Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics

DAN MOSELEY, Microsoft, USA
MARIO NISHIO, Microsoft, USA
JOSE PEREZ RODRIGUEZ, Microsoft, USA
OLLI SAARIKIVI, Microsoft, USA
STEPHEN TOUB, Microsoft, USA
MARGUS VEANES, Microsoft, USA
TIKI WAN, Microsoft, USA
ERIC XU, Microsoft, USA

We develop a new derivative based theory and algorithm for nonbacktracking regex matching that supports anchors and counting, preserves backtracking semantics, and can be extended with lookarounds. The algorithm has been implemented as a new regex backend in .NET and was extensively tested as part of the formal release process of .NET7. We present a formal proof of the correctness of the algorithm, which we believe to be the first of its kind concerning industrial implementations of regex matchers. The paper describes the complete foundation, the matching algorithm, and key aspects of the implementation involving a regex rewrite system, as well as a comprehensive evaluation over industrial case studies and other regex engines.

CCS Concepts: • **Theory of computation** → **Regular languages**; • **Computing methodologies** → **Symbolic and algebraic algorithms**.

Additional Key Words and Phrases: regex, derivative, symbolic, matching, automata, PCRE

ACM Reference Format:
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. *Proc. ACM Program. Lang.* 7, PLDI, Article 148 (June 2023), 24 pages. https://doi.org/10.1145/3591262

1 INTRODUCTION

Regular expressions play a central role in many software applications and are supported by the standard libraries of most popular programming languages. Several studies have shown that serious problems can be triggered in many matching engines through regular expression denial of service (ReDoS) attacks [OWASP 2020] as a direct result of excessive backtracking [Davis 2019; Davis et al. 2018]. Matching engines commonly use backtracking [Spencer 1994] to support *non-regular* features – such as backreferences and balancing groups – which make the language Turing complete in general. These engines may then exhibit behavior that is quadratic or exponential in the length of the input even for regular expressions *without* non-regular features, because of the generality of

Authors' addresses: Dan Moseley, Microsoft, USA, danmose@microsoft.com; Mario Nishio, Microsoft, USA, mario.nishio@microsoft.com; Jose Perez Rodriguez, Microsoft, USA, joperezr@microsoft.com; Olli Saarikivi, Microsoft, USA, olsaarik@microsoft.com; Stephen Toub, Microsoft, USA, stoub@microsoft.com; Margus Veanes, Microsoft, USA, margus@microsoft.com; Tiki Wan, Microsoft, USA, tikiwan@microsoft.com; Eric Xu, Microsoft, USA, ericxu@microsoft.com.

2475-1421/2023/6-ART148
https://doi.org/10.1145/3591262

the runtime. When exposed as part of a web service these performance issues can be exploited to cause outages [Baldwin 2016; Graham-Cumming 2019; Stack Exchange 2016].

We have developed a new algebraic framework for regular expression matching based on *derivatives* that offers *input-linear performance*, has *clear foundations*, supports a large set of *industrially relevant features*, and *is compatible with backtracking semantics.* While derivatives have been well studied in the past [Ausaf et al. 2016; Fischer et al. 2010; Owens et al. 2009; Sulzmann and Lu 2012], how to unify them with anchors, counters, and backtracking (PCRE) semantics, has not. What makes this combination especially challenging is that certain classical properties of regular languages such as $L(R{\cdot}S) = L(R){\cdot}L(S)$ no longer hold when anchors are present. Non-regular features are excluded to enable finite-state based techniques. The work presented here extends the open source SRM library [Saarikivi et al. 2019] and is integrated as a new backend in .NET's System.Text.RegularExpressions library. It is extensively tested, open-source, and ships with .NET7.[1] Since .NET is one of the main developer platforms worldwide[2] we expect this new backend to benefit many applications where predictable performance is critical.

SRM *does not support anchors* and, more importantly, lacks the foundations needed to do so; *does not support eager or lazy* interpretation of loops or *order* of alternations, and is therefore *not compatible with the match results* that in .NET otherwise rely on *backtracking*; *lacks a general rewrite system* used to optimize regexes in a way that *preserves their backtracking semantics.*

These were serious hurdles to overcome. Our first attempt to treat anchors was as new imaginary (0-width) symbols. This approach is also discussed in [Wingbrant 2019]. While the idea seems promising, it does not work in .NET: e.g., the regexes `\A\z` and `\z\A` are equivalent, but a string where an imaginary start-of-input is followed by an end-of-input is only accepted by the first regex, or similarly with the regex `\b$` that is equivalent to `$\b` because the order of a sequence of anchors is immaterial in general. A key observation for supporting anchors is that matching is *context dependent.* A stark example of this is that a pattern consisting of just the word-boundary anchor `\b` has four *empty* matches at locations 0, 5, 6 and 11 in "`Hello World`", as shown by bold borders in ‖H|e|l|l|o‖ ‖W|o|r|l|d‖ where exactly one character in the immediate vicinity of that location must be a word-letter, e.g., location 11 is surrounded by "`d`" and "".[3] The word-border anchor `\b` occurs in almost all patterns in our case study of *word phrase* pattern matching.

