 is the main match with u_1 and u_3 as the surrounding context. For many decision problems, like subsumption, it became necessary to support regexes like `(?<=`R_1`)`R_2`(?=`R_3`)&~(`$(?<=R_1')R_2'(?=R_3')$`)` that fall outside the fragment for matching in **RE#**.

Currently, **ERE#** semantics is not directly expressible in SMT-LIB as it would require support for lookarounds and span semantics. Section 7 proposes an *SMT-LIB format* for extending `RegLan` with *lookarounds* and *span semantics*.

Contributions. We introduce an extension **ERE#** of the **RE#** class that is Boolean closed and formalize a normal form theorem (Theorem 2) for it in *Lean* allowing us to develop decision procedures for **ERE#**, including *emptiness*, *subsumption*, and *equivalence*, that are of primary interest in the **ERE#** framework,

[1] See for example https://stackoverflow.com/questions/4733416.

as discussed above. It is also possible to use **ERE#** for pre-processing in SMT solvers. One can invoke **ERE#** from the simplifier in Z3 [39], to pre-process `RegLan` constraints, which can moreover be supported incrementally [12]. The main decision procedures for **ERE#** reduce, via the normal form theorem and the nonemptiness algorithm (Theorem 1), to nonemptiness in **ERE**.

We have conducted extensive evaluation on existing SMT benchmarks for the fragment of **ERE#** without lookarounds. The evaluation focuses on available SMT benchmarks that include all we could find on `RegLan` and benchmarks that reduce to the single string variable fragment of string constraints. The experiments demonstrate that **ERE#** consistently outperforms all state-of-the-art solvers, often by orders of magnitude, see Fig. 1. The main reasons are: 1) *specialization to regex decision problems* combined with alphabet compression to bit-vectors; 2) aggressive rewrite rules that rely on symbolic derivatives and transition regex rewrites [43] – in particular lazy propagation rules for *intersection* and *complement*; 3) use of reversal theorem for $\mathbf{ERE}_{\leq}$ [50] – to decide nonemptiness of the *reverse* of a regex; 4) use of XOR as a Boolean operator directly supported by derivatives, for deciding equivalence in some cases.

Artifact. The paper is accompanied by an *artifact* [47] for Lean formalization of Theorem 2, and evaluation confirming the results in Fig. 1 and Table 1.

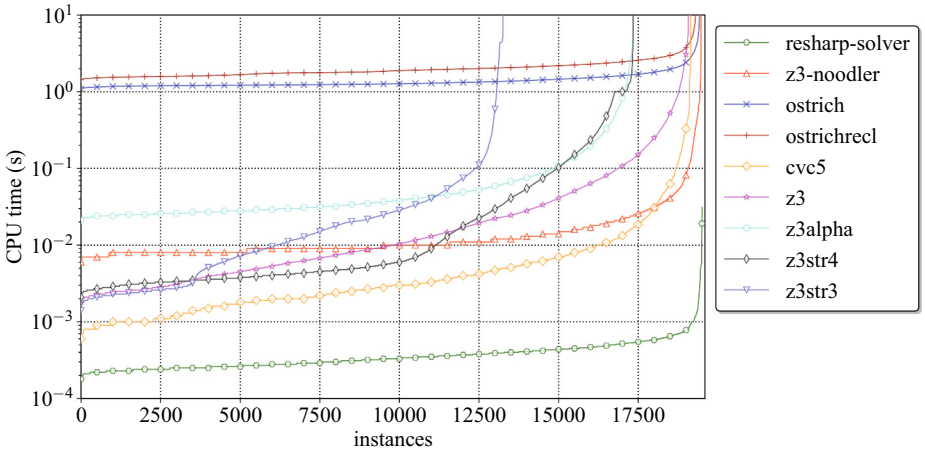


Fig. 1. Cactus plot of CPU time for 9 solvers and 19 509 SMT benchmarks. The y-axis is in **log-scale**. For **ERE#**-solver (`resharp-solver`) no benchmark needed >0.1 s and less than 100 benchmarks needed >0.01 s.

2 Related Work

The development of string constraint solvers has a long-standing history, with prominent general-purpose SMT solvers such as Z3 [39] and CVC5 [5], along-

side specialized solvers [10,11,19,36]. String constraint solving approaches vary widely, with methods based on a mixture of automata, word equations, and other techniques. Many existing tools handle a broader range of constraints, including those beyond regular languages, such as context-free constraints. However, our work is not intended to compete with these general-purpose solvers; instead, we focus on a specialized fragment relevant to our setting. For a comprehensive overview of existing techniques and tools, we refer to the recent surveys [3,26]. Below we outline the key differences between the existing tools and our approach.

The solvers which support derivative-based lazy exploration of regular expressions are the sequence solver [43] in Z3 and CVC5 [5]. The former was the first to use transition regexes as part of the solver. However, our method leverages systematic simplifications and rewrites, enabling us to solve many SMT-LIB benchmarks in a fraction of a second, whereas the sequence solver fails to do so even after hours. Additionally, our solution is specialized for regexes, avoiding the overhead of supporting other theories. CVC5 [5] also uses a derivatives based approach [32] to solve regular expression constraints in its string solver [31,41], similarly relying on aggressive simplifications.

Kepler$_{22}$, introduced in [30], employs a two-phase approach for solving string constraints. In the first phase, it constructs a cyclic reduction tree that represents all solutions to a conjunction of word equations and regular membership predicates. The second phase uses specialized procedures to infer length constraints from the set of all solutions, constituting a decision procedure for the straight line and quadratic fragments of string equations. While the authors report promising experimental results, we were not able to find an implementation to include in our evaluation. The tool NFA2SAT, presented in [33], performs an eager reduction of string operations and regular expression constraints into SAT, enabling incremental SAT solving. However, we were not able to find an artifact to include in our evaluation.

Several extensions to Z3 incorporate support for string constraints. The analysis in [9] focuses on fragments without string equality (i.e., without word equations), which aligns with the scope of our solver. Z3-Noodler [19] utilizes the MATA library [20] that implements basic algorithms for automata and also fast simulation reduction and antichain-based language inclusion checking. It uses novel techniques such as a stabilization-based procedure [14,18] for string-constraint solving. Z3str4 [36] builds on Z3str3 [10] and extends it with a string-to-bit-vector reduction. Z3alpha [34] is built on top of Z3 and the novelty lies in untilizing Monte Carlo Tree Search based SMT strategy synthesis. It ranked second in the 24-second performance category at SMT-COMP 2024, solving more tasks than all other solvers except Z3-Noodler.

Another notable tool is Ostrich [17], a solver that supports advanced regular expression features such as capture groups, lazy quantifiers, and anchors. Several extensions of Ostrich have been developed. For instance, [16] introduces a model based on prioritized streaming string transducers, while [27,28] builds on cost-enriched finite-state automata. Another recent extension to Ostrich is

the tool SECO [29] based on parametric symbolic automata which extends symbolic automata to allow free variables on the transition guards. Ostrich has also been extended using Parikh images [23] to reason about word lengths. While these Parikh image-based approaches have also been applied to accelerate string constraint solving [25], they fall outside the scope of this paper.

Equivalence algorithms for classical regular expressions have been studied in [1,2] based on *partial derivatives* [4]. The main algorithm was also implemented in [37] using Coq, and it operates incrementally by avoiding the construction of the full automaton during the process. The additional challenge in the case of **ERE#** is twofold: to support intersection and complement incrementally, and to work modulo large (or infinite) alphabets $\mathcal{A}$ (given as an EBA), which we are currently investigating. Moreover, in the case of **ERE#**, equivalence should ideally also support lookarounds and therefore be based on span semantics.

3 Preliminaries

This section includes the main background notations and definitions used in the rest of the paper. To support large alphabets such as Unicode, we use *effective Boolean algebras* [22] (EBAs) to represent *character classes* symbolically via predicates, typical examples include: *digits* `\d`, *word-letters* `\w`, and *white-space characters* `\s`. Extended regular expressions or regexes support *intersection* and *complement* as well as *lookarounds* in the full class $\mathbf{ERE}_{\leq}$ [46,50]. The latter requires their semantics to be based on *locations* and *spans* due to context conditions imposed by lookarounds. Regexes use nested if-then-else terms called *transition regexes* [43,50] to represent their *symbolic derivatives*. The main intuition is that the symbolic derivative $\delta(R)$ of a regex R is a transition regex representing a *partial evaluation* for R of the Brzozowski derivative [15] $D_a(R)$ for $a \in \Sigma$. The *evaluation* of $\delta(R)$ for $a \in \Sigma$, $\delta(R)[a]$, equals $D_a(R)$. The class **ERE#**, defined in Sect. 4, forms a *proper* subclass of $\mathbf{ERE}_{\leq}$ but its match semantics is also based on spans and coincides with the match semantics of the whole $\mathbf{ERE}_{\leq}$. The rest of this section defines these notions formally.

Before the formal definitions below, we illustrate the notions using the regex R = `_*\d_*&_*\w_*&_*\s_*` that matches any string containing a digit, a word-letter, and a white-space character, where `_` matches *all* characters and `&` is regex *intersection*. The symbolic derivative $\delta(R)$ of R is the transition regex:

$$\delta(R) = \textbf{ite(}\texttt{\textbackslash d}\textbf{, }\texttt{\_*\textbackslash s\_*}\textbf{, ite(}\texttt{\textbackslash w}\textbf{, }\texttt{\_*\textbackslash d\_*\&\_*\textbackslash s\_*}\textbf{, ite(}\texttt{\textbackslash s}\textbf{, }\texttt{\_*\textbackslash d\_*\&\_*\textbackslash w\_*}\textbf{, } R\textbf{)))}$$

The transition regex is illustrated as a binary decision tree in Fig. 2 whose *conditions* are the highlighted predicates `\d`, `\w`, and `\s`. The *else-case* branches of $\delta(R)$ are *dashed* and the *leaves* of $\delta(R)$ are regexes (including R itself). E.g., $\delta(R)$[`'a'`] = `_*\d_*&_*\s_*` and $\delta(R)$[`'#'`] = R. When computing $\delta(R)$ the alphabet EBA was used to to eliminate unreachable subterms in $\delta(R)$ through *cleaning* [43]. Therefore, $\delta(R)$ was simplified by using the facts that `\d` implies `\w` (since all digits are word-letters), while `\s` is disjoint from both `\w` and `\d`.

