RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds

IAN ERIK VARATALU, Tallinn University of Technology, Estonia
MARGUS VEANES, Microsoft Research, USA
JUHAN ERNITS, Tallinn University of Technology, Estonia

We present a tool and theory **RE#** for regular expression matching that is built on symbolic derivatives, does not use backtracking, and, in addition to the classical operators, also supports complement, intersection and restricted lookarounds. We develop the theory formally and show that the main matching algorithm has *input-linear* complexity both in theory as well as experimentally. We apply thorough evaluation on popular benchmarks that show that **RE#** is *over 71% faster than the next fastest regex engine in Rust* on the baseline, and *outperforms all state-of-the-art engines on extensions of the benchmarks often by several orders of magnitude.*

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

Additional Key Words and Phrases: regex, derivative, automata, POSIX

ACM Reference Format:
Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proc. ACM Program. Lang.* 9, POPL, Article 1 (January 2025), 32 pages. https://doi.org/10.1145/3704837

1 Introduction

In the seminal paper [Thompson 1968] Thompson describes his regular expression search algorithm for *standard* regular expressions at the high level as follows,

> "In the terms of Brzozowski, this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched."

citing Brzozowski's work [Brzozowski 1964] from four years earlier. Thompson's algorithm compiles regular expressions into a very efficient form of *automata* and *has stood the test of time*: its variants today constitute the core of many state-of-the-art industrial *nonbacktracking* regular expression engines such as *RE2* [Cox 2010; Google 2024] and the regex engine of *Rust* [Rust 2024]. Earlier automata based classical algorithms for regular expression matching include [McNaughton and Yamada 1960] and [Glushkov 1961], a variant of the latter is used in *Hyperscan* [Wang et al. 2019].

Thompson's algorithm as well as Glushkov's construction have, by virtue of their efficiency for the *standard* or *classical* subset, to some degree, influenced how regular expression features have evolved over the past decades. The standard fragment allows only *union* (|) as a Boolean operator, and, unfortunately, neither *intersection* (&) nor *complement* (~) ever made it into the official notation, not even as reserved operators. Recently [Mamouras and Chattopadhyay 2024] presented a new algorithm for matching *lookarounds* with oracle NFAs and [Barrière and Pit-Claudel 2024] presented

Authors' Contact Information: Ian Erik Varatalu, Tallinn University of Technology, Tallinn, Estonia, ian.varatalu@taltech.ee; Margus Veanes, Microsoft Research, Redmond, USA, margus@microsoft.com; Juhan Ernits, Tallinn University of Technology, Tallinn, Estonia, juhan.ernits@taltech.ee.

ACM 2475-1421/2025/1-ART1
https://doi.org/10.1145/3704837

a new algorithm with an implementation for matching JavaScript regular expressions with full support for lookarounds *without backtracking* in *linear time*. Prior to these new algorithms, the standard fragment (with *anchors*), has been considered more-or-less as the only feasible and safe fragment of regular expressions for which matching can be performed reliably *without backtracking* in *input-linear* time.

> *The broad goal of this paper is to break this decades long belief that nonbacktracking algorithms for matching are only viable for regular expressions without* & *and* ~.

Backtracking based matching [Spencer 1994], although much more general, is considered to be unsafe in security critical applications because backtracking may cause nonlinear search complexity that can expose catastrophic denial of service vulnerabilities [Davis 2019; Davis et al. 2018; OWASP 2024]. To increase expressivity of the standard fragment, extensions such as *unbounded positive and negative lookaheads* have been added that, with the exception of [Barrière and Pit-Claudel 2024] and the related contribution to Javascript V8, are only supported by some *backtracking* based regex backends. Let $\mathbf{RE}_{\leq}$ denote standard regexes with unrestricted lookarounds. Below we highlight some key similarities and differences between **RE#** and $\mathbf{RE}_{\leq}$. A typical example of a regex involving lookaheads is a *password filter* in $\mathbf{RE}_{\leq}$

```
(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[!-/])\S*
```

that checks the presence of at least one lowercase letter, one uppercase letter, one digit, and one special character in a string of non-white-space characters. If only the standard fragment is allowed then the size of an equivalent regex grows *factorially* as the number of such individual constraints is increased, and becomes not only unreadable and very difficult to formulate, but also infeasible. Using &, the above regular expression takes the following equivalent form in **RE#**, say P:

```
(.*[a-z].*)&(.*[A-Z].*)&(.*\d.*)&(.*[!-/].*)&\S*
```

Lookarounds provide additional expressivity that in some instances overlaps with, but is orthogonal to, & and ~. In **RE#**, regexes are *restricted* to a fragment also called **RE#** that, in a normalized form, correspond to regexes `(?<=`R_1`)`R_2`(?=`R_3`)`, where R_i do not contain lookarounds but may contain & and ~. This fragment enables an efficient derivative based implementation, while supporting *all* the regexes in the *rebar* benchkmark set that contains primarily *anchors* that could in all cases automatically be rewritten to an equivalent form in **RE#**.

For *match search* a typical further restriction in the case of P is that a password has a minimum and a maximum length, say between 8 and 12 characters. Match search of (hidden) stored passwords is a typical task of a *credential scanner* such as [Microsoft 2021a] that guards against security leaks. In **RE#** the corresponding regex then becomes P`&.{8,12}`. In $\mathbf{RE}_{\leq}$ the regex `.*` in the individual lookaheads would need to be replaced by `.{0,12}` to limit the scope of the lookaheads. The main pattern `\S*` can be replaced by `\S{8,12}`. More complicated additional constraints, such as, contains at least two digits `(.*\d.*\d.*)`, are more tricky to express as lookaheads. Overall, intersection and complement enable a better *separation of concerns*. However, lookarounds are often needed to establish additional *context* conditions for such search patterns. For example, that the match must occur in the first line `(?<=\A.*)`P`&.{8,12}`, where dot does not match the newline character.

On the other hand, $\mathbf{RE}_{\leq}$ includes common regexes such as `(?<=`R_1`)`R_2`(?=`R_3`)|(?<=`R_4`)`R_5`(?=`R_6`)`, where R_i are standard regexes without embedded lookarounds. Such regexes are currently not supported in **RE#**, although it is algorithmically possible, and part of future work, to extend **RE#** to this case. Regexes in $\mathbf{RE}_{\leq}$ that involve *nested* lookarounds are in general out of scope for **RE#**.

It was four decades after Thompson's work when [Owens et al. 2009] recognized that a key aspect of Brzozowski's work had been forgotten:

> "It easily supports extending the regular-expression operators with boolean operations, such as intersection and complement. Unfortunately, this technique has been lost in the sands of time and few computer scientists are aware of it."

Namely that derivatives provide an elegant algebraic framework to formulate matching in functional programming, *not only for standard regular expressions* but also supporting *intersection* and *complement*, and can moreover naturally support *large alphabets*. The first *industrial* implementation of derivatives for standard regexes in an imperative language (C#) materialized a decade later [Saarikivi et al. 2019] and was used for *credential scanning* [Microsoft 2021a] while preserving input-linear complexity of match search. This work was recently extended to support anchors and to maintain PCRE (backtracking) match semantics [Moseley et al. 2023] and is now part of the official release of .NET through the new `NonBacktracking` regular expression option, where one of the key contributions is a new formalization of derivatives that is based on *locations* in words rather than individual characters, which made it possible to support anchors using derivatives.

Here we build on the theory [Moseley et al. 2023] and extend it to include regular expressions that allow *all* of the Boolean operators, including *intersection* and *complement*, as well as any other Boolean operator that is convenient to use for the matching task at hand, such as e.g. *symmetric difference* (*XOR*). **RE#** also supports *a limited form of lookarounds* as a generalization of anchors.

To illustrate a combined use of many of the extended features, consider the regex

$$\underbrace{\texttt{(?<=author.*).*}}_{(1)} \;\&\; \underbrace{\texttt{~(.*and.*)}}_{(2)} \;\&\; \underbrace{\texttt{\textbackslash b\textbackslash w.*\textbackslash w\textbackslash b}}_{(3)}$$

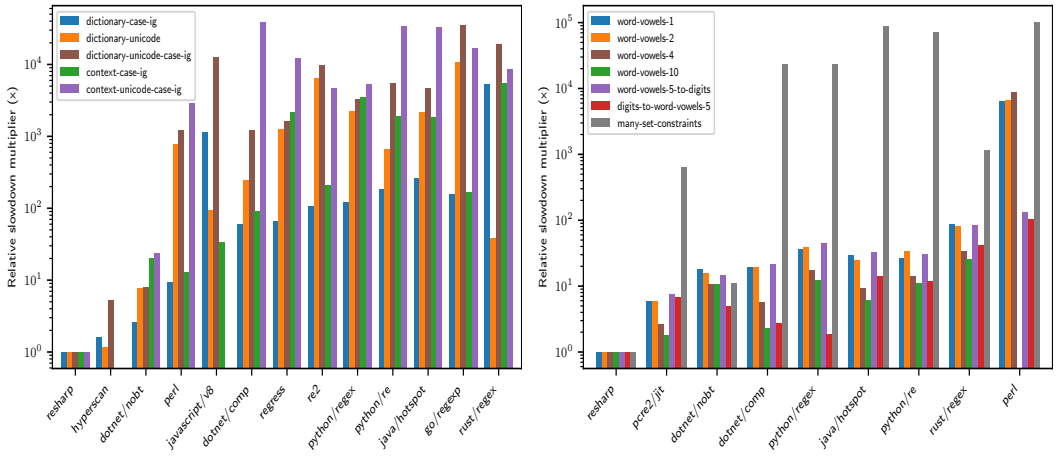
that matches all substrings in all *lines* (`.` matches any character except `\n`) that are: 1) preceded by `"author"` (via the lookbehind), 2) do not contain `"and"`, and 3) begin and end with *word letters* (`\w`) surrounded by *word boundaries* (`\b`). This regex finds all the authors in a bibtex text, such as the one shown in Figure 1, where all the match results are highlighted.

```
@article{ORT09,
  author = {Scott Owens and John H. Reppy and Aaron Turon},
  title  = {Regular-expression Derivatives Re-examined},
  journal= {J. Funct. Program.},
  year   = {2009},
}
```

Fig. 1. Sample bibtex entry.

We develop our theory formally and show that our matching algorithm is *input-linear* on single match search despite all the extensions. We have implemented the theory in a new tool **RE#** that is built on top of the open source codebase of .NET regular expressions [Microsoft 2022] where we have made use of several recent features available in .NET9, such as e.g. further SIMD vectorization of string matching functions and implementation of the Teddy [Qiu et al. 2021] algorithm. We show through comprehensive evaluation, using the *BurntSushi/rebar* benchmarking tool [Gallant 2024] evaluating engines with respect to finding *all matches*, that

- ✓ **Baseline**: **RE#** is not only on par with state-of-the-art matching engines on the benchmarks but places overall *first* in the summarized measurement (Section 6.1), *over 71% faster than the next fastest engine Rust* as shown in Table 4a ($2.54/1.48 \approx 1.716$). We explain the key aspects of each individual experiment.
- ✓ **Extensions**: We provide compelling experimental evidence on a new set of benchmarks involving many of the extended features that are either very difficult to express or fall outside the expressivity of existing tools. Here we want to draw attention to Figure 2 where **RE#** *outperforms all other engines and often by several orders of magnitude.*



(a) Monster benchmarks (Section 6.2.2). (b) Sets and Unicode benchmarks (Section 6.2.5).

Fig. 2. Two evaluations from Section 6. **RE#** is the baseline and y-axis is relative slowdown in *log* scale.

To a large extent, most optimizations rely heavily on the algebraic treatment of derivatives where regex based rewrite rules are available that would otherwise be very difficult to detect solely at the level of automata. Several of the optimizations that affect the baseline evaluation results in **RE#** are applicable also to the `NonBacktracking` engine of .NET. Some features of **RE#** were recently merged into .NET 9, as explained in [Toub 2024]. Further optimizations are possible in cases when the derivative based rewrite rules have the same semantics independent of whether the regex union operator is commutative (as in **RE#**) or not (as in .NET), e.g., for `IsMatch`.

Contributions. In summary, we consider the following as the main contributions of this paper:

- ✓ A new symbolic derivative based nonbacktracking matching algorithm for regular expression matching that supports *intersection*, *complement* and *restricted lookarounds* in regexes, with correctness theorem, while preserving *input linear* performance (Section 4).
- ✓ Explanation of the key techniques used in the implementation of **RE#** (Section 5).
- ✓ Extensive evaluation using a popular benchmarking tool that ranks **RE#** as fastest among all regex matchers today. All the evaluation results are explained in detail. (Section 6).

We start with some motivating examples for the extended operators that demonstrate how **RE#** can be used to match patterns that are currently either very difficult or infeasible to express with standard regular expressions.

2 Motivating Examples

In this section we explain the intuition behind the *intersection* (`&`) and *complement* (`~`) operators in **RE#** and illustrate their use through some examples. The examples are also available in the accompanying web application [Varatalu 2024b] and are written essentially in .NET regex syntax, with the addition of `&` and `~`, as well as the *wildcard* `_` that matches *all characters*, equivalently represented by `(.|\n)`, where dot denotes all characters *except* the newline character (`\n`).

The full syntax and semantics is explained in detail in Section 4. The overall intuition is, first of all, that matching of a regex R is relative to a *substring* of the input string. In particular, the wildcard `_*` matches all substrings of any input. For example, if the string is `" HelloWorld\n"` then the regex `(?<=\s)_*(?=\s)` matches only `"HelloWorld"` in it, because it is the only substring

surrounded by white space characters. The regex `e_*(?=\s)` matches the substring `"elloWorld"`, while `_*e_*(?=\s)` matches the substring `" HelloWorld"`.

For the upcoming examples, consult Table 1 for intuition on how to interpret the constructs.

Example 2.1 (Context-aware matching). Consider the input text in the first column below

```
- Valid
email@foo.com
email@subdomain.foo.com
email@other.com
- Invalid
email@-foo.com
email@foo@foo.com
```

```
- Valid
email@foo.com
email@subdomain.foo.com
email@other.com
- Invalid
email@-foo.com
email@foo@foo.com
- Valid
- Invalid
```

```
- Valid
email@foo.com
email@subdomain.foo.com
email@other.com
- Invalid
email@-foo.com
email@foo@foo.com
- Valid
- Invalid
```

which contains a list of valid email addresses and invalid email addresses. Our goal is not to *validate* the email addresses themselves, but to *extract* all the email addresses from the `Valid` section.

Finding the email addresses is easy, it is all lines that contain `@` which is expressed by the regex `.*@.*`, and since name and domain parts must be non-empty, we can refine the regex to `.+@.+`.

The `Valid` section requirement is more difficult, as we need a mechanism to distinguish between the two sections. This is where lookarounds come in handy. We can use a lookbehind to match the `Valid` section, and a lookahead to match the `Invalid` section, this ensures that the matches are between the two sections, using the regex `(?<=Valid_*).+@.+(?=_*Invalid)`.

However, this regex has a problem. If the input contains multiple `Valid` sections, the lookbehind will match the first `Valid` section and the lookahead will match the last `Invalid` section, as highlighted in the middle column above. But we want to match only the entries in `Valid` sections.

What we need is a way to define a window that starts with the `Valid` section and ends with the `Invalid` section. This is where *complement* comes in handy. What we really want to match is expressed with the regex `(?<=Valid~(_*Invalid_*)).+@.+` – the lookbehind requires $u{\cdot}$`Valid`$\cdot v$ for some u and v to occur before the match while prohibits `Invalid` from occurring in v.

We do not need the lookahead because the existence of the `Valid` section is enough to ensure that the match is in the `Valid` section. The complement then ensures that the `Invalid` section has not started yet, which works even if the `Invalid` section does not exist, which is exactly what we want, as highlighted by the matches of this regex in the third column above. An equivalent regex can be written as `(?<=Valid(?:(?<!Invalid)(?:.|\n))*)^.+@.+` which is not supported in **RE#** due to nested lookarounds, given `(?: )` denotes a non-capturing group. ⊠

Table 1. Basic constructs and their meaning in the extended regex syntax in **RE#**.

Lookarounds	*Prefixes/Suffixes*	*Other*
`(?<=`R`)_*` : preceded by R	R`_*` : starts with R	`_*`R`_*` : contains R
`(?<!`R`)_*` : not preceded by R	`~(`R`_*)` : does not start with R	`~(_*`R`_*)` : does not contain R
`_*(?=`R`)` : followed by R	`_*`R : ends with R	R`|`S : either R or S
`_*(?!`R`)` : not followed by R	`~(_*`R`)` : does not end with R	R`&`S : both R and S

Table 2. Real-world constraints expressed as regexes in **RE#**.