Regarding *backtracking semantics*, consider the regex `a+?|a*` and the input "`aa`". If laziness of loops and order of alternatives is ignored then the regex is classically equivalent to `a*` in which case the *earliest* match end in "`aa`" is at location 0 and the *latest* match end is at location 2, while the *backtracking* match end of `a+?|a*` in "`aa`" is at location 1. This illustrates that prioritized alternatives cannot in general be reduced to earliest or latest match semantics in classical regular expressions and is thus an orthogonal feature of regexes. The classically valid rewrite rule $R\{0,m\}|R\{0,n\} \longrightarrow R\{0,\max(m,n)\}$ from SRM *does not* preserve backtracking semantics because the loops are eager (e.g., the match end of `a{0,1}|a{0,2}` in "`aa`" is at location 1 while the match end of `a{0,2}` in "`aa`" is at location 2). Preserving the same semantics for all regular expression backends in .NET is critical not only for *consistent user experience* but also for *implementation transparency*: runtime optimizations such as substituting a *backtracking* search engine by a *nonbacktracking* search engine in the absence of non-regular features could otherwise not be applied.[4]

Other popular nonbacktracking matching engines, such as RE2 [Google 2021] and grep [GNU 2020], can at a high level be considered efficient implementations of [Thompson 1968] enhanced

[1].NET7 shipped in November 2022.
[2]For example consult https://enlyft.com/tech/products/microsoft-net for recent market analysis.
[3]A *location* in a string is intuitively the position of a border *between* two characters or the position of the two outer borders.
[4]Such substitutions are currently not being applied in the .NET7 release due to other concerns but remain possible.

with on-the-fly determinization and memoization, or in the case of Hyperscan [Intel Co 2021] as a different variant of NFA simulation via [Glushkov 1961]. Such translation results in an NFA (either via Thompson's or Glushkov's construction) where the states meaning as "regular expressions" themselves has been *lost in translation*. Moreover, these engines do not have first-class support for counting. Instead, counters are unwound up-front by a pre-processor, prior to the NFA translation. Use of large counters in general is an *Achilles heel* of *all* state-of-the-art nonbacktracking regex matchers as a recent study demonstrates [Turoňová et al. 2022]. Derivatives decrease counters by one, and thus loops such as `a{0,`n`}` end up being expanded fully only when the upper bound n has been reached in the given input. Furthermore, in those engines both *anchors* and *backtracking semantics* have been treated as implementation concerns. In contrast, we formally prove that our derivative based matcher gives backtracking semantics in the presence of anchors.

Perhaps the most contrasting aspect of our algorithm is that the relationship between *states* and *regexes* they denote is *not lost in translation*. We lift character based derivatives [Brzozowski 1964] to *location based derivatives*, which allow for a natural treatment of anchors. For example, if the regex for the current state q is `^.*?(she|he)` and the current input location x has `s` as its current character and `\n` as its previous character, then the x-derivative of q is the regex `he|.*?(she|he)` – the start-of-line anchor `^` was nullable (non-blocking) due to change of line and an alternation has been created for the two ways the `h` may be consumed. The key benefit of maintaining the link between states and regexes is used in the *rewriting system* we develop to minimize regexes and the state machine their derivatives induce. In the example, if the next character is `h` then the derivative would evolve to `e|e|.*?(she|he)`, which rewriting would simplify to `e|.*?(she|he)`. Notably, we develop subsumption based rules for eliminating unnecessary alternation which we found critical for acceptable real-world performance on some classes of patterns.

Finally, as far as we know, *our algorithm is the first industrial implementation of input-linear regex matching that has a formal proof of correctness*. Moreover, it is continuously tested on the extensive regex test suite in .NET to semantically match the other backends on all the platforms that .NET supports. There is thus also compelling experimental evidence of mutual consistency of *all the backends* relative to the fragment *RE* of regexes (defined in Section 3.1) and their backtracking semantics (defined in Section 4.1). For detailed proofs of theorems see [Moseley et al. 2023a].