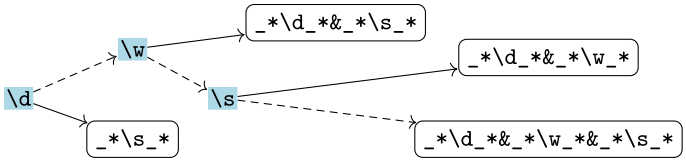


Fig. 2. Clean symbolic derivative of the regex `_*\d_*&_*\w_*&_*\s_*`.

Effective Boolean Algebras. An *Effective Boolean Algebra (EBA)* over an element universe Σ is a tuple $(\Sigma, \mathcal{A}, \vDash, \bot, \_, \sqcup, \sqcap, ^c)$ where $\mathcal{A}$ is a set of *predicates* that is closed under the Boolean connectives and contains $\bot$ and $\_$. For $a \in \Sigma$ and $\alpha \in \mathcal{A}$ the *models* relation $a \vDash \alpha$ with $[\![\alpha]\!] \stackrel{\text{DEF}}{=} \{a \in \Sigma \mid a \vDash \alpha\}$ obeys classical Tarski laws such that $[\![\bot]\!] = \emptyset$ and $[\![\_]\!] = \Sigma$. For $\alpha, \beta \in \mathcal{A}$ let $\alpha \equiv \beta \stackrel{\text{DEF}}{=} [\![\alpha]\!] = [\![\beta]\!]$. If $\alpha \not\equiv \bot$ then α is *satisfiable* ($\mathbf{SAT}(\alpha)$). All the connectives must be *computable* and $\vDash$ must be *decidable*.

We write $\mathcal{A}$ also for the EBA itself and say that $\mathcal{A}$ is *decidable* if $\mathbf{SAT}(\alpha)$ is decidable. We use $(\Sigma, \mathcal{A}, \vDash, \bot, \_, \sqcup, \sqcap, ^c)$ as a given *core alphabet* EBA. When working with regexes, it is a standard assumption that `.` is a predicate denoting all characters *except* the newline character (predicate) `\n`, i.e., $\_ \equiv \texttt{.} \sqcup \texttt{\n}$.

Locations and Spans. A *location* is a pair of words in $\Sigma^* \times \Sigma^*$. Let $w \in \Sigma^*$. A *location in* w is a location (u, v) such that $w = uv$. The intuition is that a location in w specifies a border inside w. A location (ϵ, w) is *initial* and a location (w, ϵ) is *final*. For any location x let the *kind* of x be $(\mathit{IsInitial}(x), \mathit{IsFinal}(x)) \in \mathbb{B} \times \mathbb{B}$ where $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$.

A *span* is a triple of words in $\mathbf{Span} \stackrel{\text{DEF}}{=} \Sigma^* \times \Sigma^* \times \Sigma^*$. A span $\theta = (u, v, s)$ has two locations the *beginning* location $\mathit{beg}(\theta) \stackrel{\text{DEF}}{=} (u, vs)$ and the *end* location $\mathit{end}(\theta) \stackrel{\text{DEF}}{=} (uv, s)$; $\mathit{prefix}(\theta) \stackrel{\text{DEF}}{=} u$ is the *prefix* of the span, $\mathit{suffix}(\theta) \stackrel{\text{DEF}}{=} s$ is the *suffix* of the span, and $\mathit{match}(\theta) \stackrel{\text{DEF}}{=} v$ is the *match* of the span. The *word* of the span is $\mathit{word}(\theta) \stackrel{\text{DEF}}{=} uvs$. We lift all the definitions to *sets* of spans as usual.

A *span in* w is a span (u, v, s) such that $w = uvs$. Intuitively, a span (u, v, s) represents a regex match in a word w with v as the primary matched substring of w where the prefix and the suffix are some sufficient lookaround context words. The *width* of a span (u, v, s) is the length $|v|$ of its match v.

Full Class ERE with Lookarounds. The class $\mathbf{ERE}_{\leq}$ is defined as follows. Members of $\mathbf{ERE}_{\leq}$ are denoted here by R. *Concatenation* $(\cdot)$ is often implicit by juxtaposition. All operators appear in order of precedence where *union* $(|)$ binds weakest and *complement* $(\tilde{\ })$ binds strongest. Let $\psi \in \mathcal{A}$ and let $m > 0$.

$$R ::= \psi \mid \varepsilon \mid R_1|R_2 \mid R_1\&R_2 \mid R_1 \cdot R_2 \mid R\{m\} \mid R* \mid \tilde{\ }R \mid$$
$$\texttt{(?<=}R\texttt{)} \mid \texttt{(?<!}R\texttt{)} \mid \texttt{(?=}R\texttt{)} \mid \texttt{(?!}R\texttt{)}$$

The regex denoting *nothing* is the predicate $\bot$. Let $R\{0\} \stackrel{\text{DEF}}{=} \varepsilon$. We write $R+$ for $R \cdot R*$. Union is also called *alternation* or *disjunction*. The regexes `(?=`R`)`, `(?!`R`)`,

`(?<=R)`, and `(?<!R)` are *lookarounds*; `(?=R)` is *(positive) lookahead*, `(?!R)` is *negative lookahead*, `(?<=R)` is *(positive) lookbehind*, and `(?<!R)` is *negative lookbehind*. Let `\A` $\stackrel{\text{DEF}}{=}$ `(?<!_)` and `\z` $\stackrel{\text{DEF}}{=}$ `(?!_)`.

ERE is the subclass of $\mathbf{ERE}_{\leq}$ without lookarounds and $\mathbf{ERE}_{\text{⚓}}$ extends **ERE** by allowing also the *start anchor* `\A` and the *end anchor* `\z` as primitive regexes. $\mathbf{RE}_{\leq}$ (resp. **RE**) is the subclass of $\mathbf{ERE}_{\leq}$ (resp. **ERE**) without `&` and `~`.

Match Semantics of ERE with Lookarounds. The match semantics of regexes in $\mathbf{ERE}_{\leq}$ uses spans [46,50]. Equivalent formulations of the semantics of **RE** with lookarounds appear originally in [38, Section 3.7] via derivation relations, and in [7,35] using a variant of spans. Let $\theta = (u, v, s)$ be a span in a word w. Then θ *models* R or R *matches* θ is denoted by $\theta \models R$. We say that R *matches* w iff R matches some span in w. Recall that $R\{0\} \stackrel{\text{DEF}}{=} \varepsilon$ and let $m > 0$.

$$
\begin{array}{rcl}
\theta \models L \mid R & \stackrel{\text{DEF}}{=} & \theta \models L \vee \theta \models R \\
\theta \models L \,\&\, R & \stackrel{\text{DEF}}{=} & \theta \models L \wedge \theta \models R \\
\theta \models \sim R & \stackrel{\text{DEF}}{=} & \theta \not\models R \\
\theta \models R* & \stackrel{\text{DEF}}{=} & \exists n \geq 0 : \theta \models R\{n\} \\
(u, v, s) \models \varepsilon & \stackrel{\text{DEF}}{=} & v = \epsilon \\
(u, v, s) \models \psi & \stackrel{\text{DEF}}{=} & |v| = 1 \wedge v \vdash \psi \\
(u, v, s) \models L{\cdot}R & \stackrel{\text{DEF}}{=} & \exists x, y : v = xy \wedge (u, x, ys) \models L \wedge (ux, y, s) \models R \\
(u, v, s) \models R\{m\} & \stackrel{\text{DEF}}{=} & \exists x, y : v = xy \wedge (u, x, ys) \models R \wedge (ux, y, s) \models R\{m{-}1\} \\
(u, v, s) \models \texttt{(?<=R)} & \stackrel{\text{DEF}}{=} & v = \epsilon \wedge (\epsilon, u, s) \models \_*{\cdot}R \\
(u, v, s) \models \texttt{(?<!R)} & \stackrel{\text{DEF}}{=} & v = \epsilon \wedge (\epsilon, u, s) \not\models \_*{\cdot}R \\
(u, v, s) \models \texttt{(?=R)} & \stackrel{\text{DEF}}{=} & v = \epsilon \wedge (u, s, \epsilon) \models R{\cdot}\_* \\
(u, v, s) \models \texttt{(?!R)} & \stackrel{\text{DEF}}{=} & v = \epsilon \wedge (u, s, \epsilon) \not\models R{\cdot}\_* \\
\mathcal{M}(R) & \stackrel{\text{DEF}}{=} & \{\theta \in \mathbf{Span} \mid \theta \models R\} \\
L \equiv R & \stackrel{\text{DEF}}{=} & \mathcal{M}(L) = \mathcal{M}(R)
\end{array}
$$

Intuitively, $(u, \epsilon, s) \models$ `(?=R)` means that there exists a match of R *starting* from the location (u, s), and $(u, \epsilon, s) \models$ `(?<=R)` means that there exists a match of R *ending* in the location (u, s). For all lookarounds, the matched span has always 0 width, i.e., lookarounds are a generalized form of anchors.

Observe that $\mathcal{M}(\_*) = \mathbf{Span}$, $\mathcal{M}(\bot) = \emptyset$, and all the Boolean connectives satisfy the EBA conditions of $\mathbf{ERE}_{\leq}$. Thus, $(\mathbf{Span}, \mathbf{ERE}_{\leq}, \models, \bot, \_*, |, \&, \sim)$ is an EBA, since $sp \models R$ is decidable for all $sp \in \mathbf{Span}$ and $R \in \mathbf{ERE}_{\leq}$ because $\vdash$ is decidable in $\mathcal{A}$.