Real-world constraint	*Regex equivalent*	*Notes*
a line with an email	`^.*@.*$`	`^` $\stackrel{\text{DEF}}{=}$ `(?<=\A|\n)` and `$` $\stackrel{\text{DEF}}{=}$ `(?=\z|\n)`
in the `Valid` section	`(?<=Valid~(_*Invalid_*))_*`	see Example 2.1
without a subdomain	`.*@~((_*\._*){2})`	domain not containing two dots (`\.`)
not from `@other.com`	`~(_*@other.com)`	match not ending with `@other.com`

Table 3. Advanced constructs in the extended regex syntax of **RE#**.

	Regex	*Notes*
Difference	L `&~` R	L but not R (same as `~`R `&` L)
Implies (*Negated Difference*)	$L \Rightarrow R$	if L then R (same as `~`L `|` R)
XOR (*Symmetric Difference*)	$L \nLeftrightarrow R$	exactly one of L, R (same as L`&~`R`|~`L`&`R)
IFF/XNOR (*Implies Both Ways*)	$L \Leftrightarrow R$	both or none of L, R (same as L`&`R`|~`L`&~`R)
Window	`(?<=`L`~(_*`R`_*))_*`	in a window starting with L but not past R
Between	`(?<=`L`)~(_*`L`|`R`_*)(?=`R`)`	between L and R without crossing boundaries

Example 2.2 (Separation of concerns). What if we have more requirements for the email addresses? What if we want to exclude email addresses that contain a subdomain. What if we want to exclude emails from the `other.com` domain? In real-world applications, it is common to have multiple requirements for a match, and it is important to be able to express these requirements in a concise, maintainable way. All of these requirements can be expressed as regex constraints, as shown in Table 2 where `^` and `$` are called *line anchors*. The precise specification of what we want to match in this example is the following intersection of individual constraints:

$$\underbrace{\texttt{(\^{}.*@.*\$)}}_{\textit{single line}} \;\&\; \underbrace{\texttt{((?<=Valid\~{}(\_*Invalid\_*))\_*)}}_{\textit{occurs in the } \texttt{Valid} \textit{ section}} \;\&\; \underbrace{\texttt{(.*@\~{}((\_*\textbackslash.\_*)\{2\}))}}_{\textit{without a subdomain}} \;\&\; \underbrace{\texttt{(\~{}(\_*@other.com))}}_{\textit{not from } \texttt{other.com} \textit{ domain}}$$

This regex is easy to read and understand in individual components, and can be easily modified to add or remove requirements. The only match for it in the Example 2.1 text is `email@foo.com`. ⊠

Example 2.3 (Extended expressivity but not at the cost of performance). In industrial applications, regexes with unbounded lookarounds such as `(?<=Valid.*).+@.+` do not exist for a reason. Unbounded lookbehinds are not supported by many popular backtracking regex engines, such as PCRE and even the ones that do support them, such as .NET, cannot match them with good performance because the engine has to repeatedly backtrack for the context conditions for every potential match. This reduces the impact of having such features available, as the performance is too poor to make it feasible for use in practice.

This is where **RE#** shines. It is able to match extended regexes not only in linear time, but with performance that is comparable to industrial automata based engines, such as RE2, Hyperscan, and Rust. This is due to the fact that **RE#** is internally also automata based and does not backtrack.

The addition of intersection and complement does not increase the complexity of the matching algorithm relative to the input – the overall complexity of the engine remains *input-linear*. This includes not only lookarounds but allows also for more advanced constructs shown in Table 3.

Furthermore, lookbehind assertions for context such as the ones shown in Example 2.1 can be used to find not just the first match, but *all matches* in linear time, which is due to the fact that the engine is able to locate all the matches in a single pass over the input string. Note that certain combinations of regexes and inputs can still have an all-matches search complexity of $O(n^2)$ in regex engines that are otherwise guaranteed to be linear for a single match. Scenarios illustrating this are shown in Sections 6.1.4 and 6.2.6.

For an example supported by both backtracking engines and **RE#**, we are using the regex pattern `(?<=Valid[^-]*).+@.+` to illustrate this scenario in Figure 3, where a new section is known to start with the dash (`-`) character, which is not used anywhere else in the input.

The input is similar to the one used in Example 2.1, but here the x axis shows the number of lines with email addresses that the `Valid` and `Invalid` sections contain, and the y axis shows how many times the engine is slower than **RE#** that has a constant throughput in all cases. ⊠

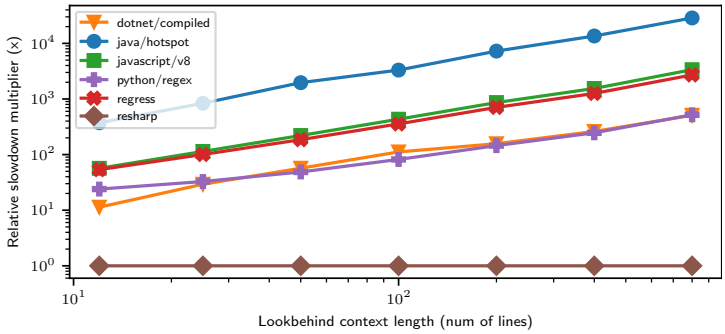


Fig. 3. All matches in *linear* time in **RE#**. Both axes are logarithmic. (We have *not evaluated* the recent linear implementation in Javascript V8 [Barrière and Pit-Claudel 2024].)

We now proceed to establish the theory for regular expressions extended with lookarounds, complement and intersection. We first recall some background material.

3 Preliminaries

Here we introduce the notation and main concepts used in the paper. The general meta-notation and notation for denoting strings and locations follows [Moseley et al. 2023] but differs in some minor aspects. We write $lhs \stackrel{\text{DEF}}{=} rhs$ to let lhs be *equal by definition* to rhs. Let $\mathbb{B} = \{\textbf{false}, \textbf{true}\}$ denote Boolean values, let $\langle x, y\rangle$ stand for pairs with $\langle x, y\rangle_1 \stackrel{\text{DEF}}{=} x$ and $\langle x, y\rangle_2 \stackrel{\text{DEF}}{=} y$.

Let Σ be a domain of *characters* and let Σ^* denote the set of all strings over Σ. We write ϵ for the empty string. The length of $s \in \Sigma^*$ is denoted by $|s|$. Strings of length one are treated as characters. Let i and l be nonnegative integers such that $i + l \leq |s|$. Then $s_{i,l}$ denotes the substring of s starting from index i having length l, where the first character has index 0. In particular $s_{i,0} = \epsilon$. For $0 \leq i < |s|$ let $s_i \stackrel{\text{DEF}}{=} s_{i,1}$ and let $s_{|s|} \stackrel{\text{DEF}}{=} \epsilon$. E.g., $\texttt{"abcdef"}_{1,4} = \texttt{"bcde"}$ and $\texttt{"abcde"}_5 = \epsilon$.

We let s^{r} denote the *reverse* of s, i.e., $s^{\mathrm{r}}_i = s_{|s|-1-i}$ for $0 \leq i < |s|$.

Let $s \in \Sigma^*$. A *location in* s is a pair $s[i] \stackrel{\text{DEF}}{=} \langle s, i\rangle$, where $0 \leq i \leq |s|$, where $s[0]$ is called *initial* and $s[|s|]$ *final*. The set of all locations in s is $\mathbf{Loc}(s)$ and $\mathbf{Loc} \stackrel{\text{DEF}}{=} \bigcup_{s\in\Sigma^*} \mathbf{Loc}(s)$. $\mathbf{Loc}^+$ stands for all the *nonfinal* locations. For $s[i] \in \mathbf{Loc}^+$ let $s[i] + 1 \stackrel{\text{DEF}}{=} s[i+1]$. Let also $hd(s[i]) \stackrel{\text{DEF}}{=} s_i$, i.e., if x is nonfinal then $hd(x)$ is the next(current) character of the location.

The *reverse* $s[i]^{\mathrm{r}}$ of a location $s[i]$ in s is the location $s^{\mathrm{r}}[|s|-i]$ in s^{r}. For example, the reverse of the final location in s is the initial location in s^{r}.

Effective Boolean Algebra. The tuple $\mathcal{A} = (\Sigma, \Psi, [\![\cdot]\!], \bot, \top, \vee, \wedge, \neg)$ is called an *Effective Boolean Algebra over* Σ or *EBA* [D'Antoni and Veanes 2021] where Ψ is a set of *predicates* that is closed under the Boolean connectives; $[\![\cdot]\!] : \Psi \longrightarrow 2^\Sigma$ is a *denotation function*; $\bot, \top \in \Psi$; $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \Sigma$, and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \Sigma \setminus [\![\varphi]\!]$. Two predicates ϕ and ψ are *equivalent* when $[\![\phi]\!] = [\![\psi]\!]$, denoted by $\phi \equiv \psi$. If $\varphi \not\equiv \bot$ then φ is *satisfiable*.

In examples Σ stands for the standard 16-bit character set of Unicode (also known as the *Basic Multilingual Plane* or *Plane 0*) and use the .NET syntax [Microsoft 2021b] of regular expression character classes. E.g., `\w` denotes all the *word-letters*, $[\![\texttt{\textbackslash W}]\!] = [\![\neg\texttt{\textbackslash w}]\!]$, `[0-9]` denotes all the *Latin numerals*, `\d` denotes all the *decimal digits*, and $[\![\,.\,]\!] = [\![\neg\texttt{\textbackslash n}]\!]$ (i.e., all characters other than the newline character).[1] $\top$ is `_` in **RE#** and corresponds to `[\s\S]` and $\bot$ corresponds to `[0-[0]]`.

[1]Note that in .NET $[\![\texttt{[0-9]}]\!] \subsetneq [\![\texttt{\textbackslash d}]\!]$ while for example in *JavaScript* $[\![\texttt{[0-9]}]\!] = [\![\texttt{\textbackslash d}]\!]$.

4 Regexes with Lookarounds and Location Derivatives

Here we formally define regexes supported in **RE#**. Regexes are defined modulo a character theory $\mathcal{A} = (\Sigma, \Psi, [\![\cdot]\!], \bot, \_, \vee, \wedge, \neg)$ that we illustrate with standard (.NET Regex) character classes in examples while the actual representation of character classes in Ψ is immaterial and Σ may even be *infinite*. $\mathcal{A}$ that is used for character classes in .NET is a *K-bit bitvector algebra* [Moseley et al. 2023, Section 5.1] using *mintermization* for compression. In most cases $K \leq 64$ and Ψ represents predicates using unsigned 64-bit integers or `UInt64` where all the Boolean operations are essentially $O(1)$ operations: bitwise-AND, bitwise-OR, and bitwise-NOT, with $\bot = 0$.

After the definition of **RE#** regexes we define their match semantics, that is based on pairs of locations called *spans* – the intuition is that a span $\langle s[i], s[j]\rangle$, where $i \leq j$, provides a match in s where the matching substring is $s_{i,j-i}$. Thereafter we formally define *derivatives* for **RE#**, develop the main matching algorithm for **RE#** and prove its correctness and input linearity.

RE# grew out of our initial work of the more general theory of $\mathbf{ERE}_{\leq}$ [Varatalu et al. 2023] that was subsequently formalized and proved correct in *Lean* [Zhuchko et al. 2024] (where **RE** stands for $\mathbf{ERE}_{\leq}$). A matching algorithm for full $\mathbf{ERE}_{\leq}$ turned out be highly nonlinear and had unreliable performance. It was also very challenging to implement optimizations in $\mathbf{ERE}_{\leq}$ while maintaining its correctness. However, the Lean formalization of $\mathbf{ERE}_{\leq}$ is *executable* and we have used it, both as an oracle during testing, as well as to prove correctness of certain optimizations that were critical for **RE#**, e.g., *elimination of negative lookarounds*. We start by introducing $\mathbf{ERE}_{\leq}$ that subsumes **RE#**. $\mathbf{ERE}_{\leq}$ also subsumes $\mathbf{RE}_{\leq}$ that is the class **RE** of standard regexes extended with *all* lookarounds. Some results make use of the formalized theorems in [Zhuchko et al. 2024] of the theory of $\mathbf{ERE}_{\leq}$.

4.1 Full Class ERE with Lookarounds

The class $\mathbf{ERE}_{\leq}$ of regexes 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* (`~`) binds strongest. Let $\psi \in \Psi$ and let m be a positive integer.

$$\mathbf{R} \quad ::= \quad \psi \mid \varepsilon \mid \mathbf{R}_1\mathbf{|}\mathbf{R}_2 \mid \mathbf{R}_1\mathbf{\&}\mathbf{R}_2 \mid \mathbf{R}_1\mathbf{\cdot}\mathbf{R}_2 \mid \mathbf{R}\{m\} \mid \mathbf{R}\mathbf{*} \mid \mathbf{\sim R} \mid \mathbf{(?<=R)} \mid \mathbf{(?<!R)} \mid \mathbf{(?=R)} \mid \mathbf{(?!R)}$$

We also write `()` for the *empty word* regex ε. The regex denoting *nothing* is just the predicate $\bot$. We let $\mathbf{R}\{0\} \stackrel{\text{DEF}}{=} \varepsilon$ for convenience. In reality also $\mathbf{R}\{1\} \stackrel{\text{DEF}}{=} \mathbf{R}$. We write $R\mathbf{+}$ for $R\mathbf{\cdot}R\mathbf{*}$. The union operator is also called *alternation*. Let $\mathbf{RE}_{\leq}$ denote $\mathbf{ERE}_{\leq}$ without intersection and complement.

The regexes `(?=R)`, `(?!R)`, `(?<=R)`, and `(?<!R)` are called *lookarounds*; `(?=R)` is *(positive) lookahead*, `(?!R)` is *negative lookahead*, `(?<=R)` is *(positive) lookbehind*, and `(?<!R)` is *negative lookbehind*. In the context of $\mathbf{ERE}_{\leq}$ let `\A` $\stackrel{\text{DEF}}{=}$ `(?<!_)` and `\z` $\stackrel{\text{DEF}}{=}$ `(?!_)`.

4.2 Regexes Supported in RE#

Here we introduce the abstract syntax of regular expressions R that are supported by **RE#** that we also denote by **RE#**. First, regexes *without lookarounds* are below denoted by E and the corresponding subclass is denoted by **ERE**. Regexes R then extend E with lookarounds and are closed under intersection.

Let **RE** stand for the *standard* subset of **ERE** without `~` and `&`. Figure 4 illustrates the relationships between the regex classes.

$$E \quad ::= \quad \mathtt{\backslash A} \mid \mathtt{\backslash z} \mid \psi \mid \varepsilon \mid E_1\mathbf{|}E_2 \mid E_1\mathbf{\&}E_2 \mid E_1\mathbf{\cdot}E_2 \mid E\{m\} \mid E\mathbf{*} \mid \mathbf{\sim}E$$
$$R \quad ::= \quad E \mid R_1\mathbf{\&}R_2 \mid \mathbf{(?<=}E\mathbf{)\cdot}R \mid \mathbf{(?<!}E\mathbf{)\cdot}R \mid R\mathbf{\cdot(?=}E\mathbf{)} \mid R\mathbf{\cdot(?!}E\mathbf{)}$$

The regex `\A` is called the *start anchor* and `\z` is called the *end anchor*. While all other standard anchors supported in .NET are also supported in the concrete syntax of **RE#**, they are defined via lookarounds and (currently) disallowed

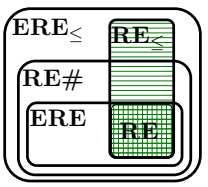


Fig. 4. Venn diagram of the regex classes: $\mathbf{RE} \subsetneq \mathbf{ERE} \subsetneq \mathbf{RE\#} \subsetneq \mathbf{ERE}_{\leq}$ and also $\mathbf{RE} \subsetneq \mathbf{RE}_{\leq} \subsetneq \mathbf{ERE}_{\leq}$.

in **ERE**. For example, the *line anchors* `^` and `$` are defined in Table 2. The presence of `\A` and `\z` as *primitive* regexes in **ERE** is used in the proof of Theorem 1.

Let also $E\{m,n\} \overset{\text{DEF}}{=} E\{m\}\cdot(E\,|\,\varepsilon)\{n-m\}$, where $0 \leq m \leq n$, as the *bounded loop* with *lower bound* m and *upper bound* n. In fact, the implementation in **RE#** uses $E\{m,n\}$ as the *core* construct, where $m = 0$ is an important special case during rewrites, and $E\{m\} \overset{\text{DEF}}{=} E\{m,m\}$ and $E* \overset{\text{DEF}}{=} E\{0,\infty\}$.