Summary of contributions:

New derivative based framework for regular expression matching with fully developed theory based on *location derivatives* with Theorem 3.3 as its main characterization, and the *reversal* Theorem 3.8 as a key corollary. We also extend the theory with *lookarounds*. (Section 3)

Derivative based backtracking simulation building on a conservative extension of the core framework through a *tail-recursive* formulation of backtracking through derivatives. This results in a new matching algorithm with proof of correctness in Theorem 4.5. (Section 4)

Industrial scale implementation involving a regex rewrite system with several key ideas of how the framework is being utilized for advanced optimizations. (Section 5)

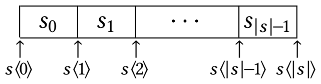
Comprehensive evaluation at industrial scale, validating the efficacy of this work. (Section 7)

2 PRELIMINARIES

Here we introduce background material used in the paper. As a general meta-notation throughout the paper we write $lhs \overset{\text{DEF}}{=} rhs$ to let *lhs* be *equal by definition* to *rhs*. Let $\mathbb{B} = \{\textbf{false}, \textbf{true}\}$ stand for basic Boolean values. Let Σ be a universe or domain of *characters*. We denote pairs of elements by $\langle x, y\rangle$ and let $\pi_1(\langle x, y\rangle) \overset{\text{DEF}}{=} x$ and $\pi_2(\langle x, y\rangle) \overset{\text{DEF}}{=} y$.

Strings. Let ϵ or "" denote the empty string and let Σ^* denote the set of all strings over Σ. Let $s \in \Sigma^*$. The length of s is denoted by $|s|$. We do not distinguish between individual characters and strings of length 1. Let i and l be nonnegative integers such that $i + l \leq |s|$. Then $s_{i,l}$ denotes the substring of s that starts from index i and has length l, where the first character has index 0. In particular $s_{i,0} = \epsilon$. For $0 \le i < |s|$ let $s_i \stackrel{\text{DEF}}{=} s_{i,1}$. Let also $s_{-1} = s_{|s|} \stackrel{\text{DEF}}{=} \epsilon$. E.g., "abcde"$_{1,3}$ = "bcd" and "abcde"$_{5,0} = \epsilon$. We denote the *reverse* of s by s^r, so that $s^r{}_i = s_{|s|-1-i}$ for $0 \le i < |s|$.

Locations. Let s be a string. A *location in* s is a pair $\langle s, i\rangle$, where $-1 \le i \le |s|$. We use $s\langle i\rangle \stackrel{\text{DEF}}{=} \langle s, i\rangle$ as a dedicated notation for locations, where s is the *string* and i the *position* of the location. Since $s\langle i\rangle$ is a pair, note also that $\pi_1(s\langle i\rangle) = s$ and $\pi_2(s\langle i\rangle) = i$. If x and y are locations, then $x < y$ iff $\pi_2(x) < \pi_2(y)$. A location $s\langle i\rangle$ is *valid* if $0 \le i \le |s|$. A location $s\langle i\rangle$ is called *final* if $i = |s|$ and *initial* if $i = 0$. Let $\mathit{Final}(s\langle i\rangle) \stackrel{\text{DEF}}{=} i = |s|$ and $\mathit{Initial}(s\langle i\rangle) \stackrel{\text{DEF}}{=} i = 0$. We let $\lightning \stackrel{\text{DEF}}{=} \epsilon\langle -1\rangle$ that is going to be used to represent *match failure* and in general $s\langle -1\rangle$ is used as a *pre-initial* location. The *reverse* $s\langle i\rangle^r$ of a valid location $s\langle i\rangle$ in s is the valid location $s^r\langle |s|-i\rangle$ in s^r. For example, the reverse of the final location in s is the initial location in s^r. When working with sets S of locations over the same string we let $\max(S)$ ($\min(S)$) denote the maximum (minimum) location in the set according to the location order above. In this context we also let $\max(\emptyset) = \min(\emptyset) \stackrel{\text{DEF}}{=} \lightning$ and $\lightning^r \stackrel{\text{DEF}}{=} \lightning$. Valid locations in a string s are illustrated by



and should be viewed as *border positions* rather than character positions.[5] For example, ϵ has only one valid location $\epsilon\langle 0\rangle$ that is both initial and final.

Boolean Algebras as Alphabet Theories. The tuple $\mathcal{A} = (\Sigma, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ is called a *Boolean algebra over* Σ where Ψ is a set of *predicates* that is closed under the Boolean connectives; $[\![\_]\!] : \Psi \to 2^\Sigma$ 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* or $\mathbf{SAT}(\varphi)$.

Character Classes. In all the examples below we let Σ stand for the standard 16-bit character set of Unicode[6] and use the .NET syntax [Microsoft 2021c] of regular expression character classes. For example, `[A-Z]` stands for all the Latin capital letters, `[0-9]` for all the Latin numerals, `\d` for all the decimal digits, `\w` for all the word-letters and dot (`.`) for all characters besides the *newline character* `\n`. (It is a standard convention that, by default, dot does not match `\n`.)