Transition Regexes. A *transition regex* is either a *leaf* $R \in \mathbf{ERE}_{\text{⚓}}$, or an ITE expression **ite**(ψ, f, g) with *condition* $\psi \in \mathcal{A}$, *then-case* f and an *else-case* g that are transition regexes. The *evaluation* of f for $a \in \Sigma$, denoted by $f[a]$, is the leaf regex reached by a.

$$
R[a] \stackrel{\text{DEF}}{=} R \qquad \textbf{ite}(\psi, f, g)[a] \stackrel{\text{DEF}}{=} \begin{cases} f[a], \textbf{ if } a \vdash \psi; \\ g[a], \textbf{ otherwise}. \end{cases}
$$

All binary operators $\diamond$ over regexes are lifted to transition regexes by propagating the operations into the leaves. Unary operators such as ~ and (?=) are propagated analogously. Here $R \in \mathbf{ERE}_{\text{⚓}}$.

$$R \diamond \mathbf{ite}(\psi, f, g) \overset{\text{DEF}}{=} \mathbf{ite}(\psi, R \diamond f, R \diamond g) \quad \mathbf{ite}(\psi, f, g) \diamond h \overset{\text{DEF}}{=} \mathbf{ite}(\psi, f \diamond h, g \diamond h)$$

The set of leaves of a transition regex f, denoted by $Lvs(f)$, are the *reachable* leaves of f. In practice, transition regexes are kept *clean* [43] by construction, but here we filter out unreachable leaves explicitly. So $\mathcal{A}$ is assumed *decidable*.

$$Lvs(f) \overset{\text{DEF}}{=} Lvs(\_, f) \qquad Lvs(\psi, R) \overset{\text{DEF}}{=} \begin{cases} \{R\}, & \textbf{if SAT}(\psi); \\ \emptyset, & \textbf{otherwise}. \end{cases}$$
$$Lvs(\psi, \mathbf{ite}(\alpha, f, g)) \overset{\text{DEF}}{=} Lvs(\psi \sqcap \alpha, f) \cup Lvs(\psi \sqcap \alpha^c, g)$$

Symbolic Derivatives in ERE with Anchors. Here we define *nullability* and *symbolic derivatives* for regexes $R \in \mathbf{ERE}_{\text{⚓}}$. A (symbolic) derivative of R is either taken with respect to an *initial* (and nonfinal) location or a *noninitial* (and nonfinal) location, *final* locations are not used for computing derivatives. Nullability is defined for all locations.

We let the *kind* κ of a location x be $(IsInitial(x), IsFinal(x)) \in \mathbb{B} \times \mathbb{B}$. The three main kinds are INI = **(true, false)**, MID = **(false, false)**, and FIN = **(false, true)**. Using the definitions below we let $Null(R) \overset{\text{DEF}}{=} Null_{\text{MID}}(R)$ and $\delta(R) \overset{\text{DEF}}{=} \delta_{\text{MID}}(R)$.

Nullability. We define nullability $Null_\kappa(R)$ of a regex $R \in \mathbf{ERE}_{\text{⚓}}$ relative to a *location kind* $\kappa \in \mathbb{B} \times \mathbb{B}$. Let $m > 0$ and recall that $R\{0\} \overset{\text{DEF}}{=} \varepsilon$.

$$Null_{(initial,final)}(\backslash\mathtt{A}) \overset{\text{DEF}}{=} initial \qquad Null_\kappa(R \mid S) \overset{\text{DEF}}{=} Null_\kappa(R) \vee Null_\kappa(S)$$
$$Null_{(initial,final)}(\backslash\mathtt{z}) \overset{\text{DEF}}{=} final \qquad Null_\kappa(R \,\&\, S) \overset{\text{DEF}}{=} Null_\kappa(R) \wedge Null_\kappa(S)$$
$$Null_\kappa(\varepsilon) \overset{\text{DEF}}{=} \mathbf{true} \qquad Null_\kappa(R{\cdot}S) \overset{\text{DEF}}{=} Null_\kappa(R) \wedge Null_\kappa(S)$$
$$Null_\kappa(R*) \overset{\text{DEF}}{=} \mathbf{true} \qquad Null_\kappa(R\{m\}) \overset{\text{DEF}}{=} Null_\kappa(R)$$
$$Null_\kappa(\psi) \overset{\text{DEF}}{=} \mathbf{false} \qquad Null_\kappa(\sim R) \overset{\text{DEF}}{=} \neg\, Null_\kappa(R)$$

So $Null_{\text{INI}}(R)$ is applied in locations of kind INI and $Null_{\text{FIN}}(R)$ is applied in locations of kind FIN. The trivial special case of the single empty location (ϵ, ϵ) in an empty word ϵ that is both initial and final is omitted from discussions.

Symbolic Derivatives. Symbolic derivatives, represented by transition regexes, are only evaluated for *nonfinal* locations, i.e., $\kappa \in \{\text{INI}, \text{MID}\}$ below. We apply the definition of derivatives from [48] and simultaneously lift the definition to transition regexes, similar to [43]. Let $\psi \in \mathcal{A}$, $m > 0$, and ⚓ $\in \{\backslash\mathtt{A}, \backslash\mathtt{z}\}$.

$$\delta_\kappa(\text{⚓}) \overset{\text{DEF}}{=} \bot \qquad \delta_\kappa(R*) \overset{\text{DEF}}{=} \delta_\kappa(R){\cdot}R*$$
$$\delta_\kappa(\varepsilon) \overset{\text{DEF}}{=} \bot \qquad \delta_\kappa(R\{m\}) \overset{\text{DEF}}{=} \delta_\kappa(R){\cdot}R\{m-1\}$$
$$\delta_\kappa(R \,\&\, S) \overset{\text{DEF}}{=} \delta_\kappa(R) \,\&\, \delta_\kappa(S) \qquad \delta_\kappa(\psi) \overset{\text{DEF}}{=} \mathbf{ite}(\psi, \varepsilon, \bot)$$
$$\delta_\kappa(R \mid S) \overset{\text{DEF}}{=} \delta_\kappa(R) \mid \delta_\kappa(S) \qquad \delta_\kappa(R{\cdot}S) \overset{\text{DEF}}{=} \begin{cases} \delta_\kappa(R){\cdot}S \mid \delta_\kappa(S), & \textbf{if } Null_\kappa(R); \\ \delta_\kappa(R){\cdot}S, & \textbf{otherwise}. \end{cases}$$
$$\delta_\kappa(\sim R) \overset{\text{DEF}}{=} \sim\delta_\kappa(R)$$

Observe that, by definition, all regex operators are lifted to transition regexes.

4 Decision Procedures for ERE#

The focus of the paper in on the subclass **ERE#** of $\mathbf{ERE}_{\leq}$ that contains $\mathbf{ERE}_{\barwedge}$ combined with a restricted fragment of lookarounds. In the formal definition below L defines $\mathbf{ERE}_{\barwedge}$, and R defines **ERE#**. Let $\psi \in \mathcal{A}$ and $m > 0$.

$$L ::= \psi \mid \varepsilon \mid \texttt{\A} \mid \texttt{\z} \mid L_1 \mid L_2 \mid L_1 \& L_2 \mid L_1 \cdot L_2 \mid L\{m\} \mid L* \mid \tilde{}\,L$$
$$R ::= L \mid \texttt{(?<=}L\texttt{)}\cdot R \mid \texttt{(?<!}L\texttt{)}\cdot R \mid R \cdot \texttt{(?=}L\texttt{)} \mid R \cdot \texttt{(?!}L\texttt{)} \mid R_1 \mid R_2 \mid R_1 \& R_2 \mid \tilde{}\,R$$

We show below that all regexes in **ERE#** have a normal form that is a union of *core regexes* that are regexes of the form $\texttt{(?<=}L_1\texttt{)}\cdot L_2 \cdot \texttt{(?=}L_3\texttt{)}$ where $L_i \in \mathbf{ERE}_{\barwedge}$, with L_2, $\texttt{(?<=}L_1\texttt{)}\cdot L_2$, $L_2 \cdot \texttt{(?=}L_3\texttt{)}$ as special cases because $\texttt{(?<=}\varepsilon\texttt{)} \equiv \texttt{(?=}\varepsilon\texttt{)} \equiv \varepsilon$. The definition of **ERE#** properly subsumes the definition of **RE#** in [46] where **RE#** contains all core regexes and is only closed under `&`.

The match semantics of **ERE#** is based on $\mathbf{ERE}_{\leq}$. In particular, it follows that $(\mathbf{Span}, \mathbf{ERE\#}, \models, \bot, \_*, |, \&, \tilde{})$ is an EBA. The main decision procedures we are focusing on for **ERE#** are *emptiness*, *subsumption*, and *equivalence*.

4.1 Deciding Nonemptiness of Core Regexes in ERE#

Here we consider nonemptiness of *core regexes* in **ERE#**. We later show how the general case of nonemptiness of all regexes in **ERE#** reduces to nonemptiness of core regexes by showing that **ERE#** is a *decidable* EBA. The nonemptiness algorithm builds on symbolic derivatives and transition terms and can be abstractly formulated as a fixpoint procedure that relies on the associativity, commutativity, and idempotence (ACI) of regex union, which guarantees finiteness of the state space and thus termination.