4.3 Match Semantics

The match semantics of regexes in $\mathbf{ERE}_{\leq}$ uses *spans*. A *span* in a string s is formally a pair of locations $\theta = \langle s[i], s[j]\rangle$ where $i \leq j$; the *width* of θ is $|\theta| \overset{\text{DEF}}{=} j - i$. We call θ_1 the *start location of* θ and θ_2 the *end location of* θ. If $|\theta| = 0$ then $\theta_1 = \theta_2$ is also called the *location of* θ. The set of all spans in s is denoted by $\mathbf{Span}(s)$ and $\mathbf{Span} \overset{\text{DEF}}{=} \bigcup_{s\in\Sigma^*} \mathbf{Span}(s)$. For all $\theta \in \mathbf{Span}$ and $R \in \mathbf{ERE}_{\leq}$, θ *is a match of* R or θ *models* R is denoted by $\theta \models R$:

$$\begin{array}{lll}
\theta \models \varepsilon & \overset{\text{DEF}}{=} & |\theta| = 0 \\
\theta \models \psi & \overset{\text{DEF}}{=} & |\theta| = 1 \textbf{ and } hd(\theta_1) \in [\![\psi]\!] \\
\theta \models L\,|\,R & \overset{\text{DEF}}{=} & \theta \models L \textbf{ or } \theta \models R \\
\theta \models L\&R & \overset{\text{DEF}}{=} & \theta \models L \textbf{ and } \theta \models R \\
\theta \models \sim R & \overset{\text{DEF}}{=} & \theta \not\models R \\
\theta \models R* & \overset{\text{DEF}}{=} & \exists\, m \geq 0 : \theta \models R\{m\}
\end{array}
\qquad
\begin{array}{lll}
\theta \models L\cdot R & \overset{\text{DEF}}{=} & \exists\, x : \langle \theta_1, x\rangle \models L \textbf{ and } \langle x, \theta_2\rangle \models R \\
\theta \models R\{m\} & \overset{\text{DEF}}{=} & \exists\, x : \langle \theta_1, x\rangle \models R \textbf{ and } \langle x, \theta_2\rangle \models R\{m-1\} \\
\theta \models (?{=}R) & \overset{\text{DEF}}{=} & |\theta| = 0 \textbf{ and } \exists\, x : \langle \theta_1, x\rangle \models R \\
\theta \models (?{!}R) & \overset{\text{DEF}}{=} & |\theta| = 0 \textbf{ and } \nexists\, x : \langle \theta_1, x\rangle \models R \\
\theta \models (?{<=}R) & \overset{\text{DEF}}{=} & |\theta| = 0 \textbf{ and } \exists\, x : \langle x, \theta_2\rangle \models R \\
\theta \models (?{<!}R) & \overset{\text{DEF}}{=} & |\theta| = 0 \textbf{ and } \nexists\, x : \langle x, \theta_2\rangle \models R
\end{array}$$

Intuitively, $\theta \models$ `(?=`R`)` means that there exists a match of R *starting* from the location of θ, and $\theta \models$ `(?<=`R`)` means that there exists a match of R *ending* in the location of θ. For any *location* x, we write $x \models R$ for $\langle x, x\rangle \models R$. For the **ERE** anchors `\A` and `\z` above we have thus that

$$x \models \texttt{\A} \Leftrightarrow \mathit{Initial}(x) \qquad x \models \texttt{\z} \Leftrightarrow \mathit{Final}(x)$$

Let B be a (positive) lookbehind and let A be a (positive) lookahead. Then it follows via the match semantics of concatenation that

$$\theta \models B\cdot R \Leftrightarrow \theta_1 \models B \textbf{ and } \theta \models R \qquad \theta \models R\cdot A \Leftrightarrow \theta_2 \models A \textbf{ and } \theta \models R$$

Example 4.1. Consider the first part `(?<=author.*).*` of the author search regex from the introduction. Then $\theta \models$ `(?<=author.*).*` implies that $\theta_1 \models$ `(?<=author.*)` and $\theta \models$ `.*`. So the *start* location θ_1 of the match must be after the string `"author"` and the matched substring itself must be on a single line (recall that $[\![\,.\,]\!] = \Sigma \setminus \{\texttt{\n}\}$). ⊠

Example 4.2. As a simple but nontrivial example of various negations, we compare the regex `(?<!\w)` with the regex `(?<=¬\w)` (or `(?<=\W)`). Let $s =$ `"a@b"`. Then $s[0] \models$ `(?<!\w)` because no word-letter precedes the initial location, but $s[0] \not\models$ `(?<=¬\w)` because no non-word-letter precedes the initial location. On the other hand both $s[2] \models$ `(?<!\w)` and $s[2] \models$ `(?<=¬\w)`. ⊠

For $R \in \mathbf{ERE}_{\leq}$ let $\mathcal{M}(R) \overset{\text{DEF}}{=} \{\theta \in \mathbf{Span} \mid \theta \models R\}$ and for $R, S \in \mathbf{ERE}_{\leq}$, $R \equiv S \overset{\text{DEF}}{=} \mathcal{M}(R) = \mathcal{M}(S)$. The implementation in **RE#** uses the following key property for normalisation of regexes in **RE#**. It also shows the key role that the start and end anchors play. We call the below normal form of $R \in$ **RE#** the *Lookaround Normal Form* of R and denote it by $\mathbf{LNF}(R)$ and the LNF of **RE#** by $\mathbf{LNF}(\mathbf{RE\#})$. Construction of $\mathbf{LNF}(R)$ is itself linear in the size of R.

Theorem 1 (LNF). *For all* $R \in$ **RE#** *there exist* $A, B, E \in$ **ERE** *such that* $R \equiv$ `(?<=`B`)`$\cdot E\cdot$`(?=`A`)`.

Proof. First, by [Zhuchko et al. 2024, Theorem 5], for all $S \in \mathbf{ERE}_{\leq}$, `(?!`$S$`)` $\equiv$ `(?=~(`$S\cdot$`_*)\z)` and `(?<!`S`)` $\equiv$ `(?<=\A·~(_*·`S`))`. If S is in **ERE** then so are `~(`$S\cdot$`_*)\z` and `\A·~(_*·`S`)`. *We thus replace all the negative lookarounds by positive ones in* R.

Any concatenation of two lookaheads $(?{=}A_1){\cdot}(?{=}A_2)$ is in $\mathbf{ERE}_{\leq}$ equivalent to the single lookahead $(?{=}(A_1{\cdot}\_*)\&(A_2{\cdot}\_*))$ where if $A_1, A_2 \in \mathbf{ERE}$ then so is $(A_1{\cdot}\_*)\&(A_2{\cdot}\_*)$. Analogously, for the case of lookbehinds, $(?{<=}B_1){\cdot}(?{<=}B_2) \equiv (?{<=}(\_*{\cdot}B_1)\&(\_*{\cdot}B_2))$.

Any intersection $(?{<=}B_1){\cdot}E_1{\cdot}(?{=}A_1)\&(?{<=}B_2){\cdot}E_2{\cdot}(?{=}A_2)$ is in $\mathbf{ERE}_{\leq}$ equivalent to the intersection $(?{<=}B_1){\cdot}(?{<=}B_2){\cdot}(E_1\&E_2){\cdot}(?{=}A_1){\cdot}(?{=}A_2)$ because, according to the formal semantics,

$$
\begin{aligned}
&\langle x, y\rangle \models (?{<=}B_1){\cdot}E_1{\cdot}(?{=}A_1)\&(?{<=}B_2){\cdot}E_2{\cdot}(?{=}A_2)\\
&\Leftrightarrow \langle x, y\rangle \models (?{<=}B_1){\cdot}E_1{\cdot}(?{=}A_1) \textbf{ and } \langle x, y\rangle \models (?{<=}B_2){\cdot}E_2{\cdot}(?{=}A_2)\\
&\Leftrightarrow x \models (?{<=}B_1) \textbf{ and } \langle x, y\rangle \models E_1 \textbf{ and } y \models (?{=}A_1) \textbf{ and } x \models (?{<=}B_2) \textbf{ and } \langle x, y\rangle \models E_2 \textbf{ and } y \models (?{=}A_2)\\
&\Leftrightarrow x \models (?{<=}B_1){\cdot}(?{<=}B_2) \textbf{ and } \langle x, y\rangle \models E_1\&E_2 \textbf{ and } y \models (?{=}A_1){\cdot}(?{=}A_2)
\end{aligned}
$$

We thus arrive at the lookaround normal form, by applying these rewrites. □

Example 4.3. Consider the author search regex from the introduction where the word border `\b` *before* `\w` corresponds to the negative lookbehind `(?<!\w)` and `\b` *after* `\w` corresponds to the negative lookahead `(?!\w)`. After normalisation the negative lookarounds have been replaced by the equivalent positive ones:

$$
\underbrace{(?{<=}\_*\texttt{author}.*\&\_*\texttt{\textbackslash A}\sim(\_*\texttt{\textbackslash w}))}_{\textit{lookbehind}}\cdot\underbrace{(.*\&\sim(.*\texttt{and}.*)\&\texttt{\textbackslash w}.*\texttt{\textbackslash w})}_{\textit{main pattern}}\cdot\underbrace{(?{=}\sim(\texttt{\textbackslash w}\_*)\texttt{\textbackslash z})}_{\textit{lookahead}}
$$

This regex is pretty much humanly unreadable and is only intended for internal processing by the matcher. Among several other simplifications, an immediate simplification that is applied here is that $(?{=}\sim(\psi\_*)\texttt{\textbackslash z}) \equiv (?{=}\neg\psi\,|\,\texttt{\textbackslash z})$ for all $\psi \in \Psi$ that gets rid of ~ and _*. ⊠

4.4 Reversal

Reversal of $R \in \mathbf{ERE}_{\leq}$, denoted by $R^{\mathbf{r}}$, is defined as follows:

$$
\begin{array}{llll}
\psi^{\mathbf{r}} \stackrel{\text{DEF}}{=} \psi & (R\,|\,S)^{\mathbf{r}} \stackrel{\text{DEF}}{=} R^{\mathbf{r}}\,|\,S^{\mathbf{r}} & (R{\cdot}S)^{\mathbf{r}} \stackrel{\text{DEF}}{=} S^{\mathbf{r}}{\cdot}R^{\mathbf{r}} & (?{<=}R)^{\mathbf{r}} \stackrel{\text{DEF}}{=} (?{=}R^{\mathbf{r}})\\
\varepsilon^{\mathbf{r}} \stackrel{\text{DEF}}{=} \varepsilon & (R\&S)^{\mathbf{r}} \stackrel{\text{DEF}}{=} R^{\mathbf{r}}\&S^{\mathbf{r}} & R\{m\}^{\mathbf{r}} \stackrel{\text{DEF}}{=} R^{\mathbf{r}}\{m\} & (?!R)^{\mathbf{r}} \stackrel{\text{DEF}}{=} (?{<}!R^{\mathbf{r}})\\
R{*}^{\mathbf{r}} \stackrel{\text{DEF}}{=} R^{\mathbf{r}}{*} & (\sim R)^{\mathbf{r}} \stackrel{\text{DEF}}{=} \sim(R^{\mathbf{r}}) & (?{=}R)^{\mathbf{r}} \stackrel{\text{DEF}}{=} (?{<=}R^{\mathbf{r}}) & (?{<}!R)^{\mathbf{r}} \stackrel{\text{DEF}}{=} (?!R^{\mathbf{r}})
\end{array}
$$

Reversal is used in the definition of the top-level matching algorithm, and is therefore a critical operation of the overall framework. It follows by induction over regexes that reversal is both size-preserving and involutive: $(R^{\mathbf{r}})^{\mathbf{r}} = R$.

The reverse of a span $\theta \in \mathbf{Span}$ is defined as the span $\theta^{\mathbf{r}} \stackrel{\text{DEF}}{=} \langle(\theta_2)^{\mathbf{r}}, (\theta_1)^{\mathbf{r}}\rangle$, that is also an involutive and width-preserving operation. It follows also that $\theta \in \mathbf{Span}(s) \Leftrightarrow \theta^{\mathbf{r}} \in \mathbf{Span}(s^{\mathbf{r}})$. We make use of the following theorem. Note that $\texttt{\textbackslash A}^{\mathbf{r}} = (?{<}!\_)^{\mathbf{r}} = (?!\_^{\mathbf{r}}) = (?!\_) = \texttt{\textbackslash z}$.

Theorem 2 (Reversal). *Let $R \in \mathbf{RE}\#$ and $\theta \in \mathbf{Span}$. Then $R^{\mathbf{r}} \in \mathbf{RE}\#$ and $\theta \models R \Leftrightarrow \theta^{\mathbf{r}} \models R^{\mathbf{r}}$.*

Proof. By using Theorem 1 let $R = (?{<=}B)E(?{=}A)$ where $A, B, E \in \mathbf{ERE}$ and observe that $\mathbf{ERE}$ is (by definition) closed under reversal. It follows that $R^{\mathbf{r}} = (?{<=}A^{\mathbf{r}})E^{\mathbf{r}}(?{=}B^{\mathbf{r}})$ is in $\mathbf{RE}\#$. The statement $\theta \models R \Leftrightarrow \theta^{\mathbf{r}} \models R^{\mathbf{r}}$ follows from [Zhuchko et al. 2024, Theorem 1] because $\mathbf{RE}\# \subseteq \mathbf{ERE}_{\leq}$. □

4.5 Nullability

Here we define *nullability* of regexes $R \in \mathbf{ERE}$. The definition is more-or-less standard with one key difference concerning the two anchors. In terms of the span based match semantics, R being *always* nullable means that R is equivalent to $R|\varepsilon$ and thus $x \models R$ for all locations x, i.e., that R

matches the empty word in any context. Let x be any location and let $\psi \in \Psi$.

$$\begin{array}{rcl@{\qquad}rcl}
Null_x(R|S) & \stackrel{\text{DEF}}{=} & Null_x(R) \textbf{ or } Null_x(S) & Null_x(\backslash\mathtt{A}) & \stackrel{\text{DEF}}{=} & Initial(x) \\
Null_x(R\&S) & \stackrel{\text{DEF}}{=} & Null_x(R) \textbf{ and } Null_x(S) & Null_x(\backslash\mathtt{z}) & \stackrel{\text{DEF}}{=} & Final(x) \\
Null_x(R{\cdot}S) & \stackrel{\text{DEF}}{=} & Null_x(R) \textbf{ and } Null_x(S) & Null_x(\varepsilon) & \stackrel{\text{DEF}}{=} & \textbf{true} \\
Null_x(R\{m\}) & \stackrel{\text{DEF}}{=} & Null_x(R) & Null_x(R*) & \stackrel{\text{DEF}}{=} & \textbf{true} \\
Null_x(\sim R) & \stackrel{\text{DEF}}{=} & \textbf{not } Null_x(R) & Null_x(\psi) & \stackrel{\text{DEF}}{=} & \textbf{false}
\end{array}$$

If a regex R in a lookaround `(?=`R`)` or `(?<=`R`)` is always nullable then the lookaround is simplified to ε. Such nullability status is maintained with each regex AST node at construction time. For example, the regex `\n\z|\z` is only nullable in a *final* location. The lookahead `(?=\n\z|\z)` corresponds to the `\Z` anchor in .NET and is also supported in the concrete syntax of **RE#**.

4.6 Lookaround Reductions in ERE with Lookarounds

Negative lookarounds can always be eliminated from regexes in $\mathbf{ERE}_{\leq}$ as well as **RE#**, by using [Zhuchko et al. 2024, Theorem 5]. For example, the negative lookahead `(?!a)` is replaced by the lookahead `(?=~(a_*)\z)` that is further simplified to the form `(?=[^a]|\z)`. For $R \in \mathbf{ERE}_{\leq}$ let $minlen(R)$ be the minimum $|\theta|$ such that $\theta \models R$ and let $maxlen(R)$ be the maximum $|\theta|$ such that $\theta \models R$ or ∞ if R is *unbounded*. A lookaround `(?=`R`)` or `(?<=`R`)` is *bounded* when $maxlen(R) \neq \infty$.[2]

The core intuition of the difference between $\mathbf{ERE}_{\leq}$ and **RE#** lies in that in **RE#** lookbehinds are only allowed to match context *before* the actual match and lookaheads *after* it. The reason for this restriction is that it allows for a well-defined match semantics and matching both the lookbehind and lookahead in *a single pass* over the input string. This single pass is crucial for the performance of the engine, as it does not impose a search-time penalty for using word boundaries or lookarounds, as is highlighted later in Section 6.1.2.