When we need to distinguish the concrete representation of character classes from the corresponding abstract representation of predicates in $\mathcal{A}$ we map each character class C to the corresponding predicate ψ_C in Ψ. For example $\psi_{\texttt{[\^{}0-9]}} \equiv \neg\psi_{\texttt{[0-9]}}$, $\psi_{\texttt{[\textbackslash w-[\textbackslash d]]}} \equiv \psi_{\texttt{\textbackslash w}} \wedge \neg\psi_{\texttt{\textbackslash d}}$, and $\psi_{\texttt{\textbackslash W}} \equiv \neg\psi_{\texttt{\textbackslash w}}$. Observe also that $[\![\psi_{\texttt{[0-9]}}]\!] \subsetneq [\![\psi_{\texttt{\textbackslash d}}]\!] \subsetneq [\![\psi_{\texttt{\textbackslash w}}]\!]$ and $[\![\psi_{\texttt{\textbackslash n}}]\!] = \{\texttt{\textbackslash n}\}$ and $\texttt{\textbackslash n} \notin [\![\psi_{\texttt{\textbackslash w}}]\!]$ because `\n` is not a word-letter. Regarding $\bot$ and $\top$ it holds e.g., that $\bot \equiv \psi_{\texttt{[0-[0]]}}$ and $\top \equiv \psi_{\texttt{[\textbackslash 0-\textbackslash uFFFF]}}$.

3 REGEXES AND LOCATION DERIVATIVES

Here we formally define regular expressions with *anchors* and *loops* supporting finite and infinite bounds as well as lazy and eager interpretations. Regexes are defined modulo a character theory $\mathcal{A} = (\Sigma, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ that we illustrate with standard (.NET Regex) character classes in examples, but it is important to keep in mind that $\mathcal{A}$ itself is abstract and later, in Section 5, used in two distinct forms, both of which are independent of the concrete syntax of character classes.

After the definition of regexes, we formally develop a framework of *derivatives* that leads to the key notion of *derivation relation* between locations that is instrumental in reasoning and proving properties in this framework. We also define *reversal* of regexes and prove the main reversal theorem that is later used in Section 4 to prove correctness of the complete matching algorithm.

[5]This intuition fits well with the semantics of anchors and matching, and is also helpful in maintaining symmetry between locations and reversed locations.
[6]Also known as *Plane 0* or the *Basic Multilingual Plane* of Unicode.

The framework developed in this section does not depend on *laziness* of loops or the *order* of alternatives in an alternation, but is extended conservatively to take order in account in Section 4 where backtracking semantics is formally defined. We also make a remark about how the framework can be extended with *lookarounds*, demonstrating its flexibility and generality.

3.1 Regexes

The class *RE* of regular expressions or *regexes* as used in this paper is defined by the following abstract grammar. Let $\psi \in \Psi$, and $0 \le m \le n \ne 0$, or $n = \infty$, and $b \in \mathbb{B}$:

$$\begin{array}{rcl} RE & ::= & ⚓ \mid \psi \mid \texttt{()} \mid RE{\cdot}RE \mid RE|RE \mid RE\{m,n,b\} \\ ⚓ & ::= & \texttt{\\A} \mid \texttt{\\z} \mid \texttt{^} \mid \texttt{\$} \mid \texttt{\\Z} \mid \texttt{\\a} \mid \texttt{\\b} \mid \texttt{\\B} \end{array}$$

Elements of ⚓ are called *anchors* and have the names: *start* (\A), *end* (\z), *start-of-line* (^), *end-of-line* ($), *final-end-of-line* (\Z), *initial-start-of-line* (\a), *word-border* (\b), *non-word-border* (\B).

The regex denoting *nothing* (the empty language) has a simple representation in *RE* as just the predicate $\bot$, so no dedicated syntax is needed.

Concatenation operator $\cdot$ is often implicit by using juxtaposition, and the *empty sequence* () is a *unit element* of concatenation, so that $\texttt{()}{\cdot}R = R{\cdot}\texttt{()} = R$. As is common, concatenation binds stronger than alternation. Both concatenation and alternation are *right associative* operators.

A *loop* $R\{m,n,b\}$ has *body* R and is called *lazy* if $b = \textbf{true}$ else *eager*. Its *lower bound* is m and *upper bound* is n. The loop is *infinite* when $n = \infty$, *finite* otherwise. While the upper bound in a finite loop must be nonzero, we let $R\{0,0,\_\} \overset{\text{DEF}}{=} \texttt{()}$ for convenience in recursive definitions.