For a union regex let $\mathit{Set}(R_1 \mid R_2) \stackrel{\text{DEF}}{=} \mathit{Set}(R_1) \cup \mathit{Set}(R_2)$. Let $\mathit{Set}(\bot) \stackrel{\text{DEF}}{=} \emptyset$ and for all other regexes R let $\mathit{Set}(R) \stackrel{\text{DEF}}{=} \{R\}$. Let f be a transition regex. The set of all *states* of f is the set of all $q \in \mathit{Set}(\ell)$ for $\ell \in \mathit{Lvs}(f)$, i.e.,

$$\mathit{States}(f) \stackrel{\text{DEF}}{=} \bigcup_{\ell \in \mathit{Lvs}(f)} \mathit{Set}(\ell)$$

For all $R = \texttt{(?<=}L_1\texttt{)}\cdot L_2 \cdot \texttt{(?=}L_3\texttt{)} \in \mathbf{ERE\#}$, where $L_i \in \mathbf{ERE}_{\barwedge}$, we decide nonemptiness of R by reducing it to nonemptiness in $\mathbf{ERE}_{\barwedge}$ as follows.

$$\mathit{IsNonempty}(\texttt{(?<=}L_1\texttt{)}\cdot L_2 \cdot \texttt{(?=}L_3\texttt{)}) \stackrel{\text{DEF}}{=} \mathit{IsNonempty}(\_*\cdot L_1 \cdot L_2 \cdot L_3 \cdot \_*)$$

Let $L \in \mathbf{ERE}_{\barwedge}$. The function $\mathit{IsNonempty}(L)$, unless L is trivially nullable, computes reachable states from L and returns **true** upon reaching a nullable state.

$\mathit{IsNonempty}(L) \stackrel{\text{DEF}}{=}$ **if** $\mathit{Null}_{(\mathbf{true},\mathbf{true})}(L) \vee \mathit{Null}_{\text{INI}}(L)$ **return true**
$\rho(L) \leftarrow \delta_{\text{INI}}(L);\ Q \leftarrow \{L\} \cup \mathit{States}(\rho(L))$
while $\nexists\, q \in Q : (\mathit{Null}(q) \vee \mathit{Null}_{\text{FIN}}(q)) \wedge \exists\, q \in Q \setminus \mathit{Dom}(\rho)$ **do**
$\rho(q) \leftarrow \delta(q);\ Q \leftarrow Q \cup \mathit{States}(\rho(q))$
return $\exists\, q \in Q : \mathit{Null}(q) \vee \mathit{Null}_{\text{FIN}}(q)$

$$\text{NOT-ELIM}\ \frac{\texttt{~((?<=}X\texttt{)}Y\texttt{(?=}Z\texttt{))}}{\texttt{(?<!}X\texttt{)\_*|~}Y\texttt{|\_*(?!}Z\texttt{)}} \qquad \text{L-DISTR}\ \frac{(Y\,|\,Z)X}{YX\,|\,ZX} \qquad \text{R-DISTR}\ \frac{X(Y\,|\,Z)}{XY\,|\,XZ}$$

$$\text{NLA-ELIM}\ \frac{\texttt{(?!}X\texttt{)}}{\texttt{(?=~(}X\texttt{\_*)\z)}} \qquad \text{LA-JOIN}\ \frac{\texttt{(?=}X\texttt{)(?=}Y\texttt{)}}{\texttt{(?=}X\texttt{\_*\&}Y\texttt{\_*)}} \qquad \text{LB-JOIN}\ \frac{\texttt{(?<=}X\texttt{)(?<=}Y\texttt{)}}{\texttt{(?<=\_*}X\texttt{\&\_*}Y\texttt{)}}$$

$$\text{NLB-ELIM}\ \frac{\texttt{(?<!}X\texttt{)}}{\texttt{(?<=\A~(\_*}X\texttt{))}} \qquad \text{AND-ELIM}\ \frac{\texttt{(?<=}X\texttt{)}Y\texttt{(?=}Z\texttt{)\&(?<=}X'\texttt{)}Y'\texttt{(?=}Z'\texttt{)}}{\texttt{(?<=}X\texttt{)(?<=}X'\texttt{)(}Y\texttt{\&}Y'\texttt{)(?=}Z\texttt{)(?=}Z'\texttt{)}}$$

Fig. 3. Main inference rules in $\mathbf{ERE}_{\leq}$ used in Theorem 2. Let NOT-ELIM#= NOT-ELIM∘NLB-ELIM∘NLA-ELIM, and AND-ELIM#= AND-ELIM∘LB-JOIN∘LA-JOIN.

Example 1. Let $R :=$ `(?<=\A)`ε`(?=~\z)`. Then $Null_{\text{INI}}$(`_*\A~\z_*`) = **true**. Here R only matches spans of 0 width whose start location is initial and end location in nonfinal, i.e., $\mathcal{M}(R) = \{(\epsilon, \epsilon, v) \mid v \in \Sigma^+\}$.

Let $R :=$ `(?<=\A)_(?=\z)`. Then $L =$ `_*\A_\z_*` and $\rho(L) = L\,|$`\z_*`. So initially $Q = \{L,$ `\z_*`$\}$ because $States(L\,|$`\z_*`$) = \{L,$ `\z_*`$\}$ and we have that $Null_{\text{FIN}}$(`\z_*`) = **true**. Note that $\mathcal{M}(R) = \{(\varepsilon, a, \varepsilon) \mid a \in \Sigma\}$.

Theorem 1 (NonEmptiness). $\mathcal{M}(R) \neq \emptyset \Leftrightarrow$ *IsNonempty*(R) *holds for all core regexes* R *in* **ERE#**.

IsNonempty(R) can be extended to produce a *witness* when **true**. In most precise form, the produced witness can be in form of a *symbolic span* $(\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \boldsymbol{\alpha}_3) \in \mathcal{A}^* \times \mathcal{A}^* \times \mathcal{A}^*$ such that, for all spans (u_1, u_2, u_3), such that $u_\imath \vDash \boldsymbol{\alpha}_\imath$, it holds that $(u_1, u_2, u_3) \models R$, where $(a_i)_{i<n} \vDash (\psi_i)_{i<m}$ stands for $n = m$ and $\bigwedge_{i<n}(a_i \vDash \psi_i)$.

4.2 Lookaround Normal Form of ERE#

We start by providing a collection of equivalence preserving inference rules in $\mathbf{ERE}_{\leq}$, see Fig. 3, that are used to rewrite regexes in **ERE#** into the desired normal form. All rules have been formalized and proved correct in Lean. Observe that all rules preserve **ERE#**, i.e., lookaround bodies remain in $\mathbf{ERE}_{\bot}$.

The key new rule is NOT-ELIM. The other rules involving lookarounds are derived from similar rewrites in [46] where the normal form is *linear* in the size of the original **RE#** formula. In the case of **ERE#** the lookaround normal form can be *exponential* in the size of the original formula.

Let R be a Boolean combination of *core regexes*. We define the *negation normal form* $NNF(R)$ of R where regex complement ~ has been propagated via de Morgan extended with the rule NOT-ELIM#. It follows that no subregex of $NNF(R)$ outside $\mathbf{ERE}_{\bot}$ is negated and all lookarounds are positive.

Many further simplification laws are used as rewrites. The basic ones are treating (`_*`,$\bot$) as the units of (`&`,`|`) and as the zeros of (`|`,`&`), and $\bot$ is also the zero of $\cdot$ and ε is the unit of $\cdot$. Further rules include `(?<!`ε`)` $\equiv \bot$ and `(?!`ε`)` $\equiv \bot$, as well as `(?=`$\bot$`)` $\equiv \bot$ and `(?<=`$\bot$`)` $\equiv \bot$. Also, `_*\A` $\equiv$ `\A` and `\z_*` $\equiv$ `\z`.

Further practical rewrites for negative lookarounds with body $\psi \in \mathcal{A}$ are `(?<!`ψ`)` $\equiv$ `(?<=`ψ^c`|\A)` and `(?!`ψ`)` $\equiv$ `(?=`ψ^c`|\z)` as equivalent but simplified variants of NLB-ELIM and NLA-ELIM, respectively.

Example 2. The regex `(?!abc\z)` is intuitive, it states that the suffix of a match must not be `abc`. So `(?=~(abc\z)\z)` is equivalent but less intuitive.

To illustrate some rules, consider `~((?<=a)_*b_*|_*c_*)` in **ERE#**. Here we also make use of `~(_*b_*)` ≡ `[^b]*` and `~(_*c_*)` ≡ `[^c]*`, as well as `[^b]*&[^c]*` ≡ `[^bc]*` (using standard negated character class notation).

$$
\begin{aligned}
\texttt{\~{}((?<=a)\_*b\_*|\_*c\_*)} &\overset{\text{DEMORGAN}}{\equiv} \texttt{\~{}((?<=a)\_*b\_*)\&\~{}(\_*c\_*)} \\
&\overset{\text{NOT-ELIM}}{\equiv} \texttt{((?<!a)\_*|\~{}(\_*b\_*))\&\~{}(\_*c\_*)} \\
&\overset{\text{NLB-ELIM}}{\equiv} \texttt{((?<=[\^{}a]|\textbackslash A)\_*|\~{}(\_*b\_*))\&\~{}(\_*c\_*)} \\
&\equiv \texttt{((?<=[\^{}a]|\textbackslash A)\_*|[\^{}b]*)\&[\^{}c]*} \\
&\equiv \texttt{(?<!a)[\^{}c]*|[\^{}bc]*}
\end{aligned}
$$

For example, $(\varepsilon, \mathtt{b}, \varepsilon) \models$ `(?<!a)[^c]*` but $(\varepsilon, \mathtt{b}, \varepsilon) \not\models$ `(?<=a)_*b_*|_*c_*`.

4.3 Formalization in Lean

We now present the key result that establishes the correctness of the Lookaround Normal Form (LNF). Our approach follows the formalization presented in [50], which defines the match semantics for the entire class of regular expressions $\mathbf{ERE}_{\leq}$. In this work, we explicitly handle three distinct types of match semantics, each corresponding to different subclasses of regular expressions: $\mathbf{ERE}_{\perp}$, **ERE#**, and $\mathbf{ERE}_{\leq}$, and establish their relationships to ensure correct correspondence between their semantics. Additionally, we introduce an internal conversion between the `RESharp` type and its positive fragment, `PosRESharp`, which excludes complements and negative lookarounds.