The two main rules that are used to eliminate some cases of *bounded* lookarounds in order to reduce some regexes in $\mathbf{ERE}_{\leq}$ (and thus from $\mathbf{RE}_{\leq}$) to **RE#** are presented in Figure 5. In the **RE#**

$$\textsc{la-elim}\ \frac{\mathtt{(?=}S\mathtt{)}R}{S\mathtt{\_*\&}R}\,(maxlen(S) \leq minlen(R)) \qquad \textsc{lb-elim}\ \frac{R\mathtt{(?<=}S\mathtt{)}}{R\mathtt{\&\_*}S}\,(maxlen(S) \leq minlen(R))$$

Fig. 5. Bounded lookaround elimination rules for $S, R \in \mathbf{ERE}_{\leq}$.

implementation the calculations of $minlen(S)$ and $maxlen(S)$ are safely approximated. Concerning the rules in Figure 5, observe that, if $maxlen(S) \leq minlen(R)$ then

$$\begin{array}{rl}
\langle s[i], s[j]\rangle \models \mathtt{(?=}S\mathtt{)}R \Leftrightarrow & \exists k : \langle s[i], s[k]\rangle \models S \textbf{ and } \langle s[i], s[j]\rangle \models R \\
\overset{\dagger}{\Leftrightarrow} & \exists k \leq j : \langle s[i], s[k]\rangle \models S \textbf{ and } \langle s[i], s[j]\rangle \models R \\
\Leftrightarrow & \langle s[i], s[j]\rangle \models S\mathtt{_*} \textbf{ and } \langle s[i], s[j]\rangle \models R \Leftrightarrow \langle s[i], s[j]\rangle \models S\mathtt{_*\&}R
\end{array}$$

where † holds because $k - i \leq maxlen(S) \leq minlen(R) \leq j - i$ implies that $k \leq j$. Symmetrically for lookbehind. Associativity of concatenation is used to enable the rules in Figure 5 more frequently, namely that $(\mathtt{(?=}S\mathtt{)}R_1)R_2 \equiv \mathtt{(?=}S\mathtt{)}(R_1R_2)$ and $R_1(R_2\mathtt{(?<=}S\mathtt{)}) \equiv (R_1R_2)\mathtt{(?<=}S\mathtt{)}$. In particular, the rules apply to typical uses of *anchors*. For example, if R is not nullable then $\$R$ (i.e. `(?=\n|\z)`R) is rewritten to `(\n|\z)_*&`R since then $maxlen(\mathtt{\backslash n|\backslash z}) = 1 \leq minlen(R)$.

The lookarounds in **RE#** are not as expressive as the (also input-linear) implementation in Javascript V8 [Barrière and Pit-Claudel 2024], which supports $\mathbf{RE}_{\leq}$ where lookarounds can be in any position in the regex, including nesting and capturing, but the restriction is not as limiting as it

[2]The case $\mathcal{M}(R) = \emptyset$ is irrelevant in this context.

may seem at first. For many use cases, in addition to rules in Figure 5, other rewrites can be applied by using *intersection* and *complement*. For example, a pattern from the Suricata IDS [OISF 2024]:

```
\x2F(?!Subtype)(S|#53)(u|#75)(b|#62)(t|#74)(y|#79)(p|#70)(e|#65)
```

which uses a negative lookahead `(?!Subtype)` in the middle, can be rewritten in **RE#** as:

```
\x2F(~(Subtype)&(S|#53)(u|#75)(b|#62)(t|#74)(y|#79)(p|#70)(e|#65))
```

An example of a regex in $\mathbf{RE}_{\leq}$ that cannot be expressed in **RE#** is `b(?<=a.*)`, where the derivative of the lookbehind `(?<=a.*)` is not well-defined in **RE#**, as the lookbehind is unbounded and needs, e.g., the algorithm in [Barrière and Pit-Claudel 2024] to be evaluated.

Many unsupported patterns are converted via built-in rewrites (Section 5.3), e.g. the unsupported pattern `\bthe\b|\band\b` is rewritten as `\b(the|and)\b`, which is supported in **RE#**. But the pattern `(\bthe\b|and)` is not supported, as **RE#** is not closed under union, which is due to an optimization in the implementation that discards the lookbehind upon finding a valid match. In the full matching algorithm (*AllEnds*), the lookbehind is discarded after the match beginning is found, and the lookahead is subsequently used to limit the match length.

In the curated *rebar* benchmark set, 4 out of the 27 benchmarks included lookarounds in terms of anchors. All of those cases could automatically be translated into **RE#**.

4.7 Derivatives in ERE

Here we first define *derivatives* of regexes in **ERE**. This definition is also more-or-less standard. Let $x \in \mathbf{Loc}^+$ be a *nonfinal* location, in which case we know that $hd(x) \in \Sigma$. For example, if $s =$ "ab" then the nonfinal locations in s are $s[0]$ and $s[1]$. Let $\psi \in \Psi$, let $\diamond \in \{\&, |\}$, and let $\text{⚓} \in \{\backslash\mathtt{A}, \backslash\mathtt{z}\}$.

$$
\begin{array}{rcl}
\boldsymbol{\delta}_x(\text{⚓}) & \overset{\text{DEF}}{=} & \bot \\
\boldsymbol{\delta}_x(\varepsilon) & \overset{\text{DEF}}{=} & \bot \\
\boldsymbol{\delta}_x(R \diamond S) & \overset{\text{DEF}}{=} & \boldsymbol{\delta}_x(R) \diamond \boldsymbol{\delta}_x(S) \\
\boldsymbol{\delta}_x(\mathtt{\sim} R) & \overset{\text{DEF}}{=} & \mathtt{\sim}\boldsymbol{\delta}_x(R) \\
\boldsymbol{\delta}_x(R\mathtt{*}) & \overset{\text{DEF}}{=} & \boldsymbol{\delta}_x(R)\cdot R\mathtt{*}
\end{array}
\qquad
\begin{array}{rcl}
\boldsymbol{\delta}_x(R\{m\}) & \overset{\text{DEF}}{=} & \boldsymbol{\delta}_x(R)\cdot R\{m-1\} \\
\boldsymbol{\delta}_x(\psi) & \overset{\text{DEF}}{=} & \begin{cases} \varepsilon, & \textbf{if } hd(x) \in [\![\psi]\!]; \\ \bot, & \textbf{otherwise}. \end{cases} \\
\boldsymbol{\delta}_x(R\cdot S) & \overset{\text{DEF}}{=} & \begin{cases} \boldsymbol{\delta}_x(R)\cdot S \,|\, \boldsymbol{\delta}_x(S), & \textbf{if } Null_x(R) = \textbf{true}; \\ \boldsymbol{\delta}_x(R)\cdot S, & \textbf{otherwise}. \end{cases}
\end{array}
$$

There is one aspect of this definition that deserves attention as it differs from derivatives of bounded loops in [Moseley et al. 2023] where $\boldsymbol{\delta}_x(R{\cdot}R)$ is not always equivalent to $\boldsymbol{\delta}_x(R){\cdot}R$ when R is not always nullable. One culprit is the *word border* anchor \b that is (currently) not allowed in **ERE** but is in **RE#** defined via lookarounds. In general, $\boldsymbol{\delta}_x(R{\cdot}R)$ and $\boldsymbol{\delta}_x(R){\cdot}R$ are always equivalent in **RE#**.

Derivation Relation. The *derivation relation* $x \xrightarrow{R} y$ between locations $x, y \in \mathbf{Loc}$ and regexes $R \in \mathbf{ERE}$ is used to reason about consecutive derivative steps. The derivation relation combines steps so that, e.g., $x \xrightarrow{\varepsilon} x$ and $x \xrightarrow{R} x{+}2$ means that $\boldsymbol{\delta}_{x+1}(\boldsymbol{\delta}_x(R))$ is nullable in location $x{+}2$.

$$x \xrightarrow{R} y \quad \overset{\text{DEF}}{=} \quad Null_x(R) \textbf{ and } x = y \quad \textbf{or} \quad Nonfinal(x) \textbf{ and } x{+}1 \xrightarrow{\boldsymbol{\delta}_x(R)} y$$

4.8 Adding Lookarounds

Here we consider $\mathbf{LNF}(\mathbf{RE\#})$. Let $R = \mathtt{(?<=}B\mathtt{)}\cdot E\cdot\mathtt{(?=}A\mathtt{)}$ where $A, B, E \in \mathbf{ERE}$. We are first interested in finding the *latest* end location of a match of R, we replace `(?<=`B`)` with `_*`$\cdot B$. Say $D =$ `_*`$\cdot B\cdot E$.

The general derivative rule for concatenation in Section 4.7 remains unchanged for concatenation in $D\cdot$`(?=`A`)` where it uses the derivative rule for lookaheads as defined below. The basic insight for lookaheads is the fact that if `(?=`A`)` was reached there was a nullable location after matching the regex D before it. In order to recall the *offsets* to those locations, lookaheads are *annotated* as $\mathtt{(?=}A\mathtt{)}_I$ where I is a set of offsets and $\mathtt{(?=}A\mathtt{)} = \mathtt{(?=}A\mathtt{)}_{\{0\}}$ where 0 is the immediate offset.

The derivative rule for lookahead is as follows, where A is treated as ε when nullable, and recall that $\boldsymbol{\delta}_x(\varepsilon) = \bot$. We also let $(?{=}\bot)_I \stackrel{\text{DEF}}{=} \bot$ and $(?{=}A)_I \stackrel{\text{DEF}}{=} \varepsilon_I$ when A is nullable, where ε_I is ε annotated with I. Let $I+1 \stackrel{\text{DEF}}{=} \{i+1 \mid i \in I\}$.

$$\boldsymbol{\delta}_x((?{=}A)_I) \stackrel{\text{DEF}}{=} \begin{cases} \bot, & \textbf{if } \mathit{Null}_x(A); \\ (?{=}\boldsymbol{\delta}_x(A))_{I+1}, & \textbf{otherwise}. \end{cases}$$

Then $(?{=}A)_I \,|\, (?{=}A)_J$ is always rewritten to $(?{=}A)_{I \cup J}$. So $\varepsilon_I \,|\, \varepsilon_J = \varepsilon_{I \cup J}$. Also $\varepsilon_I \cdot \varepsilon_J = \varepsilon_{I \cup J}$.

Example 4.4. Consider the regex `\d+(?=:-)` that looks for a price in a text and let $s =$ `"50:- "`. Then $\boldsymbol{\delta}_{s[0]}$(`\d+(?=:-)`) = `\d*(?=:-)` and $\boldsymbol{\delta}_{s[1]}$(`\d*(?=:-)`) = `\d*(?=:-)` since the lookahead did not kick in yet. Then we get $\boldsymbol{\delta}_{s[2]}$(`\d*(?=:-)`) = $(?{=}\boldsymbol{\delta}_{s[2]}(:-))_{\{1\}} = (?{=}-)_{\{1\}}$ and finally that $\boldsymbol{\delta}_{s[3]}((?{=}-)_{\{1\}}) = (?{=}\varepsilon)_{\{2\}} = \varepsilon_{\{2\}}$ in location $s[4]$, so the match end is $s[4-2]$. ⊠

Implementation of Lookahead Annotations. The set I above is represented by a pair $\langle k, X\rangle$ containing a *relative offset* k and an index set X so that I denotes $\{k+i \mid i \in X\}$ and $I+1 \stackrel{\text{DEF}}{=} \langle k+1, X\rangle$; a specialized union $I \cup J$ is also implemented that adjusts the result to the lowest relative offset.

For purposes of DFA state caching, the sets I are only ever compared with pointer equality and there is a builder to keep track of unique set instances. This makes several orders of magnitude difference in the memory footprint and construction time. In practice, the sets I are usually sparse which allows the lookahead context to be hundreds or even thousands of characters long without contributing significantly to state space.

The complete set I is needed in the generalized algorithm for finding *all* matches. In the case when only the *first* match is searched, I is only ever needed to maintain the *minimal* offset in it and in this case becomes just that offset. Then, e.g., $(?{=}A)_I \,|\, (?{=}A)_J$ rewrites to $(?{=}A)_{\min(I,J)}$.

4.9 Latest Match End

We consider again regexes in **RE#** in the normal form described earlier and focus on the simplified and transformed case $R =$ `_*`$\cdot B \cdot E$`(?=`A`)` where $A, B, E \in$ **ERE**. We search for j such that

$$s[0] \xrightarrow{\_*\cdot B\cdot E} s[j] \xrightarrow{A\cdot \_*} s[|s|]$$

and want to find the *maximal* j if it exists. To this end we use the function $\mathit{MaxEnd}(s[i], R, m)$ below where $s[i]$ is the *current location* and m is the *maximal match end so far*. We let $\varepsilon_I \in R$ denote the epsilon with the annotations that exists (implicitly) in R, e.g., $\varepsilon_{\{0,5\}} \in$ `a*` $\,|\, \varepsilon_{\{5\}}$ because `a*` contains ε implicitly. Initially $m = -1$ and the search starts from the initial location. We first consider any *nonfinal* location $s[i]$, i.e., $i < |s|$.

$$\mathit{MaxEnd}(s[i], R, m) \stackrel{\text{DEF}}{=} \begin{cases} m, & \textbf{if } R = \bot; \\ \mathit{MaxEnd}(s[i+1], \boldsymbol{\delta}_{s[i]}(R), \max(m, i - \min(I))), & \textbf{else if } \varepsilon_I \in R; \\ \mathit{MaxEnd}(s[i+1], \boldsymbol{\delta}_{s[i]}(R), m), & \textbf{otherwise}. \end{cases}$$

The latest match end so far becomes $\max(m, i-k)$ where $k = \min(I)$ is the minimal offset from the current index i to where a valid match of E ended when $s[i-k] \xrightarrow{A} s[i]$. In particular if $m = -1$ then $i-k$ is the *first* match end that was found. Later search may reveal other match ends (including both earlier and later ones) but only the latest one is remembered here.

We now consider the case of the final location in s. In the following let $R^{\backslash z \mapsto \varepsilon}$ stand for R where `\z` is replaced by ε. In particular, any lookahead $(?{=}A)_I$ such that $A^{\backslash z \mapsto \varepsilon}$ is nullable is now automatically rewritten to $(?{=}\varepsilon)_I \stackrel{\text{DEF}}{=} \varepsilon_I$.

$$\mathit{MaxEnd}(s[|s|], R, m) \stackrel{\text{DEF}}{=} \begin{cases} \max(m, |s| - \min(I)), & \textbf{if } \varepsilon_I \in R^{\backslash z \mapsto \varepsilon}; \\ m, & \textbf{otherwise}. \end{cases}$$

For example if $R =$ `\z` $\,|\,$ $(?{=}$`a*\z`$)_{\{5\}}$ then $R^{\backslash z \mapsto \varepsilon} = \varepsilon \,|\, \varepsilon_{\{5\}} = \varepsilon_{\{0,5\}}$ and thus $\max(m, |s| - 0) = |s|$.

The following lemma is key in establishing the formal relationship between the derivation relation for **RE#** and the formal match semantics. It makes fundamental use of the theory of derivatives of $\mathbf{ERE}_{\leq}$ that has recently been fully formalized and proved correct [Zhuchko et al. 2024] using the *Lean* proof assistant. The general derivative theory developed for $\mathbf{ERE}_{\leq}$ is highly *nonlinear* for use in practice but subsumes **RE#**, which enables us to apply the main correctness result of $\mathbf{ERE}_{\leq}$ relating the general theory of derivatives in $\mathbf{ERE}_{\leq}$ with the formal match semantics.

Lemma 1. *Let* $s[i_0] \in \mathbf{Loc}$ *and* $R \in \mathbf{LNF}(\mathbf{RE\#})$*, and let* $MaxEnd(s[i_0], R, -1) = j$*. Then*

(1) $j = -1 \Leftrightarrow \nexists \imath \geq i_0, \jmath : \langle s[\imath], s[\jmath]\rangle \models R$.
(2) *If* $j \geq 0$ *then* j *is the maximal* $\jmath$ *such that* $\exists \imath \geq i_0 : \langle s[\imath], s[\jmath]\rangle \models R$.

Proof Outline. Consider $i_0 = 0$ and let $R = (?{<=}B)E(?{=}A)$ where $A, B, E \in \mathbf{ERE}$. First observe that when $\boldsymbol{\delta}_{s[j]}((?{=}A))$ is invoked it is when $s[0] \xrightarrow{\_{*}\cdot B\cdot E} s[\jmath]$. This follows because $(?{<=}B)$ is replaced by $\_{*}\cdot B$ and *MaxEnd* just iterates derivatives from one location to the next. From this point forward the offsets after taking each derivative are increased. For the current location $s[i]$ and $(?{=}D)_I$ where D has been derived from A we know that the latest *candidate* match exists at index $i - \min(I)$. *MaxEnd* then keeps track of the latest *valid* match end index when D is nullable. So $MaxEnd(s[0], R, -1)$ returns the latest such index or -1 if there is none. The final location is handled separately which is the only location where \z is equivalent to ε.