We abbreviate an eager loop by $R\{m,n\}$ and a lazy loop by $R\{m,n\}?$ and we also use the standard shorthands $R{*}$ for $R\{0,\infty\}$ and $R{+}$ for $R\{1,\infty\}$, with $R{*}?$ and $R{+}?$ as their lazy versions. A loop $R\{m,m,\_\}$ is also denoted by $R\{m\}$. A finite eager loop $R\{m,n\}$ can be written equivalently as $R\{m\}{\cdot}(R|\texttt{()})\{n-m\}$, a finite lazy loop $R\{m,n\}?$ is equivalent to $R\{m\}{\cdot}(\texttt{()}|R)\{n-m\}$, an infinite eager loop $R\{m,\infty\}$ is equivalent to $R\{m\}{\cdot}R{*}$, and an infinite lazy loop $R\{m,\infty\}?$ is equivalent to $R\{m\}{\cdot}R{*}?$. These are useful simplifying normal forms when reasoning about properties of loops.

3.2 Nullability and Anchor-Contexts

The language semantics of regexes is in general *context dependent*. An important factor in defining the semantics is played by the immediately surrounding symbols of a matching substring of an input s being searched. Let x be a valid location. A regex being *nullable in* x means that it matches the *empty string*, that in general is context dependent. The *anchor-context* of x is $\widehat{x} \overset{\text{DEF}}{=} \langle \boldsymbol{\kappa}(x{-}1), \boldsymbol{\kappa}(x)\rangle$ where $\boldsymbol{\kappa}(y) \in \mathit{KIND} \overset{\text{DEF}}{=} \{\boldsymbol{\varepsilon}, \mathbf{n}, \mathbf{N}, \mathbf{o}, \mathbf{w}\}$. Intuitively *KIND* is an enum with $\boldsymbol{\varepsilon}$ = EOF, $\mathbf{n}$ = EOL, $\mathbf{N}$ = last-EOL, $\mathbf{w}$ = word-letter, $\mathbf{o}$ = other-character, and $\boldsymbol{\kappa}(y)$ is the kind of location y.

$$\boldsymbol{\kappa}(s\langle i\rangle) \overset{\text{DEF}}{=} \begin{cases} \boldsymbol{\varepsilon} & \textbf{if } i = -1 \textbf{ or } i = |s| \\ \mathbf{N} & \textbf{else if } s_i = \texttt{\\n} \textbf{ and } (i = 0 \textbf{ or } i = |s|-1) \\ \mathbf{n} & \textbf{else if } s_i = \texttt{\\n} \\ \mathbf{w} & \textbf{else if } s_i \in [\![\psi_{\texttt{\\w}}]\!] \\ \mathbf{o} & \textbf{otherwise}. \end{cases} \qquad \left(\begin{array}{c} \overbrace{\boxed{s_{i-1}}\boxed{s_i}}^{\widehat{x}=\langle\boldsymbol{\kappa}(x-1),\boldsymbol{\kappa}(x)\rangle} \\ \uparrow \\ x \end{array} \right)$$

Intuitively $\widehat{x}$ describes the kinds of the immediately surrounding symbols of x. Nullability of an anchor is now defined relative to $\widehat{x}$. Let $\mathit{Null}_x(A) \overset{\text{DEF}}{=} \mathit{Null}_{\widehat{x}}(A)$ for $A \in$ ⚓.

$$\begin{array}{rclrcl} \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{\\A}) & \overset{\text{DEF}}{=} & \kappa_1 = \boldsymbol{\varepsilon} & \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{\\z}) & \overset{\text{DEF}}{=} & \kappa_2 = \boldsymbol{\varepsilon} \\ \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{\\a}) & \overset{\text{DEF}}{=} & \kappa_1 \in \{\boldsymbol{\varepsilon}, \mathbf{N}\} & \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{\\Z}) & \overset{\text{DEF}}{=} & \kappa_2 \in \{\boldsymbol{\varepsilon}, \mathbf{N}\} \\ \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{^}) & \overset{\text{DEF}}{=} & \kappa_1 \in \{\boldsymbol{\varepsilon}, \mathbf{N}, \mathbf{n}\} & \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{\$}) & \overset{\text{DEF}}{=} & \kappa_2 \in \{\boldsymbol{\varepsilon}, \mathbf{N}, \mathbf{n}\} \\ \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{\\b}) & \overset{\text{DEF}}{=} & \kappa_1 = \mathbf{w} \Leftrightarrow \kappa_2 \ne \mathbf{w} & \mathit{Null}_{\langle\kappa_1,\kappa_2\rangle}(\texttt{\\B}) & \overset{\text{DEF}}{=} & \kappa_1 = \mathbf{w} \Leftrightarrow \kappa_2 = \mathbf{w} \end{array}$$