First, we present some auxiliary definitions. The central definition in our formalization is the `lnf` function, which converts a regex in **ERE#** into a list of core regexes. The normalization function `LNF` in Theorem 2 is defined in terms of `lnf`. The `lnf` function is defined inductively and implements the inference rules presented in Fig. 3. Components such as `la_join`, `lb_join`, and `intersection` closely follow the corresponding rules.

The `lnf` function definition, returning a list of core regexes, is as follows.[2]

```
def lnf (r : RESharp 𝒜) : List (CoreRegex 𝒜) :=
match r with
| Ere r            => [EREa_to_CoreRegex r]
| Lookahead r la   => map (la_join la) (lnf r)
| Lookbehind lb r  => map (lb_join lb) (lnf r)
| NLookahead r la  => map (la_join (nLookahead la)) (lnf r)
| NLookbehind lb r => map (lb_join (nLookbehind lb)) (lnf r)
| Alt l r          => lnf l ++ lnf r
| Inter l r        => productWith intersection (lnf l) (lnf r)
| Compl r          => takeNegations (lnf r)
```

[2] The full Lean formalization is available in the supplemental material.

The most intricate case involves the complement operation. To handle this case more easily and to simplify the proof structure, we introduce an auxiliary function, `takeNegations`, that specifically deals with negated expressions.

```
def takeNegations (rs : List (CoreRegex 𝒜)) : List (CoreRegex 𝒜) :=
match rs with
| []      => [EREa_to_CoreRegex _*]
| c :: cs => let cs' := takeNegations cs
  map (.lb_join (nLookbehind c.left)) cs' ++
  map (.la_join (nLookahead c.right)) cs' ++
  map (intersection (EREa_to_CoreRegex (~c.regex))) cs'
```

Intuitively, the three lists correspond to the inference rule NOT-ELIM where any of the three components of the union can be satisfied in order to satisfy the negation of the whole expression.

Let us give the remaining definition, `LNF`, in the Theorem 2.

```
def LNF (r : RESharp 𝒜) : RESharp 𝒜 := re_sum_pos (map sem (lnf r))
```

It first transforms the input regex into its Lookaround Normal Form (LNF) using the `lnf` function. Then, it applies the `sem` function to each core regex in the LNF list, converting them into their corresponding **ERE#** representation e.g., the core regex (L_1, L_2, L_3) is converted into $(?<=L_1)\cdot L_2 \cdot (?=L_3)$. Finally, `re_sum_pos` combines the resulting list of **ERE#** expressions with the union operator.

Finally, we are ready to state the main theorem where `RESharp` denotes **ERE#**.

Theorem 2 (LNF)

```
theorem lnf_correct {R : RESharp 𝒜} {sp : Span Σ} : sp ⊨ R ↔ sp ⊨ LNF R
```

Theorem 2 states that the match semantics of a regex R remains unchanged when it is transformed into its equivalent Lookaround Normal Form (LNF). In other words, converting a regex into LNF preserves its matching behavior. The proof proceeds by induction on R, where we demonstrate that the normalization function behaves as expected for each case.

Theorem 3 (Decidability). *If $\mathcal{A}$ is decidable then so is* **ERE#** *modulo $\mathcal{A}$.*

Proof. Let $R \in$ **ERE#**. Then $\mathcal{M}(R) \neq \emptyset \Leftrightarrow \exists S \in \mathit{LNF}(R) : \mathcal{M}(S) \neq \emptyset$ holds by Theorem 2 and $\mathcal{M}(S) \neq \emptyset \Leftrightarrow \mathit{IsNonempty}(S)$ holds by Theorem 1, where $\mathit{IsNonempty}(S)$ uses *cleaning* which requires $\mathcal{A}$ to be decidable. ⊠

Decidability of **RE** extended with arbitrary lookaheads has been shown in [8]. Note that **ERE#** supports a restricted form of lookaheads, e.g., nested lookaheads are not supported and **ERE#** is not closed under concatenation. On the other hand, **ERE#** supports also restricted lookbehinds and is closed under reversal. Although we have provided a decision procedure for **ERE#**, an efficient symbolic decision procedure for the full class $\mathbf{ERE}_{\leq}$ remains an open problem.

4.4 Subsumption and Equivalence in ERE#

Let $R, S \in$ **ERE#**. Then S *subsumes* R or R *is subsumed by* S, denoted by $R \sqsubseteq S$ means that $\mathcal{M}(R) \subseteq \mathcal{M}(S)$. Subsumption is equivalent to *emptiness* of R `&` `~`S, since $\mathcal{M}(R \,\&\, \tilde{}S) = \emptyset \Leftrightarrow \mathcal{M}(R) \setminus \mathcal{M}(S) = \emptyset \Leftrightarrow \mathcal{M}(R) \subseteq \mathcal{M}(S)$.

One can also consider various *language semantics* of R such as *word*$(\mathcal{M}(R))$ and *match*$(\mathcal{M}(R))$ and related equivalence relations. In general, those are weaker than $\equiv$. E.g., $\mathcal{M}($`(?<=\Aa\z)`$) = \{(\mathtt{a}, \varepsilon, \varepsilon)\}$ and $\mathcal{M}($`(?=\Aa\z)`$) = \{(\varepsilon, \varepsilon, \mathtt{a})\}$ while the *word* language is $\{\mathtt{a}\}$ and the *match* language is $\{\varepsilon\}$ in both cases. The important special case is for regexes `\A`R`\z` whose language *match*$(\mathcal{M}($`\A`R`\z`$))$ corresponds exactly to $\mathcal{M}($`\A`R`\z`$)$.

One method to decide equivalence $R \equiv S$ is to decide that both $R \sqsubseteq S$ and $S \sqsubseteq R$ hold, that is emptiness of R `&` `~`S `|` S `&` `~`R. A different method is to use an additional Boolean operator for *symmetric difference* (XOR) $\oplus$ whose derivative rule in $\mathbf{ERE}_{\perp}$ would remain the same as for all the other binary operators:

$$\delta_\kappa(R \oplus S) = \delta_\kappa(R) \oplus \delta_\kappa(S) \qquad \mathit{Null}_\kappa(R \oplus S) \stackrel{\text{DEF}}{=} (\mathit{Null}_\kappa(R) \neq \mathit{Null}_\kappa(S))$$

Thus, *IsNonempty*$(R \oplus S) \Leftrightarrow R \not\equiv S$. While this works for $R, S \in \mathbf{ERE}_{\perp}$, it is unclear if this method generalizes to the whole **ERE#** in a "useful" manner, since *LNF*$(R \oplus S)$ would have to break the problem into separate disjuncts. The $\oplus$ operator also has a dual *XNOR* operator $\odot$ so that `~`$(R \oplus S) \equiv$ `~`$R \odot$ `~`S:

$$\delta_\kappa(R \odot S) = \delta_\kappa(R) \odot \delta_\kappa(S) \qquad \mathit{Null}_\kappa(R \odot S) \stackrel{\text{DEF}}{=} (\mathit{Null}_\kappa(R) = \mathit{Null}_\kappa(S))$$

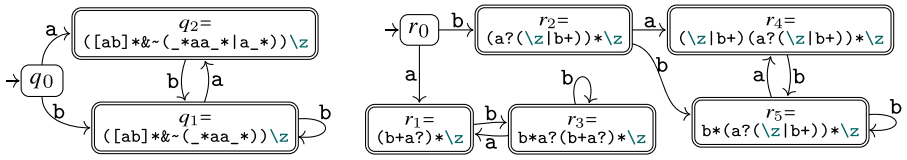
So $\oplus$ and $\odot$ can be used like any other binary operator in any regex.

Example 3. Given the regex q_0 = `\A([ab]+&~(_*aa_*))\z` and the regex r_0 = `\A(a(b+a?)*|b(a?(\z|b+))*)\z`, we consider their equivalence. Both regexes match all strings in `[ab]+` without `aa` as a substring. While q_0 is quite transparent in this regard, r_0 is less clear but avoids `&` and `~`. We prove their equivalence below, by using $\oplus$. We first describe their symbolic derivatives and resulting (symbolic) DFAs separately, as illustrated in Fig. 4. For example, for the regex q_0 we get that $\delta_{\text{INI}}(q_0) =$ **ite(**`a`**,** q_2**, ite(**`b`**,** q_1**,** $\perp$**))** $= \delta(q_1)$, and for the regex r_0 we get that $\delta_{\text{INI}}(r_0) =$ **ite(**`a`**,** r_1**, ite(**`b`**,** r_2**,** $\perp$**))** and $\delta(r_1) =$ **ite(**`b`**,** r_3**,** $\perp$**)**, etc., where the leaf regexes are shown as states in Figs. 4a and 4b. Now, returning to equivalence, the DFA for $r_0 \oplus q_0$ looks similar to Fig. 4b, e.g.,

$$\begin{aligned}\delta_{\text{INI}}(r_0 \oplus q_0) &= \textbf{ite(}\mathtt{a}\textbf{,}\ q_2\textbf{, ite(}\mathtt{b}\textbf{,}\ q_1\textbf{,}\ \perp\textbf{))} \oplus \textbf{ite(}\mathtt{a}\textbf{,}\ r_1\textbf{, ite(}\mathtt{b}\textbf{,}\ r_2\textbf{,}\ \perp\textbf{))} \\ &\equiv \textbf{ite(}\mathtt{a}\textbf{,}\ q_2 \oplus r_1\textbf{, ite(}\mathtt{b}\textbf{,}\ q_1 \oplus r_2\textbf{,}\ \perp\textbf{))}\end{aligned}$$

except that the DFA has no accepting states. E.g., $\mathit{Null}_{\text{FIN}}(q_2 \oplus r_1) =$ **false** because both q_2 and r_1 are nullable, similarly for all the other states. Thus $r_0 \equiv q_0$.