We now use the fact that **RE#** is a fragment of $\mathbf{ERE}_{\leq}$ and that the derivative rules for **ERE** are the same as in $\mathbf{ERE}_{\leq}$. We use [Zhuchko et al. 2024, Theorem 2] (say †)

$$\begin{aligned}\exists \imath : \langle s[\imath], s[\jmath]\rangle \models R &\Leftrightarrow \exists \imath : s[\imath] \models (?{<=}B) \textbf{ and } \langle s[\imath], s[\jmath]\rangle \models E \textbf{ and } s[\jmath] \models (?{=}A)\\ &\overset{\dagger}{\Leftrightarrow} \exists \imath : s[0] \xrightarrow{\_{*}\cdot B} s[\imath] \xrightarrow{E} s[\jmath] \xrightarrow{A\cdot\_{*}} s[|s|] \overset{\dagger}{\Leftrightarrow} s[0] \xrightarrow{\_{*}\cdot B\cdot E} s[\jmath] \xrightarrow{A\cdot\_{*}} s[|s|]\end{aligned}$$

This completes the proof because *MaxEnd* returns the maximal such $\jmath$ iff it exists or else -1. □

4.10 Leftmost-Longest Match Algorithm

We now describe the main match algorithm in **RE#**. It uses reversal and the *MaxEnd* algorithm above in two directions to compute the match such that the so-called POSIX semantics holds. What is unique about this algorithm is that *in the general case* described below it traverses the input string in *reverse* in the first phase in order to find the earliest or leftmost start index. We describe the algorithm for $R = (?{<=}B)\cdot E\cdot(?{=}A)$ as follows.

$$\begin{aligned}LLMatch(s, R) \overset{\text{DEF}}{=} &\textbf{ let } k = MaxEnd(s^{\mathbf{r}}[0], R^{\mathbf{r}}, -1) \textbf{ in}\\ &\quad \textbf{if } k = -1 \textbf{ then return } \bot \textbf{ else}\\ &\quad\quad \textbf{let } i = |s| - k;\ j = MaxEnd(s[i], E\cdot(?{=}A), -1) \textbf{ in}\\ &\quad\quad\quad \textbf{return } \langle s[i], s[j]\rangle\end{aligned}$$

Theorem 3 (LLMatch). *$LLMatch(s, R)$ returns $\bot$ if there exists no match of R in s else returns the match $\langle s[i], s[j]\rangle$ of R where i is minimal and j is maximal for i.*

Proof. Let $R = (?{<=}B)\cdot E\cdot(?{=}A)$. Then $R^{\mathbf{r}} = (?{<=}A^{\mathbf{r}})\cdot E^{\mathbf{r}}\cdot(?{=}B^{\mathbf{r}})$. Let $k = MaxEnd(s^{\mathbf{r}}[0], R^{\mathbf{r}}, -1)$. If $k = -1$ then, by Lemma 1(1), $\nexists \theta \in \mathbf{Span}(s^{\mathbf{r}}) : \theta \models R^{\mathbf{r}}$ and, by Theorem 2, $\nexists \theta \in \mathbf{Span}(s) : \theta \models R$.

Assume $k \geq 0$. Then, by Lemma 1(2), k is the *maximal* index such that $\exists x : \langle x, s^{\mathbf{r}}[k]\rangle \models R^{\mathbf{r}}$. So, by Theorem 2, $i = |s| - k$ is the *minimal* index such that $\exists x : \langle s[i], x\rangle \models R$. (Recall that $s^{\mathbf{r}}[k]^{\mathbf{r}} = s[|s| - k]$.) Thus, i is the minimal index such that

$$\exists x : s[i] \models (?{<=}B) \textbf{ and } \langle s[i], x\rangle \models E \textbf{ and } x \models (?{=}A)$$

In particular, it follows that $\exists x : \langle s[i], x\rangle \models E\cdot(?{=}A)$. Now, by using Lemma 1(2) again, it follows that that $j = MaxEnd(s[i], E\cdot(?{=}A), -1)$ is the maximal index such that $\langle s[i], s[j]\rangle \models R$. □

In the implementation of *LLMatch* for a *single* POSIX match search as described above, the sets I in $(?{=}A)_I$ are implemented by only keeping their *minimal* elements – then $I \cup J = \min(I, J)$. This does not affect any of the statements above, because only the minimal element is ever used above but the more general formulation is needed in *AllEnds* below.

Theorem 4 (InputLinearity). *The complexity of LLMatch(s, R) is linear in $|s|$.*

Proof. The main search algorithm runs twice over the input s. The regexes reached by reading the symbols from s are internalized and cached as states in a DFA with $q_0 = R$ as the initial state and $\boldsymbol{\delta}_a(q)$ as the transition function of the DFA, where the operators | and & are treated as associative, commutative and idempotent operators, which results in a finite state space whose size is independent of $|s|$. The offset annotation I maintained in $(?{=}A)_I$ is incremented linearly up to the point when A becomes nullable and where $(?{=}A)_I \,|\, (?{=}A)_J = (?{=}A)_{\min(I,J)}$. □

All nonbacktracking engines are in principle input linear for a single match search and internally maintain some form of DFA. When the number of DFA states grows too large they fall back in an NFA mode. Such a fallback mechanism is currently not supported in **RE#** but can be implemented by working with a generalized form of Antimirov derivatives [Antimirov 1996].

The **RE#** engine capitalizes on the fact that the symbolic derivative based automata construction is small and independent of the alphabet size, but the state space can still grow super-exponentially with respect to the size of the regex in the worst case. For the most critical use cases, we provide the option to precompile the regex into a *complete DFA* up front. This allows for extremely fast matching at the cost of a potentially large memory footprint, which can be known ahead of time.

The worst-case scenario for **RE#** involves the lazy construction of one extra unique transition and node per *compressed* character of the input. For example, the regex `\d+` uses two compressed characters, one for all digits and the other one for all non-digits. Taking the size of the regex into account, this involves $O(m * n)$ operations, where m is the number of unique syntax nodes created per-transition and n is the length of the input; m is independent of n but can be super-exponential in the size of R (that includes the size of the compressed alphabet that is typically very small related to the size of the Unicode alphabet Σ). As a last resort, the engine has a configurable memory limit that can be set to trigger an exception if the memory usage exceeds a certain threshold.

Finding All Nonoverlapping Leftmost-Longest Matches. The key algorithm *MaxEnd* above is generalized in **RE#** into an algorithm *AllEnds* that produces *all* match ends as follows. We then discuss more informally how *AllEnds* is used in the general match algorithm in **RE#** to locate all nonoverlapping POSIX matches. Similar to *MaxEnd* the algorithm takes a regex $R = (?{<=}B){\cdot}E{\cdot}(?{=}A) \in \mathbf{LNF}(\mathbf{RE\#})$ and a start location $s[i]$ but in this case a *set* M of *match end indices found so far*. We consider only the case of $i < |s|$ with the case $i = |s|$ being analogous to above.

$$AllEnds(s[i], R, M) \stackrel{\text{DEF}}{=} \begin{cases} M, & \textbf{if } R = \bot; \\ AllEnds(s[i+1], \boldsymbol{\delta}_{s[i]}(R), M \cup i - J), & \textbf{else if } \varepsilon_J \in R; \\ AllEnds(s[i+1], \boldsymbol{\delta}_{s[i]}(R), M), & \textbf{otherwise}. \end{cases}$$

where $i - J \stackrel{\text{DEF}}{=} \{i - \jmath \mid \jmath \in J\}$. Let $MaxEnd(x, R) \stackrel{\text{DEF}}{=} MaxEnd(x, R, -1)$. Thus $MaxEnd(s[0], R) = \max(AllEnds(s[0], R, \emptyset))$, provided that $\max(\emptyset) \stackrel{\text{DEF}}{=} -1$ here, where *all* match ends, including the maximal one, are collected in M.

If we now first compute $I = |s| - AllEnds(s^{\mathrm{r}}[0], R^{\mathrm{r}}, \emptyset)$ then it holds, similarly to case above, that I contains *all the start indices* i such that $s[i] \models (?{<=}B)$ and $\exists\, j : \langle s[i], s[j]\rangle \models E{\cdot}(?{=}A)$.

Starting with $i = \min(I)$ we compute $j = MaxEnd(s[i], E{\cdot}(?{=}A))$. This gives us the first POSIX match $\langle s[i], s[j]\rangle$. We now repeat the same search from $I := \{\imath \in I \mid i < \imath, j \leq \imath\}$ to ignore overlapping matches. Observe that $i < \imath$ is necessary to make progress when $j = i$.

This concludes our high-level overview of the main matching algorithm $LLMatches(s, R)$ in **RE#**. The implementation of $LLMatches(s, R)$ is not linear, but may in the worst case be quadratic in $|s|$. However, our extensive evaluation does consistently indicate linear behavior, even for the *quadratic* benchmark (see Section 6.1.4).

Example 4.5. Let $R =$ `b+(?=c)` and $s =$ `"aaaaabcababbc"`. Then initially

$$I = |s| - AllEnds(\texttt{"cbbabacbaaaaa"}[0], \texttt{(?<=c)b+}, \emptyset) = 13 - \{2, 3, 8\} = \{5, 10, 11\}$$

The first match starts from 5 and ends at $MaxEnd(s[5], \texttt{b+(?=c)}) = 6$. This leaves $I := \{10, 11\}$. The next match starts from 10 and ends at $MaxEnd(s[10], \texttt{b+(?=c)}) = 12$. This leaves $I := \emptyset$ and concludes the search. Thus $LLMatches(s, R) = \{\langle s[5], s[6]\rangle, \langle s[10], s[12]\rangle\}$. ⊠

5 Implementation

Here we give a brief overview of the implementation of the engine along with some key optimizations and performance considerations. At the high level, derivatives are computed lazily and cached in a DFA with regexes internalized as states and use the transition function $\boldsymbol{\delta}_a(q)$ for states q.

The core parser was taken directly from the .NET runtime, but was modified to read the symbols `&` and `~` as intersection and complement respectively. The parser was also extended to interpret the symbol `_` as the set of all characters, since it is very commonly used in our regexes. Fortunately the escaped variants `\&`, `\~` and `\_` were not assigned to any regex construct, and existing regex patterns can be used by escaping these characters.

The parser also supports Unicode symbols for operators ($\not\Leftrightarrow, \Leftrightarrow, \Rightarrow$), as explained in Table 3, for more advanced Boolean operations. Moreover, rather than implementing those operators through the core Boolean operators, one can add specialized rules. Derivative rules for the extended operations are in fact *identical* as for $\diamond$ in Section 4.7. Nullability, for example for XOR, can be defined by $Null_x(L \not\Leftrightarrow R) \stackrel{\text{DEF}}{=} (Null_x(L) \neq Null_x(R))$, and analogously for the other operators.

The matching implementation of nearly all industrial regex engines consists of two separate components: a prefilter and a matcher. The prefilter is essential to be competitive with other engines, and is used to quickly eliminate non-matching strings. Our derivative-based approach is used in both components, which in some cases provides a significant advantage over other engines.

5.1 Prefilter

The prefilter is an input-parallel operation on the side that is applied aggressively. For simple regexes, such as `[abc]`, the prefilter can locate the entire match in parallel, where the only real-world limitation is availability of space per parallel operation. For more complex regexes, the prefilter is used to locate the *prefix* of the input string that is guaranteed to match the regex, before the core engine kicks in. In **RE#** the prefilter is using vectorized bitwise operations, which are very efficient on modern CPUs. The speedup of processing 64 bytes at a time, e.g., using AVX512 instructions, is significant and immediately visible in the overall performance.

It is important to note that, unlike in many other engines, the prefilter optimizations in our engine are not limited to simple regexes, but are applied to all regexes in a derivative-based manner, including lookarounds, which makes the extensions very competitive in real-world scenarios. The impact of this systematic approach to prefilters is shown in Sections 6.2.6 and 6.2.4.

5.1.1 Breadth-First Derivative Calculation. Derivatives are used to examine the optimizations available for the regex pattern. One key optimization is to explore *all* derivatives symbolically, or in a *breadth-first* manner, and bitwise-merge their conditions until reaching the first successful match. This provides a way to optimize the matching process with more specialized algorithms.

5.1.2 Prefix Search. Often, the entire regex pattern is a single string literal or a set of words. In such cases, we optimize the matching process by using a dedicated string matching algorithm. We first check if the regex pattern is a string literal or a small set (up to 20) of string literals, and if so, we use the *Teddy* [Qiu et al. 2021] algorithm, recently supported in .NET9 to locate matches.

5.2 Combined Techniques and Inner Loop Vectorization

A key difference from other regex engines is that we do not just use specialized search algorithms for locating the prefix, but the algorithms are deeply integrated with the core engine. We combine the search algorithms with automaton transitions that have been cached from derivatives, which allows us to use specialized algorithms for the prefix in the input text, and transition through multiple steps in the automaton right away.

One such example is the regex pattern `abcd.*efg`, which can be optimized to match the string literal `abcd` in the input text and then immediately transition to the automaton state representing `.*efg` for the rest of the match.

We also use the derivatives to perform intermediate prefix computations and appropriate skipping in inner loop of the match. The breadth-first calculation of derivatives is used to compute the prefix of the remaining regex, which is then used to skip over large parts of the input string in a single step, and only perform the more expensive automaton transitions on the remaining positions. This means that the rest of the aforementioned pattern `.*efg` makes use of input-parallel algorithms as well.

All of the DFA states have pre-computed optimizations during construction, which are used whenever possible to skip over parts of the input string. A benchmark scenario illustrating the power of inner-loop optimizations is shown in Section 6.2.4, where **RE#** is shown to be significantly faster on long matching strings than other engines.

However, it is important to note that vectorization is not always beneficial. For example, even though the AVX512 instruction set can process 64 bytes at a time, the overhead of setting up the vectorized operations can be significant for large common character sets, such as `[a-zA-Z]`. In such cases, the engine falls back to automaton transitions. Even the Teddy algorithm is not always beneficial, as it has a certain upper limit (roughly 20) on the number of strings it can process efficiently, otherwise the engine falls back to automaton transitions for large alternations as well.

5.3 Rewrite Rules and Subsumption

Our system implements a number of regex rewrite rules, which are essential for the efficiency of the implementation. Figure 6 illustrates the basic rewrite rules that are always applied when regular expressions are constructed. Intersection and union are implemented as commutative, associative and idempotent operators, so changing the order of their arguments does not change the result.

Each $R \in \mathbf{ERE}$ comes with a predicate $\varphi_R \in \Psi$ that approximates its relevant characters. The definition is: $\varphi_\psi \stackrel{\text{DEF}}{=} \psi$, $\varphi_{\sim R} \stackrel{\text{DEF}}{=} \_$, $\varphi_{L|R} = \varphi_{L \cdot R} \stackrel{\text{DEF}}{=} \varphi_L \vee \varphi_R$, $\varphi_{L\&R} \stackrel{\text{DEF}}{=} \varphi_L \wedge \varphi_R$, and $\varphi_{R\{m\}} = \varphi_{R*} \stackrel{\text{DEF}}{=} \varphi_R$. Also $\varphi_\varepsilon = \varphi_{\backslash\mathtt{A}} = \varphi_{\backslash\mathtt{z}} \stackrel{\text{DEF}}{=} \bot$. All operations of $\mathcal{A}$ are $O(1)$ operations and if $\varphi \equiv \psi$ then $\varphi = \psi$. For example, the test $\mathtt{\backslash n} \notin [\![\varphi_R]\!]$ is $\varphi_{\mathtt{\backslash n}} \wedge \varphi_R = \bot$ and the test $[\![\phi]\!] \subseteq [\![\psi]\!]$ is $\phi \vee \psi = \psi$ in Figure 6.

There are many further derived rules that can be beneficial in reducing the state space. Unions and intersections are both implemented by sets. If a union contains a regex S, such as a predicate ψ, that is trivially subsumed by another regex R, such as $\psi*$, then S is removed from the union. This is an instance of the loop rule in Figure 6 that rewrites $\psi*|\psi\{1\}$ to $\psi*$ (where $\psi* = \psi\{0,\infty\}$).

A further simplification rule (using φ_R in Figure 6) for unions is that if a union contains a regex $\psi*$ and all the other alternatives only refer to elements from $[\![\psi]\!]$ then the union reduces to $\psi*$.

$$\frac{\sim(\_*)}{\bot} \quad \frac{\sim\bot}{\_*} \quad \frac{\sim\sim R}{R} \quad \frac{\sim\varepsilon}{\_+} \quad \frac{\sim(\_+)}{\varepsilon} \quad \frac{\bot\cdot R}{\bot} \quad \frac{R\cdot\bot}{\bot} \quad \frac{\varepsilon\cdot R}{R} \quad \frac{R\cdot\varepsilon}{R} \quad \frac{\bot*}{\varepsilon} \quad \frac{\_*\,|\,R}{\_*} \quad \frac{\_*\&R}{R} \quad \frac{(?{=}\bot)_I}{\bot}$$

$$\textsc{loop}\,\frac{R\{l,m\}\,|\,R\{k,n\}}{R\{l,\max(m,n)\}}\,(l \le k \le m, m \le \infty) \qquad \frac{.*\,|\,R}{.*}\,\backslash\mathtt{n} \notin [\![\varphi_R]\!] \qquad \frac{.*\&R}{R}\,\backslash\mathtt{n} \notin [\![\varphi_R]\!]$$

$$\textsc{sub}1\,\frac{(R1\&R2)\,|\,R1}{R1} \qquad \textsc{dedup}\,\frac{R1 \diamond R2 \diamond R1}{R1 \diamond R2} \qquad \textsc{sub}2\,\frac{(R1\&(R2|R3))\,|\,(R1\&R2)}{(R1\&(R2|R3))}$$

$$\frac{R1R2|R1R3}{R1(R2|R3)} \qquad \frac{R1R3|R2R3}{(R1|R2)R3} \qquad \frac{\phi\{0,m\}\psi*}{\psi*}\,[\![\phi]\!] \subseteq [\![\psi]\!] \qquad \frac{(?{=}R)_I}{\varepsilon_I}\,Null(R)$$

Fig. 6. Basic rewrite rules where $\diamond \in \{|, \&\}$ and $\phi, \psi \in \Psi$.