All the remaining cases extend the classical notion of nullability conservatively:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{Null}_x(\psi) & \overset{\text{DEF}}{=} & \textbf{false} & \mathit{Null}_x(()) & \overset{\text{DEF}}{=} & \textbf{true} \\ \mathit{Null}_x(R \mid S) & \overset{\text{DEF}}{=} & \mathit{Null}_x(R)\ \textbf{or}\ \mathit{Null}_x(S) & \mathit{Null}_x(R\cdot S) & \overset{\text{DEF}}{=} & \mathit{Null}_x(R)\ \textbf{and}\ \mathit{Null}_x(S) \\ \mathit{Null}_x(R\{m,n,\_\}) & \overset{\text{DEF}}{=} & m=0\ \textbf{or}\ \mathit{Null}_x(R) & & & \end{array}$$

Let $\mathit{Null}_\forall(R) \overset{\text{DEF}}{=} \forall x(\mathit{Null}_x(R) = \textbf{true})$ and $\mathit{Null}_\nexists(R) \overset{\text{DEF}}{=} \nexists x(\mathit{Null}_x(R) = \textbf{true})$. For example, the loop `.*` is always nullable, i.e., $\mathit{Null}_\forall(\texttt{.*}) = \textbf{true}$ while the concatenation `.*b` is never nullable, i.e., $\mathit{Null}_\nexists(\texttt{.*b}) = \textbf{true}$. Both properties are statically determined from the regex, essentially based on that nullability does not depend on any anchor. We consider some cases in Example 3.1.

Example 3.1. Consider the regex `^$` (that matches an empty line) and the string s = "IT\n\nIS" and find some valid location x in s s.t. $\mathit{Null}_x(\texttt{^\$}) = \textbf{true}$. Then such a location is $s\langle 3\rangle$ as marked bold in $\boxed{\texttt{I}\mid\texttt{T}\mid\texttt{\textbackslash n}\,\|\,\texttt{\textbackslash n}\mid\texttt{I}\mid\texttt{S}}$ – no other location both ends and starts a line. Consider now the regex `\b` and the same question. Then all such locations in s would be where exactly one of the surrounding character kinds is $\mathbf{w}$, as marked bold in $\boxed{\|\,\texttt{I}\mid\texttt{T}\,\|\,\texttt{\textbackslash n}\mid\texttt{\textbackslash n}\,\|\,\texttt{I}\mid\texttt{S}\,\|}$ (locations $s\langle 0\rangle$, $s\langle 2\rangle$, $s\langle 4\rangle$, and $s\langle 6\rangle$). ⊠

3.3 Derivatives and Match End Location

In contrast to a classical definition of a complete string being accepted or matched by a regex, here the matching of a *substring* $s_{i,n}$ of s depends on its *surrounding* locations $s\langle i\rangle$ and $s\langle i+n\rangle$ in s. We lift the classical definition of the *derivative* [Brzozowski 1964] $D_a(R)$ for a *character* a to the *derivative* $\mathit{Der}_x(R)$ for a valid nonfinal location x. This extension is *conservative* so that when R does not use anchors then the acceptance condition of a string s is classically preserved (see Theorem 3.4).

In the following definition let $x = s\langle i\rangle$ be a valid nonfinal location. Note in particular that $s_i \in \Sigma$. For example, if s = "abc" then such locations in s are $s\langle 0\rangle$, $s\langle 1\rangle$, and $s\langle 2\rangle$.

$$\begin{array}{rcl} \mathit{Der}_x(R) & \overset{\text{DEF}}{=} & \bot\ \textbf{if}\ R \in \text{⚓}\ \textbf{or}\ R = () \\ \mathit{Der}_{s\langle i\rangle}(\psi) & \overset{\text{DEF}}{=} & \textbf{if}\ s_i \in [\![\psi]\!]\ \textbf{then}\ ()\ \textbf{else}\ \bot \\ \mathit{Der}_x(R|S) & \overset{\text{DEF}}{=} & \mathit{Der}_x(R) \mid \mathit{Der}_x(S) \\ \mathit{Der}_x(R\cdot S) & \overset{\text{DEF}}{=} & \textbf{if}\ \mathit{Null}_x(R)\ \textbf{then}\ \mathit{Der}_x(R)\cdot S \,|\, \mathit{Der}_x(S)\ \textbf{else}\ \mathit{Der}_x(R)\cdot S \\ \mathit{Der}_x(R\{m,n,l\}) & \overset{\text{DEF}}{=} & \begin{cases} \mathit{Der}_x(R)\cdot R\{m\dot{-}1, n\dot{-}1, l\},\ \textbf{if}\ m{=}0\ \textbf{or}\ \mathit{Null}_\forall(R){=}\textbf{true}\ \textbf{or}\ \mathit{Null}_x(R){=}\textbf{false}; \\ \mathit{Der}_x(R\cdot R\{m\dot{-}1, n\dot{-}1, l\}),\ \textbf{otherwise}. \end{cases} \end{array}$$