An algorithm for regex equivalence based on $\oplus$ would not need to construct a complete DFA for $\oplus$, in particular when a witness of inequivalence is not needed. Such an algorithm is ongoing work that falls outside the scope of this paper.



(a) DFA of q_0 (b) DFA of r_0

Fig. 4. q_0=`\A([ab]+&~(_*aa_*))\z` and r_0=`\A(a(b+a?)*|b(a?(\z|b+))*)\z`. The *sink* state $\perp$ is implicit – from each state s there is a symbolic transition to the sink state for all the other characters, e.g., $\delta(q_1)$[c] = $\perp$ and $\delta(r_1)$[a] = $\perp$.

5 Implementation

Our implementation of the decision procedures is done in the Rust programming language. The implementation is available on GitHub [45]. It supports a subset of the Unicode Strings theory in SMT-LIB [13], with a parser that converts smt2 format into our internal representation of regexes.

All the supported operations are implemented as operations on **ERE#**, and we support all the `RegLan` operations in the SMT-LIB standard. Operations that can not be represented as regexes in a straightforward manner such as `str.replace` are considered out of scope, as well as operations on a higher level of abstraction, such as the loop $R\{n\}$, where n is not a constant. In the formal semantics, the regex $R\{m,n\}$, where $0 < m \leq n$, stands for $R\{m\}\cdot(R|\varepsilon)\{n-m\}$.

The operations supported in the implementation include the following.

- All operations starting with `re.` on constants or other regexes, such as `re.++`, `re.union`, `re.*`, etc.
- **ERE** membership `str.in_re` on variables and constants
- String assertions that translate intuitively into **ERE**, such as:
  `str.len` x `<` 5 → `_{0,4}`; `str.prefixof"abc"` x → `abc_*`; `str.contains "abc"` x → `_*abc_*`
- Boolean operations such as `and`, `or`, `not` on any of the above, e.g.,
  `(str.len` x `>` 3`) and (str.len` x `<` 9`)` → `_{4,}&_{0,8}` (≡ `_{4,8}`)
- Standard conversions and utilities such as `str.to_int`, `str.to_re`, `str.++`

5.1 Representation of Regexes

In the implementation, each regex is represented as a tree structure of nodes, where each node has:

1. a unique identifier (unsigned 32-bit integer)
2. a kind specifier (e.g., predicate, union, concatenation, etc.)
3. the unique identifiers of the left and right child (or 0 if missing)
4. various additional information (e.g., bit-flags, and relevant character set)

Data stored as additional information includes a compact integer encoding of a predicate φ_r for each node r that approximates the relevant characters in r, as defined in [46, Section 5.3] with related rewrites. Additionally, r maintains several bit-flags that can be inferred during construction, such as whether the language contains the empty string (i.e., r is nullable), the presence of anchors in r, or whether r contains extended operations such as `&` or `~`.

When working with UTF8 representation, the character EBA $\mathcal{A}$ is over *bytes* as Σ. The representation of valid UTF8 strings is the regex $R_{\texttt{UTF8}}$:

$$\texttt{([\x00-\x7F]|[\xC0-\xDF]}\beta\texttt{|[\xE0-\xEF]}\beta\texttt{\{2\}|[\xF0-\xF7]}\beta\texttt{\{3\})*}$$

where β = `[\x80-\xBF]` is the *continuation byte* predicate. When working with any regex $R \in \mathbf{ERE}_{\L}$, the intersection R `&` `\A`$R_{\texttt{UTF8}}$`\z` restricts the accepted word language to $\mathcal{L}(R_{\texttt{UTF8}})$ that is a proper subset of Σ^*.

The predicates ψ in $\mathcal{A}$ are represented as 256-bit bit-vectors, allowing for efficient constant-time Boolean operations. Each $i \in \Sigma$ corresponds to the i'th bit of ψ. $\mathcal{A}$ has trivial checks for $\psi \equiv \bot$ and for $\psi \equiv \_$ since $\bot = 0$ and $\_ = 2^{256}-1$.

To accelerate the bit-vector operations further, we perform the operations in parallel over the 64-bit integers using Single Instruction Multiple Data (`SIMD`) instructions available in modern CPUs, e.g., `AVX2` or `SSE4.2`.

We assign a unique identifier to each predicate, allowing us to represent the predicate of a node as a 32-bit integer instead of a 256-bit bit-vector. This reduces the maximum number of unique predicates in a given regex to 2^{32}, but in practice this is a reasonable trade-off, as it is very unlikely that a regex would contain more than a thousand unique character sets, let alone 2^{32}.

The identifier of the node represents the syntactic structure of the regex, and is used to memoize the results of the operations on the regexes. The integers are assigned sequentially during construction, for example, the regex `(a|b)` would be assigned the integer 3, and point to the corresponding leaf nodes 1 and 2, representing the characters `a` and `b` respectively. To prevent duplicate identifiers, we use a hash map to store the regexes that have been constructed so far.

While not strictly relevant to the theory, we also have several rules for normalization, such as always having the left child of a union or intersection operation be the smaller identifier. Thus, the regexes `(a|b)` and `(b|a)` would be normalized to the same regex, which ever is constructed first. We also enforce that concatenations can only contain inner concatenations on the right to prevent ambiguous nesting. Together, these rules ensure that the syntactic construction of the regex is unique up to commutativity and associativity, but this does not prevent the construction of equivalent regexes in terms of the language they represent. For example, `(`ε`|b){50}` and `(b{0,25}){2}` represent the same language, but are constructed as different regexes – to detect equivalence between such examples, we would need to use the equivalence decision procedure.

5.2 Rewrite Rules and Simplifications

An important part of the implementation is the simplification and memoization of regexes during construction. The simplifications are applied bottom-up,

i.e., when a regex is constructed, the children of the regex are already simplified. The simplifications are based on the rules in [46], but in a more general way, as, in addition to the algebraic rules, we can perform language-level operations on the regexes directly.

For example, the regex `_*a_*b_*|_*b_*` would be simplified to `_*b_*` during construction, as the left branch of the union is subsumed by the right branch. In other words, the language of `_*a_*b_*` is a subset of the language of `_*b_*`. Subsumption plays a crucial role in the construction of intersections and unions, as it allows us to prevent construction of redundant definitions, often considerably reducing the size of the regex. We can also check satisfiability of a regex during construction, which allows us to rewrite unsatisfiable regexes to $\bot$, which represents the empty language.

For the key rewrite rules on the *match semantics* level, which are used in the implementation, see Fig. 5.

$$\frac{R}{\bot}(\mathcal{M}(R)=\emptyset) \quad \frac{R}{\texttt{\_*}}(\mathcal{M}(R)=\textbf{Span}) \quad \frac{R_1\,|\,R_2}{R_2}(\mathcal{M}(R_1)\subseteq\mathcal{M}(R_2)) \quad \frac{R_1\&R_2}{R_1}(\mathcal{M}(R_1)\subseteq\mathcal{M}(R_2))$$

Fig. 5. Main rewrite rules at the level of *match semantics*.

Performing these operations during construction gives us a very important property for optimization, which is that any remaining regex after construction is either $\bot$ or satisfiable. This allows us to skip many unnecessary satisfiability checks during the decision procedure.

This satisfiability check (leftmost rule in Fig. 5) only needs to be performed when constructing a concatenation with an anchor, an intersection, or complement, as these operations can result in an unsatisfiable regex, given that the children of the operation are satisfiable. The other operations will always result in a satisfiable regex, which we propagate as bit-flags in the implementation.

While the construction of a concatenation node with a satisfiable node and an anchor may result in an unsatisfiable regex, this satisfiability check takes constant time, as it only requires to check whether the language of the other child of the concatenation contains the empty string, which is already known as a bit-flag during construction. For example, the regex `b\A` is trivially unsatisfiable, and rewritten to $\bot$, as the language of `b` does not contain the empty string, therefore the anchor `\A` cannot be at the beginning. Similarly, the regex `\zb` is rewritten to $\bot$, as the language of `b` does not contain the empty string, therefore the anchor `\z` cannot be at the end.

The second rule in Fig. 5 provides a more efficient way to check for satisfiability of the complement operation, as the only unsatisfiable complement is `~(_*)`. This means that we can check that a complement is satisfiable by ensuring that the complement body is not equivalent to `_*`, effectively searching for the existence of a non-accepting state in the body of the complement. While it is not more efficient asymptotically, it is slightly more efficient memory-wise, as we do

not need to construct derivatives of the (outer) complement node itself. Often, in practice, such non-accepting state is the very state we start from, skipping the check entirely.

In the solver implementation, we do not perform the satisfiability check for the intersection operation eagerly, as the tasks often consist of many intersections, and proving nonemptiness of a subset of intersections does not necessarily help us in proving nonemptiness of all the intersections combined. Instead, we label each regex with a bit-flag, that indicates whether the regex contains an intersection, and only perform the satisfiability check when requested. If the flag is missing, we can be sure that the regex is already known to be satisfiable.

The solver also includes several variants of rewrite rules that are applied to *bounded loops*. In particular, it uses the LOOP rule and the predicate subsumption rule from [46, Figure 6]. To summarize, the majority of time spent in the solver is proving nonemptiness of intersection, which is done lazily, while the rest of the operations are performed eagerly during construction.

6 Evaluation

The evaluation is based on existing SMT benchmarks. All regular expressions supported in SMT belong to the class **ERE** and therefore do not support anchors, i.e., in terms of $\mathbf{ERE}_{\perp}$ R can be treated as `\A`R`\z`. For language semantics in $\mathbf{ERE}_{\perp}$, anchors can also be systematically eliminated through preprocessing. We did not use such preprocessing here, because our evaluation did not require it.