This rule rewrites any union such as (`.*ab.*|.*`) to just `.*` (recall that `. ≡ [^\n]`), which significantly reduces the number of alternatives in unions.

5.4 Overhead Elimination

To make the engine competitive in scenarios with frequent matches, it is important to keep the engine as lightweight as possible and to avoid unnecessary operations. Many optimizations are cached into bit flags, which are used to quickly determine if a certain operation is necessary. For example, there is a bit flag for checking if a regex is always nullable, which is immediately marked as true if the regex accepts the empty string ε. There is also a specialized flag for anchor nullability, which is used to quickly determine if an anchor was valid in the previous position.

There are also shortcuts to quickly return $\bot$ if a dead state is reached, and for checking if an automaton state has more specialized algorithms available. We are also extensively using pointer comparisons for equality checks, e.g. for checking if two regexes are the same, or if a regex is a subset of another regex.

For match end lookups, as we know the position of the match start, we often skip a number of transitions in the automaton, e.g. if the regex is `abcd.*efg`, we can skip the transitions for `abcd` entirely and start the match 4 characters ahead with the transitions for `.*efg`, which is a significant optimization for many regexes.

The engine also supports using ASCII bytes as input, which effectively doubles the speed of vectorized operations, as the engine can process double the characters per parallel step. We do not use this optimization in any of the benchmark comparisons, as the UTF-16 input is more mature in .NET and has more algorithms readily available, but it is a significant optimization for many real-world scenarios, especially when processing large amounts of data. We are also planning to support UTF-8 input directly in the future.

When the engine detects no opportunities for more-specialized algorithms and falls back to automaton transitions, it uses a highly optimized loop, which does not store any intermediate results, and only uses the automaton transitions to determine the match end. Additionally, the engine compiles very small regexes directly into full DFAs, which eliminates a conditional branch dead center in the hot-path, which would otherwise be used to lazily create new states.

5.5 Pending Nullable Position Representation in Lookahead Annotations

The set of pending match positions is a key component of the engine, and is used to keep track of context throughout the matches. As the set can grow very large, it is important to keep the representation and operations on the set lightweight in terms of memory and CPU usage.

The set is represented as a sorted list of ranges, which is minimized during construction. One frequent operation is to increment all the positions in the set, which is done by simply incrementing

the start and end of each range. Contiguous ranges are also merged during this operation, which often results in a very small memory overhead. For example, a pending match set of 10000 sequential positions can be represented as a single range, which can take up as little as 8 bytes of memory for int32 positions. This allows the context length to be very large in practice, and the engine can handle complex regexes with many lookarounds.

5.6 Validating Correctness of RE# Implementation Using Formalized Lean Semantics

There are many low-level optimizations and rewrite rules in the **RE#** engine. For example, for obvious reasons, no input string s is ever actually reversed but $s^{\mathbf{r}}$ is an abstraction that hides underlying index calculations. We use the Lean formalization of $\mathbf{ERE}_{\leq}$ and its POSIX matching semantics [Zhuchko et al. 2024] that is *executable* because the membership test $a \in [\![\psi]\!]$ in $\mathcal{A}$ is executable. The *span* universe **Span** is in Lean defined as $\Sigma^* \times \Sigma^* \times \Sigma^*$ with $\langle u, v, w\rangle$ representing $\langle s[i], s[j]\rangle$ where $s = u^{\mathbf{r}}vw$, $i = |u|$ and $j = |uv|$. Although the semantics in Lean is highly *nonlinear* it does have the same semantics for **RE#** (as **RE#** $\subsetneq \mathbf{ERE}_{\leq}$) and is the only test oracle available.

The **RE#** engine was extensively tested with thousands of regexes, and the results where compared with the expected results according to the Lean specification. This helped to find numerous bugs throughout the development of the engine, such as the handling of the edges of the input string and detecting off-by-one errors in reversal. One such bug we found during implementation was in the regex `^\n+`, which should have matched the full input string `"\n\n"`, but instead matched only the second `\n`, because the engine did not handle the edge of the input string correctly.

6 Evaluation

We have evaluated the performance of our engine on a number of regex benchmarks, and compared it to other regex engines available in the *BurntSushi/rebar* benchmarking tool [Gallant 2024]. The benchmarks are split into two categories: the baseline comparison (Section 6.1) consists of the curated regex benchmark suite from the *BurntSushi/rebar* tool; the extended comparison (Section 6.2) consists of a set of regexes that are designed to emphasize the strengths of our engine.

The benchmarks report the throughput of the engine in terms of the number of bytes processed per second, and the geometric mean (μ_g) ratio of the throughput is used as the primary metric for comparison. Each of the benchmarks is reported by the ratio of throughput compared to the best performing engine in the benchmark, where **1x** is the leader in each individual benchmark. The overall μ_g ratio is displayed for each major category, where the displayed ratio means, e.g., **3x** is twice as fast as **6x**. Any number larger than **1x** implies that the engine lost some benchmarks in the category. The results are shown in Table 4. Below we analyze the results in some detail.

The measurements were performed on a machine running an Ubuntu 22.04 Docker image with an AMD Ryzen Threadripper 3960X 24-Core Processor and 128 GB of memory. As an important note, the only form of parallelism that is used in the engine is vectorization, which is also used by many other engines to achieve shown results.

6.1 Baseline Comparison

The baseline comparison is done on the popular curated regex benchmark suite and using the publicly available *BurntSushi/rebar* benchmarking tool, from which we included all benchmarks that our engine supports. There are 27 benchmarks in total. Benchmarks requiring unsupported features, e.g., capture groups, are excluded from the comparison. Since **RE#** (resharp) uses *leftmost-longest* matching semantics, the match results are carefully compared with the other engines. The benchmarks also use *earliest* matching semantics in the case of Hyperscan. In 25 out of 27 benchmarks, the match results are *identical* to *leftmost-greedy* engines.

Table 4. Benchmark μ_g slowdown. Top three outcomes in each benchmark category are indicated in bold.

(a) Baseline evaluation (Section 6.1).

Engine	μ_g *relative slowdown ratio*					
	Sec 6.1	Sec 6.1.1	Sec 6.1.2	Sec 6.1.3	Sec 6.1.4	Sec 6.1.5
resharp	**1.48**	**1.98**	**1.12**	**1.43**	**1.86**	**1**
rust/regex	**2.54**	**1.49**	3.69	**1.38**	21.9	**1.2**
hyperscan	**2.76**	**1.69**	**1.76**	102	**1**	**2.26**
dotnet/comp	5.29	2.7	3.71	**3.77**	**4.52**	208
pcre2/jit	7.86	3.48	**2.3**	615	23.6	17.6
dotnet/nobt	9.14	5.88	6.61	27.7	42.7	3.58
re2	12.3	11.7	5.05	16.7	39.1	25.6
javascript/v8	19.5	6.26	4.78	1503	25.4	140
regress	50.5	23.4	5.34	3690	77.1	521
python/re	65.2	40	11.6	900	149	599
python/regex	66.1	21.7	17	4516	122	800
perl	74.7	41.1	44.9	2572	146	20.8
java/hotspot	77.4	86.5	9.13	3841	40.1	618
go/regexp	166	237	29.5	189	423	997
pcre2	285	472	42.6	8519	161	651

(b) Extended evaluation (Section 6.2).

Engine	μ_g *relative slowdown ratio*						
	Sec 6.2	Sec 6.2.1	Sec 6.2.2	Sec 6.2.3	Sec 6.2.4	Sec 6.2.5	Sec 6.2.6
resharp	**1.09**	**1**	**1**	**1.09**	**1.02**	**1**	**1.14**
hyperscan	**3.77**	**1.08**	**2.14**	**1.79**	**5.85**	-	-
dotnet/nobt	**10.7**	**2.13**	**9.19**	-	67.4	**11.6**	-
pcre2/jit	20	23.8	-	86.2	38.5	**9.21**	**15.8**
rust/regex	29.5	4.16	2747	**7.63**	20.1	82.9	-
dotnet/comp	41.3	80.9	554	694	**7.48**	25.6	**5.26**
re2	48.9	175	1440	28	16.5	-	-
python/regex	141	139	1673	1196	50.7	48.3	79.6
javascript/v8	214	86.5	449	66.9	-	-	141
python/re	233	93	2005	936	188	62.4	135
regress	360	99.6	1264	749	-	-	188
pcre2	503	2399	-	599	1076	943	255
java/hotspot	698	176	2597	769	423	56.3	718
go/regexp	957	318	2733	1135	763	-	-
perl	1143	6.32	199	1310	7195	2863	1189

The baseline benchmarks are ran "as is", without any modifications to the regexes or the input strings. Certain apples-to-apples benchmarks, when appropriate, are included in the Section 6.2 to display the performance of the engine in a more controlled environment. The baseline summary geometric mean of speed ratios is shown in Table 4a *with a separate column for each benchmark category below as indicated by the column title.*

In the figures dotnet/comp is the regex option `Compiled` in .NET and dotnet/nobt is the regex option `NonBacktracking` in .NET. The javascript/v8 engine is used in backtracking mode in all experiments. We have omitted the engine version numbers, but have used the most recent available stable versions in all cases.

6.1.1 Literal and Literal-Alternate Categories (10 benchmarks). The literal category consists of regexes that are simple string literals, and the literal-alternate category consists of regexes that are simple alternations of string literals.

These categories are orthogonal to the engine itself, as the performance is mostly determined by the string matching algorithms used in the engine, and whether an how well the engine supports the literal optimizations, e.g., those in Section 5.1.

RE# does not win in either of these categories, see Table 4a, but it is consistently close to the top performer, with the worst performance being in the literal-alternate 'sherlock-ru' benchmark, where the engine is 3x slower than the best performing engine, rust/regex.

Having more highly optimized 8-bit string literal matching algorithms would be beneficial to compete in the ASCII categories of this benchmark, but the engine is still competitive here, and not far off from the top in both categories. Rust and Hyperscan perform excellent in both of these categories with their strong string literal optimizations and geometric mean performance ratio. Hyperscan has a single outlier in the unicode literal 'sherlock-ru' benchmark, where it is 10x slower, which brings the geometric mean performance down to 1.69x.

6.1.2 Words and Bounded-Repeat Categories (8 benchmarks). The words and bounded repeat (or counters) categories consist mostly of benchmarks that are simple with many short matches, such

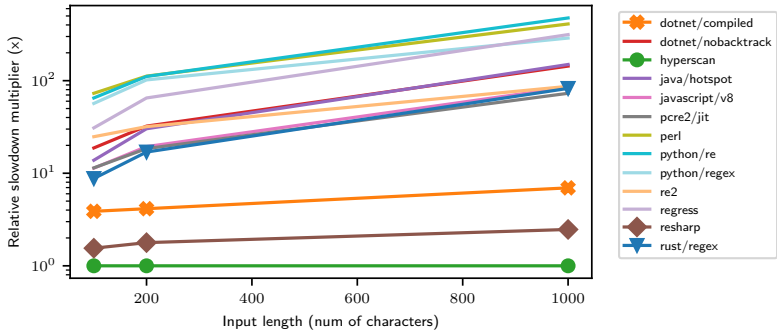


Fig. 7. *Quadratic* benchmark results for 1x=100, 2x=200, 10x=1000

as `\b\w+\b` and `[A-Za-z]{8,13}`. **RE#** performs very well in these categories, as do other automata-based engines. What sets **RE#** apart is the ability to efficiently handle *Unicode* as discussed also later in Section 6.2. On patterns such as `\b\w{12,}\b`, **RE#** is over 7x faster than the next best engine, pcre2/jit, and over 10x faster than the rest of the competition.

The reason for this gap is that `\w` denotes a very large character set, and other automata-based engines cannot handle it as efficiently, as `\w` may contribute with tens of thousands of individual transitions. **RE#** also has a very efficient implementation of the word boundary `\b` – represented via negative lookarounds in the engine and encoded directly into DFA transitions – which causes the engine to have a very fast inner matching loop. Another benchmark where **RE#** excels at is the 'context' benchmark, which uses `[A-Za-z]{10}\s+[\s\S]{0,100}Result[\s\S]{0,100}\s+[A-Za-z]{10}`, it is over 8x faster than other automata-based engines, which struggle with the `[\s\S]{0,100}` part of the pattern, as it creates many transitions in the automaton. This is where the algebraic approach to regex matching shines, as the engine can easily detect redundant transitions through the LOOP rule in Figure 6 and thereby minimize the automaton on the fly.

6.1.3 CloudFlare-ReDOS (3 benchmarks). This category is designed to showcase worst-case performance of regex engines. The regexes themselves are not very practical, but useful to distinguish between the engines that have good worst-case performance and those that do not.

The cloud-flare-redos category is a set of regexes that are designed to trigger catastrophic backtracking in backtracking regex engines. The benchmark comes in three variants: the 'original' variant is using the pattern that caused the CloudFlare outage in 2019, while the 'simplified-short' and 'simplified-long' variants are matching the regex `.*.*=.*`, which on linear complexity engines is essentially benchmarking "how fast can you find the equals sign", and on backtracking engines is a worst-case scenario, where certain backtracking engines are hundreds of thousands, even millions of times slower than the best performing engines. **RE#** is the top performer in the 'original' variant of the benchmark, which does not really show anything meaningful about the engine, apart from the fact that it does not suffer from catastrophic backtracking.

6.1.4 Quadratic (3 benchmarks). The regexes here are intended to trigger quadratic behavior in all-match scenarios. The benchmark comes in three variants: '1x', '2x', and '10x', which illustrate how the performance of the engines scales with the input length. While the reason for Hyperscan's excellent performance is *earliest* match semantics, which guarantees linearity, the reason for **RE#**'s close-to-linear performance is different. While Table 4a shows the μ_g relative slowdown ratios, Figure 7 illustrates the three variants separately.

The reason for **RE#**'s excellent performance in the quadratic category is that the engine finds a match in a fixed number of input-parallel steps, which it detects at the level of the algebraic

representation of the regex. The engine still suffers from the number of matches, which brings the performance down to $O(pn)$, where p is the number of parallel steps required to process the input, and n is the number of matches. The engine is still significantly faster than other engines in the category apart from Hyperscan, which has a linear time complexity by design.

6.1.5 Date and Dictionary (3 benchmarks). These are large regexes with many alternations. The date benchmark is described as a tokenizer for dates in various formats, which has numerous short matches of <5 characters. The dictionary benchmark consists of approximately 2500 words, which measures the speed of traversing a string with many alternations.

While **RE#** wins here, see Table 4a, both of these categories have flaws and should be taken with a grain of salt. Neither of the patterns are sorted by descending length, which means that the pattern consists of many alternations completely unreachable to PCRE engines, such as `may|mayo`, where the engine will never match `mayo` at all.

Upon further inspection, this behavior seems to originate from a near decade old semantic bug in a Python library for finding dates [Koumjian 2024], that gets millions of downloads per month, but has somehow gone unnoticed. And the dictionary benchmark consists of many alternations ordered such as `(absentmindedness|absentmindedness's)`, where the second alternation will never be matched. Furthermore, the dictionary benchmark contains only *one match*, which barely explores any of the state space of the automaton than can arise from the regex.

Since **RE#** uses a larger regex in both of these benchmarks, it also reports a slightly higher match length sum of 111832 instead of 111825 in the two *date* benchmarks, where certain matches are longer than their *leftmost-greedy* counterparts. For this reason, these benchmarks are separately analyzed in the extended benchmark in Section 6.2, where the patterns are sorted by length, and the performance of the engines is compared in a more controlled environment.

6.2 Extended Comparison

The extended comparison consists of a set of regexes to emphasize the strengths of our engine. The benchmarks have been split into several categories. The first category consists of modified versions of the *date* and *dictionary* benchmarks from the rebar benchmark suite. Hyperscan is included in very few of these benchmarks as it does not support `\b` with Unicode characters or lookarounds or patterns that exceed a certain length, but for the sake of comparison, we include it in benchmarks using multi-pattern mode and *earliest* match semantics whenever possible.