where $\infty\dot{-}1 \overset{\text{DEF}}{=} \infty$, $0\dot{-}1 \overset{\text{DEF}}{=} 0$, $k\dot{-}1 \overset{\text{DEF}}{=} k-1$ for $k>0$. Note the special case for standard infinite loops: $\mathit{Der}_x(R*) = \mathit{Der}_x(R)\cdot R*$. Recall also that $R\{0,0,\_\} \overset{\text{DEF}}{=} ()$ and $R\cdot() = R$, and so $\mathit{Der}_x(R\{0,1,\_\}) = \mathit{Der}_x(R\{1,1,\_\}) = \mathit{Der}_x(R)$. We let also $R\{m,n,l\} - 1 \overset{\text{DEF}}{=} R\{m\dot{-}1, n\dot{-}1, l\}$.

Consider R = `.*b` and s = "abba". Then $\mathit{Der}_{s\langle 1\rangle}(R) = \mathit{Der}_{s\langle 1\rangle}(\texttt{.*})\texttt{b} \,|\, \mathit{Der}_{s\langle 1\rangle}(\texttt{b}) = \texttt{R}\,|\,()$ by using the rules for concatenations, predicates (where $s_1 \in [\![\psi_{\texttt{b}}]\!] = \{\texttt{b}\}$ and $s_1 \in [\![\psi_{\texttt{.}}]\!]$), and loops.

We are now ready to define what it means to find a match *end* location from a valid *start* location x in a string s by a regex $R \in \mathit{RE}$. $\mathit{MatchEnd}(x, R)$ returns the *latest match end location* from a valid x or $\lightning$ if none exists. Note that $\max(x, \lightning) = \max(\lightning, x) = x$.

$$\begin{array}{rcl} \mathit{Null}^{\lightning}_x(R) & \overset{\text{DEF}}{=} & \textbf{if}\ \mathit{Null}_x(R)\ \textbf{then}\ x\ \textbf{else}\ \lightning \\ \mathit{MatchEnd}(x,R) & \overset{\text{DEF}}{=} & \textbf{if}\ \mathit{Final}(x)\ \textbf{then}\ \mathit{Null}^{\lightning}_x(R)\ \textbf{else}\ \max(\mathit{Null}^{\lightning}_x(R), \mathit{MatchEnd}(x{+}1, \mathit{Der}_x(R))) \\ \mathit{IsMatch}(x,R) & \overset{\text{DEF}}{=} & \mathit{MatchEnd}(x,R) \neq \lightning \end{array}$$

The definition of $\mathit{MatchEnd}(x,R)$ with $x = s\langle i\rangle$ computes the transition from the source state (regex) R to the target state $S = \mathit{Der}_x(R)$ for the character s_i and then continues matching from location $x+1$ and state S. The *existence* of a match, i.e., $\mathit{IsMatch}(x,R)$, is *independent* of backtracking semantics. The additional notions required for $\mathit{MatchEnd}(x,R)$ to respect backtracking semantics are discussed in Section 4 where *pruning* of regexes is introduced that primarily affects the definition of $\mathit{Der}_x(R\cdot S)$, while the top-level definition of $\mathit{MatchEnd}(x,R)$ as stated above remains unchanged.

In the case $\mathit{IsMatch}(x,R)=\mathbf{true}$, x need not be the final location in s. If x is nonfinal the recursive call continues from the next location with the x-derivative. The definition accurately reflects the semantics of the actual implementation.[7] We illustrate *IsMatch* in Example 3.2.

Example 3.2. Consider the regex `\A.+$` that matches the first line of the input, and that line must be nonempty, where dot denotes $\Sigma\setminus\{\texttt{\n}\}$. Let $s=$ `"I\nAm"`. Then $\mathit{Null}_{s\langle 0\rangle}(\texttt{\A})=\mathbf{true}$. So

$$\mathit{IsMatch}(s\langle 0\rangle,\texttt{\A.+\$})\Leftrightarrow \mathit{IsMatch}(s\langle 1\rangle,\mathit{Der}_{s\langle 0\rangle}(\texttt{\A.+\$}))$$
$$\Leftrightarrow \mathit{IsMatch}(s\langle 1\rangle,\mathit{Der}_{s\langle 0\rangle}(\texttt{.+\$})\,|\,\mathit{Der}_{s\langle 0\rangle}(\texttt{\A}).+\$)$$
$$\Leftrightarrow \mathit{IsMatch}(s\langle 1\rangle,\mathit{Der}_{s\langle 0\rangle}(\texttt{.+\$}))\Leftrightarrow \mathit{IsMatch}(s\langle 1\rangle,\mathit{Der}_{s\langle 0\rangle}(\texttt{.})\texttt{.*\$})\Leftrightarrow \mathit{IsMatch}(s\langle 1\rangle,\texttt{.*\$})$$