We have collected a set of SMT benchmarks for regexes. The benchmarks consist of approximately 20 000 SMT-LIB files, which are a mix of satisfiability problems for regexes, such as equivalence, subsumption, and membership queries. The benchmarks include AutomatArk [21], Denghang and Sygus-qgen benchmark sets from the `QF_S` and `QF_SLIA` categories of the SMT-LIB [6] repository, as well as additional benchmarks from the repository [42] designed specifically around difficult regular expression problems.

We compared our implementation (**ERE#**-solver) with the following state-of-the-art tools: cvc5 [5], Z3 [49], Z3str3 [10] , Z3str4 [36] , Z3alpha [34] , OSTRICH [17], OSTRICH^{RECL} [27,28] and Z3-Noodler [19]. The benchmark results (when solved) are identical between **ERE#**-solver, cvc5, Z3 and Z3-Noodler. There are discrepancies in the results of some solvers that we have not yet investigated. Table 1 shows the number of unsolved problems for each benchmark category.

Our solver is a specialized tool for regex constraint solving and analysis, rather than a general purpose SMT solver. As a result, it only supports a subset of the SMT operations that the other tools support. Nevertheless, for the regex benchmarks, our solver consistently outperforms the other tools, see Fig. 1 in the introduction for the cactus plot of the time taken to solve the benchmarks, and Table 1 for the number of unsolved benchmarks.

For our solver, under 100 of the benchmarks took more than 0.01 s to solve, and none of the benchmarks took more than 0.1 s to solve. Many individual

Table 1. Unsolved problems by solver in the various benchmark categories, where **min** and **max** are highlighted in color. Timeout for each problem is **6 s**.

	Syg	Denghang	Aut	Ark	Bool	Date	Blowup	Passwd	Mem	Int	Sub	State	∑
Included	**343**	**999**	**15995**	**21**	**19**	**14**	**34**	**1907**	**55**	**100**	**22**	**19509**
ERE#-solver	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Z3-Noodler	**0**	1	41	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	42
OSTRICH	**0**	**0**	75	**0**	**0**	**0**	1	1	2	3	**0**	82
OSTR^{RECL}	**0**	26	76	**0**	**0**	**0**	**0**	1	1	3	1	108
cvc5	1	26	152	**2**	3	1	7	3	27	28	**0**	250
Z3	**0**	124	322	1	**0**	2	16	2	20	9	**8**	504
Z3alpha	**0**	126	210	1	**0**	**3**	18	1537	**55**	99	**8**	2057
Z3str4	**0**	3	58	1	5	1	17	**1907**	**55**	**100**	6	2153
Z3str3	**37**	**773**	**5353**	1	**14**	**0**	**20**	129	21	26	7	**6381**

benchmarks take very little time to solve for other solvers as well, e.g., cvc5 solves thousands of the benchmarks in less than 0.01s, and is generally closest to the time of our solver in Fig. 1. The high lower-bound of the time taken for OSTRICH is likely due to a cold start of the process, as it is a Java-based tool, but in terms of the number of benchmarks solved, it is the third best tool after our solver and Z3-Noodler. The key factors contributing to our solver's significant performance improvement are discussed in the following section.

Performance Analysis. The key differences between our solver and others are in the way in which we represent and solve the problem:

- Our solver contains the problem fully within the regex theory, and does not need to convert any part of the problem to a different theory, such as linear arithmetic. Several other solvers solve regex problems with a combination of regex and linear arithmetic, e.g., encoding length constraints [19].
- Our solver never constructs an automaton for the regex, neither as a DFA nor as an NFA. Instead, it performs the operations symbolically, and the nonemptiness check is done through lazy unfolding of the problem encoded in our regex theory. This is in contrast to many other solvers, which construct an automaton for the regex and perform operations on the automaton.
- Our solver explores *language complement* lazily through derivatives, which is not done in other derivative-based string solvers such as cvc5. Complement is often the bottleneck in equivalence and subsumption checks.

The time required for checking nonemptiness of a regex is *proportional to the number of derivatives constructed*, where each derivative is a deterministic memoized operation that is cached upon the first computation. This reduces the amount of work done to the number of *unique* derivatives reached.

SMT benchmarks for regexes typically fall into two categories: they are either solved within a few milliseconds or remain unsolved due to excessive memory

consumption, often caused by an exponential blowup in automaton size. However, our tool avoids the explicit automaton construction, which mitigates the memory usage and allows us to solve many of the benchmarks that other tools fail to solve.

It is important to note that even the lazy graphs constructed by unwinding the problem can grow exponentially in size, and we have to perform simplifications whenever possible to reduce the effect of such blowup. It is done by rewriting the regexes, both algebraically and on the language level, as described in Sect. 4.1.

Many benchmarks are specifically designed to expose the exponential blowup. However, they can be solved almost immediately by performing certain simplifications and short-circuiting. In the following we explain the simplifications and their effect on the benchmarks in more detail.

State Space Considerations. Although the graphs can grow exponentially in size, our solver has several optimizations to mitigate blowup. In many practical cases where other solvers run out of memory, our heuristics enable us to solve the problems much more efficiently. Here we highlight some key optimizations and heuristics demonstrated on problems from the set of collected benchmarks.

Many benchmarks where solvers run out of memory involve the use of large quantifiers, such as `str.len x > 1000` or `re.loop 100 100`. These come with an enormous growth of state space, but as we *do not construct an automaton*, and are not looking for the shortest solution, a depth-first traversal of the state graph is sufficient. Table 2 shows the impact of the approach on satisfiable benchmarks with large quantifiers. The benchmarks are measured by the number of derivatives constructed to reach a solution, with depth-first search significantly outperforming breadth-first search. E.g., depth-first search solves the last benchmark in Table 2 with 495 derivatives in less than 0.002 s, including parsing and pre-processing the problem itself, while breadth-first search takes about 0.725 s.

Table 2. Effect of lazy depth-first search on satisfiable problems

Satisfiability Problem	DFS derivatives	BFS derivatives
`(_*a_*){25}&(_*b_*){25}&_{0,50}`	120	6749
`(_*a_*){50}&(_*b_*){50}&_{0,100}`	245	47874
`(_*a_*){100}&(_*b_*){100}&_{0,200}`	495	358249

While it has an impressive effect on satisfiable problems with a large number of possible solutions, it brings limited advantage for unsatisfiable benchmarks, as search has to exhaust all possibilities before concluding that the regex is unsatisfiable. For unsatisfiable problems, early elimination of trivially unsatisfiable regexes is crucial, as it can reduce search time by several orders of magnitude.

A key simplification for unsatisfiable problems is to prioritize checking length constraints of intersections, as this can be done without taking derivatives and will often eliminate the need for further search.

For example, the regex `_{4000,5000}&_{8000,9000}` can be rewritten to $\bot$ in constant time, as there is no overlap between the minimum and maximum lengths of the two. If there is an overlap, the regex can be rewritten to the intersection of the two length constraints, as was shown in Sect. 5.

Another important heuristic is checking the emptiness of a regex in reverse. It is particularly useful for addressing a common pathological case where large state space is combined with an unsatisfiable suffix. For example, the satisfiability of the regex `_*b_{100}& _*a_{100}` can be checked for emptiness by reversing the regex to `_{100}b_* & _{100}a_*` and checking it for emptiness. The reversed approach simplifies the problem significantly, as illustrated in Table 3.

Table 3. Effect of solving unsat suffix problems in reverse

Satisfiability Problem	Rev. derivatives	Fwd. derivatives
`_*b_{10}&_*a_{10}& _{10,}abc_{10,}`	9	279
`_*b_{20}&_*a_{20}& _{20,}abc_{20,}`	19	11 833
`_*b_{30}&_*a_{30}& _{30,}abc_{30,}`	29	539 145
`_*b_{40}&_*a_{40}& _{40,}abc_{40,}`	39	5 000 000+

7 Proposal for SMT-LIB

We believe that a conservative extension of SMT-LIB to support `RegLan` = $\mathbf{ERE}_{\leq}$ would be noncontroversial. For positive and negative lookaheads, positive and negative lookbehinds, and *span membership* or *matching* the minimal extension could be as follows where the start anchor `re.A` and the end anchor `re.z` could be included as regex constants for convenience:

`(re.lb r)` $\stackrel{\text{DEF}}{=}$ `(?<=r)` `(re.nlb r)` $\stackrel{\text{DEF}}{=}$ `(?<!r)` `re.A` $\stackrel{\text{DEF}}{=}$ `(re.nlb re.allchar)`
`(re.la r)` $\stackrel{\text{DEF}}{=}$ `(?=r)` `(re.nla r)` $\stackrel{\text{DEF}}{=}$ `(?!r)` `re.z` $\stackrel{\text{DEF}}{=}$ `(re.nla re.allchar)`
`(str.matches u v s r)` $\stackrel{\text{DEF}}{=}$ $(u, v, s) \models r$

Regarding compatibility with **ERE**, `(str.in_re v r)` fully retains its original interpretation for $r \in \mathbf{ERE}$. It follows by easy induction over $r \in \mathbf{ERE}$ that $\forall u, v, s : (u, v, s) \models r \Leftrightarrow v \in \mathcal{L}(r)$, i.e., $v \in match(\mathcal{M}(r)) \Leftrightarrow v \in \mathcal{L}(r)$. Thus,

$\forall u, v, s \in$ `String`$, r \in \mathbf{ERE}$: `(str.matches u v s r)` $\Leftrightarrow$ `(str.in_re`v `r)`

Using lookarounds, one can support other standard anchors such as the *line* anchors `^` and `$` and the *wordborder* anchor `\b` as `(?<!\w)(?=\w)|(?<=\w)(?!\w)`. One can also express different match semantics like *POSIX match* semantics of $w = uvs \wedge (u, v, s) \models r$, where $|u|$ is minimal and $|v|$ maximal for $|u|$ – the *leftmost longest* match in w – as an SMT optimization problem.