The extended summary μ_g of speed ratios is shown in Table 4b *with a separate column for each benchmark category below as indicated by the column title.* The actual μ_g of several engines is larger than shown, as many of the benchmarks are designed to push the engines to their limits, and the engine may not finish the benchmark in **1 minute** that is the *cut-off* time.

6.2.1 Date and Dictionary Amended (2 benchmarks). The benchmarks are the same as in the baseline comparison apart from two small, but significant, changes:

- ✓ alternations (unions) are sorted in descending order by length
- ✓ inputs contain not just one but over a thousand unique matches

The large amount of *unique* matches is especially important, as it prevents the lazy automata engines from creating a tiny purpose-built automaton for matching the exact same string over and over. Adding more matches to the input drops the performance of the lazy automata engines significantly, including **RE#**.

The throughput reported for **RE#** in the *dictionary* benchmark is 564.5MB/s with 1 match, and 107.3MB/s with 2663 matches. But what is notable here is that the performance of **RE#** does not fall with complexity at the same rate as the other automata engines. Where the throughput of rust/regex

falls from 535.6MB/s to 8.9MB/s, and the throughput of re2 falls from 3.6MB/s to 618KB/s. Even the throughput of Hyperscan falls from 5.4 GB/s to 104.6MB/s by increasing the number of matches, which is just slightly below the throughput of **RE#**. **RE#** still maintains this level of performance with even far more complex regexes, as show in the *monster* regex category in Section 6.2.2.

6.2.2 Monster Regexes (5 benchmarks). This category comprises of large regexes designed to stress the engines to their limits. These regexes are challenging for both backtracking as well as automata engines, where the former will suffer from redundant work and the latter will suffer from large state space complexity. This category illustrates one of the biggest strengths of **RE#**, where it dominates the competition in all of the benchmarks in this category, see Figure 2a, thanks to both its small symbolic automaton and algebraic rewrites. Hyperscan is included in the first three benchmarks as these patterns can be split into multiple individual patterns and run in multi-pattern mode, but not in the last two benchmarks, as these consist of one large pattern, which exceeds the maximum size supported by Hyperscan. All of the patterns exceed the maximum size supported by pcre2/jit as well, which is why it is not included.

The first benchmark in this category is the same dictionary regex as in the previous benchmark, but with *case insensitivity* enabled (`IgnoreCase` option or `(?i:`R`)`). Ignoring case on the dictionary regex significantly increases the state space complexity of the regex, and neither backtracking nor the automata engines can handle it. Hyperscan with multi-pattern mode does well here, albeit with an easier pattern than the others because of *earliest* match semantics. Perl seems to have some interesting optimizations for ASCII dictionaries specifically, being the only backtracking engine that can handle it. But the interesting part in this benchmark is how, very counterintuitively, the performance of **RE#** and dotnet/nobt *increases* when case is ignored.

The throughput of **RE#** with the case-insensitive dictionary regex actually increases by $\approx 40\%$ over the case-sensitive version, which is due to the size of the automaton *decreasing* when case is ignored, as the engine can merge transitions together. This is a very interesting result, as many others completely fall apart with case insensitivity enabled, with their throughput falling hundreds of times compared to the case-sensitive version.

The second and third benchmarks are similar dictionary regexes, but with unicode characters. This benchmark illustrates the performance of the engine on unicode character classes, which are difficult to handle for most engines. On the case-insensitive version of the unicode dictionary, most engines are several orders of magnitude behind **RE#**, apart from hyperscan and dotnet/nobt, which are 5.2x and 5.5x slower than **RE#**, which are still very good results, as the fourth-fastest engine, perl, is 862x slower than **RE#**.

The last two benchmarks add a context of 50 characters in the form of `.{0,50}` on either side of an already difficult case-insensitive dictionary regex, which forces the engine to explore a significant amount of possibilities, as the context can be anything. These benchmarks are a difficult scenario for even dotnet/nobt, which otherwise manages to keep up in these difficult scenarios, here even dotnet/nobt is 20x slower than **RE#**.

6.2.3 Hidden Passwords (11 benchmarks). This category illustrates something that is very difficult to express in standard regex syntax, which causes the pattern to be very large and slow to handle for most engines. **RE#** uses *intersection* to demonstrate how the performance does not degrade at the rate of other engines that use *union* to express an equivalent pattern but at a *factorial* cost. The main regex is an intersection of constraints, where the match must contain at least one character in all of `[0-9]`, `[a-z]`, `[A-Z]`, `[!-/]`, and the password must have a certain length that varies throughout the benchmark. To simplify, all of the inputs here have been limited to ASCII.

This benchmark, see Figure 8, illustrates the same principle as the *monster* regexes, where the performance of the engine does not degrade at the rate of other engines. While re2 and rust/regex

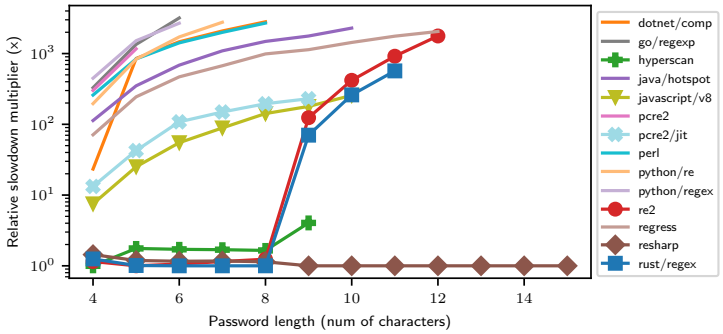


Fig. 8. Searching for hidden passwords of increasing length.

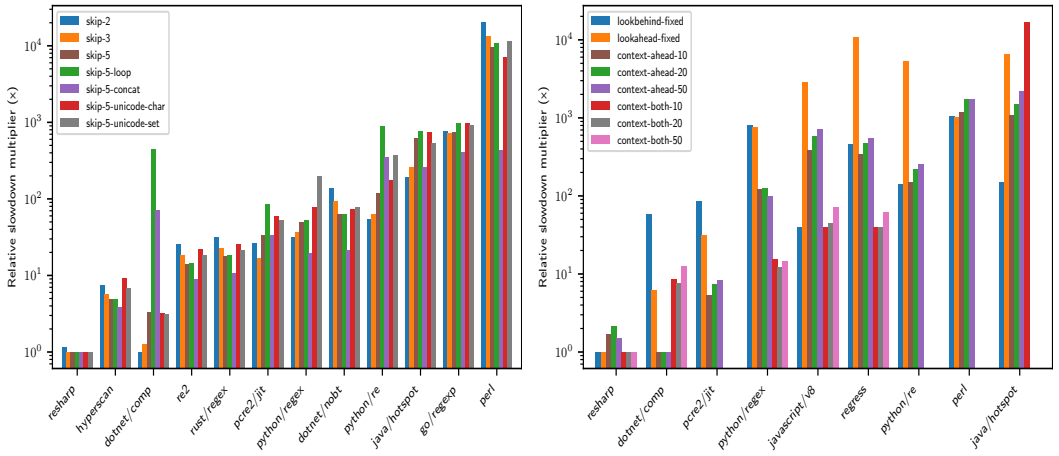
are able to handle the pattern up to 8 characters, both of the engines hit a wall at 9 characters, where the performance of the engines drops significantly. The performance of Hyperscan is also dropping at 9 characters, but it stops accepting the pattern at 10 characters. The search-time performance of **RE#** is still reasonable at 15 characters and above, and the throughput of the engine is still in the hundreds of MB/s. Without intersections, the performance of **RE#** would also hit a wall soon after the others, as the state space of the automaton grows at an exponential rate, but using intersections allows to keep the automaton small and the performance of the engine high.

6.2.4 Long Matches (7 benchmarks). This category is designed to test the engines' ability to accelerate long matching patterns, see Figure 9a. The input used in this category consists of long lines, averaging around 3000 characters in length. The patterns used in the category are designed so that the engine has to scan the entire line to find a match, but the engine has many opportunities to skip characters during the inner loop of matching.

An interesting observation from this category is that many of the engines have one-off optimizations for long matches, where certain patterns with the exact same language are significantly faster than others. For example the pattern used in the *skip-5* benchmark, `(?m)^.*1.*1.*1.*1.*1.*$` is 120x faster than the pattern in the *skip-5-loop* benchmark `(?m)^.*(1.*){5}$` for dotnet/comp, as optimizations get detected and applied in the former, but not in the latter. The same is true for python/re and pcre2/jit which both lose performance noticeably with the loop variant of the pattern. **RE#** actually loses one benchmark in this category, the *skip-2* benchmark with the pattern `(?m)^.*1.*1.*$`, where dotnet/comp vectorizes the first part of the pattern as well. The benchmarks *skip-3* and *skip-5* show that this behavior does not apply to the remainder of the pattern, as the performance of dotnet/comp drops significantly with the number of skips.

The reason why **RE#** is able to outperform the other engines in this benchmark is that derivatives allow the engine to cheaply infer which transitions are redundant, which lets the engine use input-parallelism to skip over large parts of the input, which gives the engine an advantage of an order of magnitude over most other engines in this category.

6.2.5 Character Sets and Unicode (7 benchmarks). This category shows the performance of symbolic character sets in **RE#**, i.e., the power of $\mathcal{A}$. The patterns used in this category are to find words containing a certain character set, such as `\b\w*[abc]w*\b`. To add a layer of complexity, both the input and character set are unicode characters, which makes the pattern difficult to handle for most engines, see Figure 2b.



(a) Long matches (Section 6.2.4). (b) Lookarounds (Section 6.2.6).

Fig. 9. Long and Lookaround benchmarks. *y*-axis is relative slowdown in *log* scale.

The first two benchmarks *word-vowels-1* and *word-vowels-2*, illustrate how **RE#** is able to use the derivative-based framework to incorporate vectorized character set matching into simple word patterns and be an order of magnitude faster than the other engines.

The *word-vowels-3 and word-vowels-4* benchmarks are more complex both in terms of number of matches and the complexity of the pattern, where the performance of **RE#** is still very good, but not as dominant as in the first two benchmarks.

The *word-vowels-5-to-digits and word-digits-to-vowels-5* benchmarks illustrate that this character set efficiency works in both directions, where certain engines (e.g., python/re and dotnet/comp) locate the digits-first variant significantly faster than the vowels, but the throughput for **RE#** is nearly identical for both. The *many-set-constraints* benchmark illustrates a more complex scenario, where the engine has to find many of these set constraints in a single match, which severely slows several engines down, but **RE#** is still able to maintain a throughput of 757.7MB/s, where most other engines are in KB/s.

6.2.6 Lookarounds (8 benchmarks). The lookarounds category illustrates that all the optimizations apply to lookarounds in **RE#** as well. There are many benchmarks in this category where the performance of the engine is several orders of magnitude faster than in other engines, see Figure 9b. Note that certain engines (e.g. pcre2/jit) are omitted from the context-both benchmarks as they do not support unbounded lookbehinds.

The *lookbehind-fixed* and *lookahead-fixed* benchmarks demonstrate the efficacy of simple string literal prefilter optimizations, e.g. those in Section 5.2, where the engine is able to vectorize the search for patterns containing both prefix and suffix lookarounds, which makes **RE#** several orders of magnitude faster than the other engines.

While Figure 3 showed that the performance of lookbehinds is linear for all matches, the same is not true for lookaheads. The only guarantee for lookaheads is that a single match is input-linear, but the performance of the engine degrades if there are multiple matches depending on the same lookahead context, which is located far away from the match. The *context-ahead* benchmarks illustrate this, where the performance of the engine is slightly behind dotnet/comp, as both of the engines suffer from quadratic all-matches behavior in this case.

Despite this, the performance of **RE#** is still very good in this category compared to the rest of the engines, where there is a noticeable lack of optimizations for lookarounds. How to eliminate quadratic all-matches behavior for lookaheads is a topic for future work.

The *context-both* benchmarks illustrate the scenario where each match is dependent on both a lookbehind and a lookahead, where **RE#** has a lead similar to Figure 3, as the linear time *all-matches* complexity of lookbehinds is not applicable to other engines.

6.3 Optimizations and Overall Effect on Performance

The performance of the engine is a result of a combination of many optimizations, which are described in the paper. Out of the list of optimizations mentioned, the two with the most significant influence on the performance of the engine are:

- DFA memory efficiency, enabling the strong worst-case $O(n)$ search-time complexity.
- Opportunistic algorithms reaching beyond standard $O(n)$ DFA speeds, e.g. vectorization.

Many benchmarks are designed around string literals, which are necessary to optimize for, as they are the most common comparison of regex engines, where most implementations have some kind of vectorized optimizations in place. Simply having a search-time $O(n)$ DFA algorithm is not enough to compete with the best engines here, as the performance is dependent on clever parallel algorithms and the ability to optimize the search for string literals, which we have shown in the *literal* and *literal-alternate* categories in Section 6.1.1. We emphasize that our approach to vectorization is general and applies to a large class of regexes, including those with lookarounds, as shown in Sections 6.1.4, 6.2.4 and 6.2.6.

In the cases where context-sensitive features e.g. anchors, lookarounds, are necessary, the engine boasts a unique *zero-cost* implementation of the word boundary `\b` – represented via negative lookarounds in the engine and encoded directly into DFA transitions, which is highlighted in the *words* benchmark in Section 6.1.2.

To a large extent, the real performance of the engine is due to its memory efficiency, which it owes to algebraic rewrites and redundancy elimination. This efficiency becomes most evident in large patterns, as there are more opportunities for rewrites. It is nontrivial to consistently simplify a large set of active states while also detecting opportunities to optimize, which is achieved in RE#. The memory efficiency of the engine is also partly due to alphabet compression, as done in [Moseley et al. 2023].

In the *monster* regex category in Section 6.2.2, the engine is able to dominate the competition, as it is able to detect and eliminate redundant states in the automaton. RE# explicitly uses a search-time $O(n)$ algorithm not an $O(m * n)$ algorithm, and not having to fall back to the $O(m * n)$ algorithm in complex scenarios such as shown in Section 6.2.2 is only possible due to the memory efficiency gained from rewrites and detection of redundant states in derivatives. The fact that all transitions in the automaton can be cached, including those on lookarounds, is a major contributor to the performance of the engine, as it allows to maximize the throughput of the engine by reusing the transitions created.

A key design decision in the engine is to give strong guarantees on the expected performance similar to e.g. Hyperscan, which has many lesser-known restrictions, such as restricted anchors with unicode support, yet guarantees consistently high performance. In comparison to recent work on linear complexity lookarounds, such as [Mamouras and Chattopadhyay 2024] or [Barrière and Pit-Claudel 2024] our work is less general, but our performance target is in line with the very top, which is enabled by some features, such as the ability to cache transitions on lookarounds for subsequent inputs and locating matches in strictly one reverse pass over the input, as opposed to one-or-more passes.

7 Related and Future Work

This work builds upon and uses the theory of location based derivatives introduced in [Moseley et al. 2023], and the implementation builds upon the open source .NET regular expression library [Microsoft 2022]. The match semantics supported in **RE#** is *leftmost-longest* (POSIX) rather than *leftmost-greedy* (a.k.a., *backtracking* or PCRE) semantics. It is unclear how to support extended Boolean operators in backtracking in the first place and what their intended semantics would be – this is primarily related to that `|` is *non-commutative* in the backtracking semantics and therefore some key distributivity laws such as $X(Y|Z) \equiv XY|XZ$ no longer preserve match semantics. For example, in PCRE `(a|ab)(c|b)` matches the prefix `"ab"` of `"abc"` but `(a|ab)c|(a|ab)b` matches the whole string `"abc"`. Consequently, many rewrite rules based on derived Boolean laws, such as SUB1 and LOOP in Figure 6, become invalid in PCRE.

In functional programming derivatives were studied in [Fischer et al. 2010; Owens et al. 2009] for *IsMatch*. [Ausaf et al. 2016; Sulzmann and Lu 2012] study matching with Antimirov derivatives and POSIX semantics and also Brzozowski derivatives in [Ausaf et al. 2016] with a formalization in Isabelle/HOL. The algorithm of [Sulzmann and Lu 2012] has been recently further studied in [Tan and Urban 2023; Urban 2023]. It is also mentioned in [Urban 2023, p.22] that *reversal*, as used in [Moseley et al. 2023], is not directly applicable in the context of the [Sulzmann and Lu 2012] algorithm. Partial derivatives of regular expressions extended with complement and intersection have also been studied in [Caron et al. 2011]. These works do not support lookarounds (or anchors). The key difference with the work of *transition regexes* used in SMT [Stanford et al. 2021] is that the theory of transition regexes does not support lookarounds. However, generalizing transition regexes to location based derivatives is an interesting direction for future work.