It is also practical to support $\oplus$, say `re.xor`, and $\odot$, say `re.xnor`, natively, in order to avoid their indirect encodings in various algorithms, such as equivalence checking, since $r_1 \equiv r_2 \Leftrightarrow r_1 \oplus r_2 \equiv \bot$. Moreover, *reversal* of sequences and regexes is beneficial, as illustrated above. In particular, $(u, v, s) \models r \Leftrightarrow (s^{\mathrm{r}}, v^{\mathrm{r}}, u^{\mathrm{r}}) \models r^{\mathrm{r}}$.

Acknowledgement. E. Zhuchko is supported by the Estonian Research Council grant *Automata in Learning, Interaction and Concurrency* (PRG1210).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Almeida, M., Moreira, N., Reis, R.: Antimirov and Mosses's rewrite system revisited. Int. J. Found. Comput. Sci. **20**(4), 669–684 (2009). https://doi.org/10.1142/S0129054109006802
2. Almeida, M., Moreira, N., Reis, R.: Testing the equivalence of regular languages. J. Autom. Lang. Comb. **15**(1/2), 7–25 (2010). https://doi.org/10.25596/jalc-2010-007
3. Amadini, R.: A survey on string constraint solving. ACM Comput. Surv. **55**(2), 16:1–16:38 (2023). https://doi.org/10.1145/3484198
4. Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. Theor. Comput. Sci. **155**(2), 291–319 (1996). https://doi.org/10.1016/0304-3975(95)00182-4
5. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). https://www.SMT-LIB.org
7. Barrière, A., Pit-Claudel, C.: Linear matching of javascript regular expressions. Proc. ACM Program. Lang. **8** (2024). https://doi.org/10.1145/3656431
8. Berglund, M., van der Merwe, B., van Litsenborgh, S.: Regular expressions with lookahead. J. Univ. Comput. Sci. **27**(4), 324–340 (2021). https://doi.org/10.3897/jucs.66330
9. Berzish, M., et al.: Towards more efficient methods for solving regular-expression heavy string constraints. Theor. Comput. Sci. **943**, 50–72 (2023). https://doi.org/10.1016/j.tcs.2022.12.009
10. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: Stewart, D., Weissenbacher, G. (eds.) FMCAD, pp. 55–59. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102241
11. Berzish, M., et al.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) CAV. LNCS, vol. 12760, pp. 289–312. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_14
12. Bjørner, N., Fazekas, K.: On incremental pre-processing for SMT. In: Pientka, B., Tinelli, C. (eds.) Automated Deduction – CADE 29, pp. 41–60. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-38499-8_3

13. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB format for sequences and regular expressions. In: Fontaine, P., Goel, A. (eds.) SMT, pp. 76–86 (2012). https://doi.org/10.29007/w5m5
14. Blahoudek, F., et al.: Word equations in synergy with regular constraints. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods. LNCS, vol. 14000, pp. 403–423. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_23
15. Brzozowski, J.A.: Derivatives of regular expressions. JACM **11**, 481–494 (1964). https://doi.org/10.1145/321239.321249
16. Chen, T., et al.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498707
17. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019). https://doi.org/10.1145/3290362
18. Chen, Y., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síc, J.: Solving string constraints with lengths by stabilization. Proc. ACM Program. Lang. **7**(OOPSLA2), 2112–2141 (2023). https://doi.org/10.1145/3622872
19. Chen, Y., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síc, J.: Z3-noodler: an automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) TACAS. LNCS, vol. 14570, pp. 24–33. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_2
20. Chocholatý, D., et al.: Mata: a fast and simple finite automata library. In: Finkbeiner, B., Kovács, L. (eds.) TACAS. LNCS, vol. 14571, pp. 130–151. Springer (2024). https://doi.org/10.1007/978-3-031-57249-4_7
21. D'Antoni, L.: AutomatArk (2018). https://github.com/lorisdanto/automatark
22. D'Antoni, L., Veanes, M.: Automata modulo theories. Commun. ACM **64**(5), 86–95 (2021). https://doi.org/10.1145/3419404
23. Eriksson, B., Stjerna, A., Masellis, R.D., Rümmer, P., Sabelfeld, A.: Black ostrich: web application scanning with string solvers. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) CCS, pp. 549–563. ACM (2023). https://doi.org/10.1145/3576915.3616582
24. Gallant, A.: BurntSushi: rebar (2024). https://github.com/BurntSushi/rebar
25. Hague, M., Jez, A., Lin, A.W.: Parikh's theorem made symbolic. Proc. ACM Program. Lang. **8**(POPL), 1945–1977 (2024). https://doi.org/10.1145/3632907
26. Hojjat, H., Rümmer, P., Shamakhi, A.: On strings in software model checking. In: Lin, A.W. (ed.) APLAS. LNCS, vol. 11893, pp. 19–30. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_2
27. Hu, D., Wu, Z.: String constraints with regex-counting and string-length solved more efficiently. In: Hermanns, H., Sun, J., Bu, L. (eds.) SETTA. LNCS, vol. 14464, pp. 1–20. Springer (2023). https://doi.org/10.1007/978-981-99-8664-4_1
28. Hu, D., Wu, Z.: An efficient string solver for string constraints with regex-counting and string-length. J. Syst. Archit., 1–38 (2025). https://doi.org/10.1016/j.sysarc.2025.103340
29. Jez, A., Lin, A.W., Markgraf, O., Rümmer, P.: Decision procedures for sequence theories. In: Enea, C., Lal, A. (eds.) CAV. LNCS, vol. 13965, pp. 18–40. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_2
30. Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: Ryu, S. (ed.) APLAS. LNCS, vol. 11275, pp. 350–372. Springer (2018). https://doi.org/10.1007/978-3-030-02768-1_19

31. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Formal Methods Syst. Des. **48**(3), 206–234 (2016). https://doi.org/10.1007/s10703-016-0247-6
32. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.W.: A decision procedure for regular membership and length constraints over unbounded strings. In: Lutz, C., Ranise, S. (eds.) FroCoS. LNCS, vol. 9322, pp. 135–150. Springer (2015). https://doi.org/10.1007/978-3-319-24246-0_9
33. Lotz, K., et al.: Solving string constraints using SAT. In: Enea, C., Lal, A. (eds.) CAV. LNCS, vol. 13965, pp. 187–208. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_9
34. Lu, Z., Siemer, S., Jha, P., Day, J.D., Manea, F., Ganesh, V.: Layered and staged Monte Carlo tree search for SMT strategy synthesis. In: IJCAI, pp. 1907–1915. ijcai.org (2024). https://doi.org/10.24963/ijcai.2024/211
35. Mamouras, K., Chattopadhyay, A.: Efficient matching of regular expressions with lookaround assertions. Proc. ACM Program. Lang. **8**(POPL), 2761–2791 (2024). https://doi.org/10.1145/3632934
36. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: a multi-armed string solver. In: FM, pp. 389–406. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-030-90870-6_21
37. Moreira, N., Pereira, D., de Sousa, S.M.: Deciding regular expressions (in-)equivalence in CoQ. In: Kahl, W., Griffin, T.G. (eds.) RAMiCS. LNCS, vol. 7560, pp. 98–113. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33314-9_7
38. Moseley, D., et al.: Derivative based nonbacktracking real-world regex matching with backtracking semantics. Proc. ACM Program. Lang. **7**(PLDI), 148:1–148:24 (2023). https://doi.org/10.1145/3591262
39. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
40. Owens, S., Reppy, J.H., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. **19**(2), 173–190 (2009). https://doi.org/10.1017/S0956796808007090
41. Reynolds, A., Woo, M., Barrett, C.W., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kuncak, V. (eds.) CAV. LNCS, vol. 10427, pp. 453–474. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_24
42. Stanford, C.: Boolean Regular Expression SMT Benchmarks (2021). https://github.com/cdstanford/regex-smt-benchmarks
43. Stanford, C., Veanes, M., Bjørner, N.S.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Freund, S.N., Yahav, E. (eds.) PLDI, pp. 620–635. ACM (2021). https://doi.org/10.1145/3453483.3454066
44. Turoňová, L., Holík, L., Homoliak, I., Lengál, O., Veanes, M., Vojnar, T.: Counting in regexes considered harmful: exposing ReDoS vulnerability of nonbacktracking matchers. In: 31st USENIX Security Symposium (USENIX Security 2022), pp. 4165–4182. USENIX Association, Boston (2022). https://www.usenix.org/conference/usenixsecurity22/presentation/turonova
45. Varatalu, I.E.: Implementation of RE#-solver (2025). https://github.com/ieviev/cav25-resharp-smt
46. Varatalu, I.E., Veanes, M., Ernits, J.: RE#: high performance derivative-based regex matching with intersection, complement, and restricted lookarounds. Proc. ACM Program. Lang. **9**(POPL), 1:1–1:32 (2025). https://doi.org/10.1145/3704837

47. Varatalu, I.E., Zhuchko, E., Ernits, J.: Artifact for regex decision procedures in extended RE# (2025). https://doi.org/10.5281/zenodo.15210805
48. Veanes, M., Ball, T., Ebner, G., Zhuchko, E.: Symbolic automata: Omega-regularity modulo theories. Proc. ACM Program. Lang. **9**(POPL), 2:1–2:34 (2025). https://doi.org/10.1145/3704838
49. Z3 Prover. https://github.com/Z3Prover/z3/wiki
50. Zhuchko, E., Veanes, M., Ebner, G.: Lean formalization of extended regular expression matching with lookarounds. In: Timany, A., Traytel, D., Pientka, B., Blazy, S. (eds.) CPP, pp. 118–131. ACM (2024). https://doi.org/10.1145/3636501.3636959