The conciseness of using intersection and complement in regular expressions is demonstrated in [Gelade and Neven 2012] where the authors show that using intersection and complement in regular expressions can lead to a double exponentially more succinct representation of regular expressions. Here we have experimentally shown how the enriched expressivity can enable practical scenarios for matching that are otherwise not possible.

Regular expressions have in practice many extensions, such as *backreferences* and *balancing groups*, that reach far beyond *regular* languages in their expressive power. Such extensions, see [Loring et al. 2019], fall outside the scope of **RE#**. Lookaheads do maintain regularity [Morihata 2012] and regular expressions with lookaheads can be converted to Boolean automata [Berglund et al. 2021b]. [Chida and Terauchi 2023] consider extended regular expressions in the context of backreferences and lookaheads. They build on [Carle and Narendran 2009] to show that extended regular expressions involving backreferences and both positive and negative lookaheads leads to *undecidable* emptiness, but, when restricted to positive lookaheads only is closed under complement and intersection. [Miyazaki and Minamide 2019] present an approach to finding match end with derivatives in regular expressions with lookaheads using *Kleene algebras with lookahead* as an extension of Kleene algebras with tests [Kozen 1997] where the underlying semantic concatenation is *commutative* and *idempotent* – it is unclear how lookbehinds and reversal fit in here. Derivatives combined with Kleene algebras are also studied in [Pous 2015].

Some aspects of our work here are related to SRM [Saarikivi et al. 2019] that is the predecessor of the NONBACKTRACKING regex backend of .NET [Moseley et al. 2023], but SRM lacks support for lookarounds as well as anchors and is neither POSIX nor PCRE compliant. Intersection was also included as an experimental feature in the initial version of SRM by building directly on derivatives in [Brzozowski 1964], and used an encoding via regular expression *conditionals* that unfortunately conflicts with the intended semantics of conditionals and therefore has, to the best of our knowledge, never been used or evaluated.

State-of-the-art nonbacktracking regex matchers based on automata such as RE2 [Cox 2010] and grep [GNU 2023] using variants of [Thompson 1968], and Hyperscan [Wang et al. 2019] using a variant of [Glushkov 1961], as well as the derivative based NonBacktracking engine in .NET make heavy use of *state graph memoization*. None of these engines currently support lookarounds, intersection or complement. A general advantage of using derivatives is that they often minimize the state graph (but do not guarantee minimization), as was already shown in [Owens et al. 2009, Table 1] for DFAs. Similar discussion appears also in [Sulzmann and Lu 2012, Section 5.4] where NFA sizes are compared for Thompson's and Glushkov's, versus Antimirov's constructions, showing that Antimirov's construction consistently yields a smaller state graph. Further comparison with automata based engines appears in [Moseley et al. 2023].

The two main standards for matching are PCRE (backtracking semantics) and POSIX [Berglund et al. 2021a; Laurikari 2000]. *Greedy* matching algorithm for backtracking semantics was originally introduced in [Frisch and Cardelli 2004], based on ϵ-NFAs, while maintaining matches for eager loops. In the current work we focused on the expressivity of a *single* regular expression. Compared to lookarounds, there are different approaches to achieving contextual information, e.g. it can be done programmatically by matching multiple regular expressions, or by the use of transducers, e.g. Kleenex [Grathwohl et al. 2016], that can produce substrings in context at high throughput rates.

The theory of derivatives based on locations that is developed here can potentially be used to extend regular expressions with lookarounds in SMT solvers that support derivative based lazy exploration of regular expressions as part of the sequence theory, such solvers are CVC5 [Barbosa et al. 2022; Liang et al. 2015] and Z3 [de Moura and Bjørner 2008; Stanford et al. 2021]. A further extension is to lift the definition of location derivatives to a fully *symbolic* form as is done with *transition regexes* in Z3 [Stanford et al. 2021]. [Chen et al. 2022] mention that the OSTRICH string constraint solver could be extended with backreferences and lookaheads by some form of alternating variants of prioritized streaming string transducers (PSSTs), but it has, to our knowledge, not been done. Such extensions would widen the scope of analysis of string verification problems that arise from applications that involve regexes using anchors and lookarounds. It would then also be beneficial to extend the SMT-LIB [SMT-LIB 2021] format to support lookarounds.

Counters are a well-known Achilles heel of essentially all nonbacktracking state-of-the-art regular expression matching engines as recently also demonstrated in [Turoňová et al. 2022], which makes any algorithmic improvements of handling counters highly valuable. In [Turoňová et al. 2020], Antimirov-style derivatives [Antimirov 1996] are used to extend NFAs with counting to provide a more succinct symbolic representation of states by grouping states that have similar behavior for different stages of counter values together using a data-structure called a *counting-set*. It is an intriguing open problem to investigate if this technique can be adapted to work with location derivatives within our current framework. [Glaunec et al. 2023] point out that it is important to optimize specific steps of regular expression matching to address particular performance bottlenecks. The specific BVA-Scan algorithm is aimed at finding matches with regular expressions containing counters more efficient. [Holík et al. 2023] report on a subset of regexes with counters called synchronizing regexes that allow for fast matching.

Recently [Mamouras and Chattopadhyay 2024] presented a new algorithm for matching lookarounds with Oracle NFAs, which are essentially cached queries to the oracle that can be used to match lookarounds. The semantics presented in their paper is consistent with the semantics of $\mathbf{RE}_{\leq}$, as described in [Moseley et al. 2023, Section 3.7] using derivation relations, for example, $\langle s[i], s[j]\rangle \models (?{=}R)$ iff $i = j$ and $s[i] \xrightarrow{R\cdot\_^*} s[|s|]$. We could not find any implementation of the algorithm, but it would be interesting to compare the lookaround approaches.

[Barrière and Pit-Claudel 2024] present a new nonbacktracking algorithm for matching JavaScript regular expressions with lookarounds in linear time. The first steps of the algorithm construct an oracle similar to the one in [Mamouras and Chattopadhyay 2024], but leveraging JavaScript semantics for the unique capability of matching capture groups both in the main regex and lookarounds in linear time. The algorithm is implemented as an NFA engine, whereas **RE#** is a lazy DFA engine. The fragment $\mathbf{RE}_{\leq}$ of regexes is orthogonal to the regexes in **RE#**: while intersection and complement are not allowed, at the same time lookarounds can be used freely in any context. But there is a common subset of regexes of the form `(?<=`R_1`)`R_2`(?=`R_3`)` where $R_i \in \mathbf{RE}$ on which it would be interesting to see how the two approaches compare in practice. Moreover, *alternations* of such regexes are currently not supported in **RE#** but their support is ongoing work.

The *full* fragment $\mathbf{RE}_{\leq}$ is supported in both of the above works, including *nested* lookarounds. In particular, the new `--enable-experimental-regexp-engine` flag in Javascript V8 supports nested lookarounds. We believe that **RE#** could support some fragment of nested lookarounds in a similar manner to Javascript V8.

Supporting capture groups in **RE#** is an interesting extension for future work, which could potentially be done as in [Moseley et al. 2023] that is related to tagged-NFA's [Laurikari 2000], or building on the works in [Barrière and Pit-Claudel 2024; Mamouras and Chattopadhyay 2024]. A fundamental challenge in **RE#** is to first specify the *intended semantics* of capture groups that overlap in an intersection.

8 Conclusion

We have presented both a theory and an implementation for extended regular expressions including complement, intersection and positive and negative lookarounds that have not previously been explored in depth in such a combination. Prior work has analyzed different other sets of extensions and their properties, but several such combinations veer out of the scope of regular languages.

We have demonstrated the practicality of the class **RE#** and the power of algebraic simplification rules through derivatives. We have included extensive evaluation using popular benchmarks and compared to industrial state-of-the-art engines that come with decades of expert level automata optimizations, where **RE#** shows 71% improvement over the fastest industrial matcher today, already for the *baseline*, while enabling reliable support for features out of reach for all other engines. There are also many interesting open problems and extensions remaining.

We expect that these new insights will change how regular expressions are perceived and the landscape of their applications in the future. Potentially enabling new applications in LLM prompt engineering frameworks, new applications in medical research and bioinformatics, and new opportunities in access and resource policy language design by web service providers, where regexes are an integral part but today limited to *very restricted fragments of* **RE** due to their application in security critical contexts with high reliability requirements in policy engines.

Acknowledgments

We thank all the anonymous reviewers for their detailed and valuable comments and numerous helpful suggestions.

Artifact Availability

The artifact [Varatalu 2024a] covers all the results reported in Section 6. The .NET library for **RE#** is available as a nuget package [Varatalu 2024c]. The online web application [Varatalu 2024b] provides an interactive experience with the features of **RE#** and includes several examples.

References

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

Fahad Ausaf, Roy Dyckhoff, and Christian Urban. 2016. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Interactive Theorem Proving, 7th International Conference, ITP 2016* (Nancy, France) *(LNCS, Vol. 9807)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer, Cham, 69–86. https://doi.org/10.1007/978-3-319-43144-4_5

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022* (Munich, Germany) *(LNCS, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, Cham, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Aurèle Barrière and Clément Pit-Claudel. 2024. Linear Matching of JavaScript Regular Expressions. *Proc. ACM Program. Lang.* 8, Article 201 (June 2024), 25 pages. https://doi.org/10.1145/3656431

Martin Berglund, Willem Bester, and Brink van der Merwe. 2021a. Formalising and implementing Boost POSIX regular expression matching. *Theoretical Computer Science* 857 (2021), 147–165. https://doi.org/10.1016/j.tcs.2021.01.010

Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. 2021b. Regular Expressions with Lookahead. *Journal of Universal Computer Science* 27, 4 (2021), 324–340. https://doi.org/10.3897/jucs.66330

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

Benjamin Carle and Paliath Narendran. 2009. On Extended Regular Expressions. In *Language and Automata Theory and Applications, Third International Conference, LATA 2009* (Tarragona, Spain) *(LNCS, Vol. 5457)*, Adrian-Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide (Eds.). Springer, Berlin, Heidelberg, 279–289. https://doi.org/10.1007/978-3-642-00982-2_24

Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. 2011. Partial Derivatives of an Extended Regular Expression. In *Language and Automata Theory and Applications - 5th International Conference, LATA 2011* (Tarragona, Spain) *(LNCS, Vol. 6638)*, Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide (Eds.). Springer, Berlin, Heidelberg, 179–191. https://doi.org/10.1007/978-3-642-21254-3_13

Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving String Constraints with Regex-Dependent Functions through Transducers with Priorities and Variables. *Proc. ACM Program. Lang.* 6, POPL, Article 45 (jan 2022), 31 pages. https://doi.org/10.1145/3498707

Nariyoshi Chida and Tachio Terauchi. 2023. On Lookaheads in Regular Expressions with Backreferences. *IEICE Trans. Inf. Syst.* 106, 5 (2023), 959–975. https://doi.org/10.1587/transinf.2022edp7098

Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html

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

James C. Davis. 2019. Rethinking Regex Engines to Address ReDoS. In *Proceedings of ESEC/FSE'19* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 1256–1258. https://doi.org/10.1145/3338906.3342509

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of ESEC/FSE'18* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)* (Budapest, Hungary) *(LNCS, Vol. 4963)*. Springer, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

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

Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *Automata, Languages and Programming (ICALP'04)* (Turku, Finland) *(LNCS, Vol. 3142)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, Berlin, Heidelberg, 618–629. https://doi.org/10.1007/978-3-540-27836-8_53

Andrew Gallant. 2024. BurntSushi: rebar. https://github.com/BurntSushi/rebar

Wouter Gelade and Frank Neven. 2012. Succinctness of the Complement and Intersection of Regular Expressions. *ACM Trans. Comput. Log.* 13, 1 (2012), 4:1–4:19. https://doi.org/10.1145/2071368.2071372

Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching using Bit Vector Automata. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 492–521. https://doi.org/10.1145/3586044

Victor Mikhailovich Glushkov. 1961. The abstract theory of automata. *Russian Math. Surveys* 16 (1961), 1–53. https://doi.org/10.1070/RM1961v016n05ABEH004112

GNU. 2023. grep. https://www.gnu.org/software/grep/.

Google. 2024. RE2. https://github.com/google/re2.

Niels Bjørn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm, and Sebastian Paaske Tørholm. 2016. Kleenex: compiling nondeterministic transducers to deterministic streaming transducers. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016* (St. Petersburg, FL, USA), Rastislav Bodík and Rupak Majumdar (Eds.). ACM, USA, 284–297. https://doi.org/10.1145/2837614.2837647

Lukás Holík, Juraj Síc, Lenka Turonová, and Tomás Vojnar. 2023. Fast Matching of Regular Patterns with Synchronizing Counting. In *Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 2023* (Paris, France) *(LNCS, Vol. 13992)*, Orna Kupferman and Pawel Sobocinski (Eds.). Springer, Cham, 392–412. https://doi.org/10.1007/978-3-031-30829-1_19

Alec Koumjian. 2024. akoumjian: datefinder. https://github.com/akoumjian/datefinder

Dexter Kozen. 1997. Kleene algebra with tests. *TOPLAS* 19, 3 (1997), 427–443. https://doi.org/10.1145/256167.256195

Ville Laurikari. 2000. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *7th International Symposium on String Processing and Information Retrieval* (A Curuna, Spain). IEEE, Piscataway, NJ, USA, 181–187. https://doi.org/10.1109/SPIRE.2000.878194

Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2015. A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings?. In *Frontiers of Combining Systems, FroCoS 2015* (Wroclaw, Poland) *(LNCS/LNAI, Vol. 9322)*. Springer, Cham, 135–150. https://doi.org/10.1007/978-3-319-24246-0_9

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'19* (Phoenix, AZ, USA). ACM, New York, NY, USA, 425–438. https://doi.org/10.1145/3314221.3314645

Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proc. ACM Program. Lang.* 8, POPL (2024), 2761–2791. https://doi.org/10.1145/3632934

Robert McNaughton and Hisao Yamada. 1960. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers* EC-9 (1960), 39–47. https://doi.org/10.1109/TEC.1960.5221603

Microsoft. 2021a. CredScan. https://secdevtools.azurewebsites.net/helpcredscan.html.

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

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

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

Akimasa Morihata. 2012. Translation of Regular Expression with Lookahead into Finite State Automaton. *Computer Software* 29, 1 (2012), 147–158. https://doi.org/10.11309/jssst.29.1_147

Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. In *PLDI '23: 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Orlando, FL, USA), Nate Foster et al. (Ed.). ACM, New York, NY, USA, 1026–1049. https://doi.org/10.1145/3591262

OISF. 2024. Suricata. https://suricata.io/

OWASP. 2024. Regular expression Denial of Service - ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS

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

Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. *ACM SIGPLAN Notices – POPL'15* 50, 1 (2015), 357–368. https://doi.org/10.1145/2775051.2677007

Kun Qiu, Harry Chang, Yang Hong, Wenjun Zhu, Xiang Wang, and Baoqian Li. 2021. Teddy: An Efficient SIMD-based Literal Matching Engine for Scalable Deep Packet Inspection. In *ICPP 2021: 50th International Conference on Parallel Processing* (Lemont, IL, USA), Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen (Eds.). ACM, USA, 62:1–62:11. https://doi.org/10.1145/3472456.3473512

Rust. 2024. The Rust Programming Language: regex. https://github.com/rust-lang/regex

Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'19* (Prague, Czech Republic) *(LNCS, Vol. 11427)*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer, Cham, 372–378. https://doi.org/10.1007/978-3-030-17462-0_24

SMT-LIB. 2021. The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu/

Henry Spencer. 1994. A Regular-expression Matcher. In *Software Solutions in C*. Academic Press Professional, Inc., USA, 35–71. https://dl.acm.org/doi/10.5555/156626.184689

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual Event). ACM, New York, NY, USA, 620–635. https://doi.org/

10.1145/3453483.3454066

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

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

Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (jun 1968), 419–422. https://doi.org/10.1145/363347.363387

Stephen Toub. 2024. Performance Improvements in .NET 9. Microsoft .NET Blog. https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-9/

Lenka Turoňová, Lukáš Holík, Ivan Homoliak, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. 2022. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4165–4182. https://www.usenix.org/conference/usenixsecurity22/presentation/turonova

Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex Matching with Counting-Set Automata. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 218 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428286

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

Ian Erik Varatalu. 2024a. Artifact for this paper. https://doi.org/10.5281/zenodo.13937348

Ian Erik Varatalu. 2024b. RE# Interactive. https://ieviev.github.io/resharp-webapp/

Ian Erik Varatalu. 2024c. Resharp. https://www.nuget.org/packages/Resharp

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

Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. 2024. Lean Formalization of Extended Regular Expression Matching with Lookarounds. In *13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'24)* (London, UK). ACM, New York, NY, USA, 118–131. https://doi.org/10.1145/3636501.3636959

Received 2024-07-10; accepted 2024-11-07