where in the last step $\mathit{Der}_{s\langle 0\rangle}(\texttt{.})=\texttt{()}$ because $s_0\neq\texttt{\n}$. Since $\mathit{Null}_{s\langle 1\rangle}(\texttt{.*\$})=\mathbf{true}$ it follows that $\mathit{IsMatch}(s\langle 1\rangle,\texttt{.*\$})=\mathbf{true}$. Note also that $\bot\cdot R\longrightarrow\bot$, $R|\bot\longrightarrow R$ and $\texttt{()}\cdot R\longrightarrow R$ are always applied as immediate simplifications (rewrites) when regexes are constructed. ⊠

3.4 Properties of Derivatives

Here we introduce some fundamental properties of derivatives that are later needed in proving correctness theorems of matching. The key concept is the *derivation relation* $x\xrightarrow{R}y$ between locations, also denoted by $\langle x,y\rangle\models R$, that is instrumental in reasoning about properties. Let the *universe of all matches* be $\mathcal{U}\overset{\text{DEF}}{=}\{\langle s\langle i\rangle,s\langle j\rangle\rangle\mid s\in\Sigma^*,0\le i\le j\le|s|\}$. Then

$$\begin{array}{rcl}
\mathit{Null}^{\emptyset}_x(R) & \overset{\text{DEF}}{=} & \mathbf{if}\ \mathit{Null}_x(R)\ \mathbf{then}\ \{x\}\ \mathbf{else}\ \emptyset\\
\mathit{AllMatchEnds}(x,R) & \overset{\text{DEF}}{=} & \mathit{Null}^{\emptyset}_x(R)\cup\mathbf{if}\ \mathit{Final}(x)\ \mathbf{then}\ \emptyset\ \mathbf{else}\ \mathit{AllMatchEnds}(x{+}1,\mathit{Der}_x(R))\\
\langle x,y\rangle\models R\ \overset{\text{DEF}}{=}\ x\xrightarrow{R}y & \overset{\text{DEF}}{=} & y\in\mathit{AllMatchEnds}(x,R)\\
\mathcal{M}(R) & \overset{\text{DEF}}{=} & \{M\in\mathcal{U}\mid M\models R\}
\end{array}$$

We say that $M\in\mathcal{U}$ is a *match* of R if $M\models R$, and $\mathcal{M}(R)$ is called the *match language* of R. For example, $\mathcal{M}(\bot)=\emptyset$ and $\mathcal{M}(\top\star)=\mathcal{U}$. Two regexes R and S are *equivalent*, denoted by $R\equiv S$, if $\mathcal{M}(R)=\mathcal{M}(S)$. Note that $\mathit{MatchEnd}(x,R)=\max(\mathit{AllMatchEnds}(x,R))$ follows directly.

Observe the invariant that if $x\xrightarrow{R}y$ then $\pi_1(x)=\pi_1(y)$, and $\pi_2(x)\le\pi_2(y)$, i.e., the string of the locations remains fixed and only the *distance* $\pi_2(y)-\pi_2(x)$ between them can increase. Note also that if $x=s\langle i\rangle$ is valid and nonfinal then $x+1\overset{\text{DEF}}{=}s\langle i+1\rangle$ is valid.

The following is the main derivation theorem. The proofs of (3) and (4) are by induction over $\pi_2(y)-\pi_2(x)$, (5) uses (3,4), and (6) uses (3–5).

THEOREM 3.3 (DERIVATION). *For all regexes and valid locations, let $A\in⚓\cup\{\texttt{()}\}$ and $\psi\in\Psi$:*

(1) $x\xrightarrow{A}y\Leftrightarrow\mathit{Null}_x(A)$ **and** $x=y$;
(2) $s\langle i\rangle\xrightarrow{\psi}y\Leftrightarrow s_i\in[\![\psi]\!]$ **and** $y=s\langle i+1\rangle$;
(3) $x\xrightarrow{R|S}y\Leftrightarrow(x\xrightarrow{R}y$ **or** $x\xrightarrow{S}y)$;
(4) $x\xrightarrow{R\cdot S}y\Leftrightarrow\exists z(x\xrightarrow{R}z\xrightarrow{S}y)$;
(5) $\forall m>0:R\{m,n\}\equiv R\cdot R\{m-1,n-1\}\equiv R\{m-1,n-1\}\cdot R$;
(6) $R\{0,n\}\equiv R\{1,n\}|\texttt{()}$;

A notable special case is $\bot\star\equiv\texttt{()}$ because $\bot\star\equiv\bot\cdot\bot\star|\texttt{()}\equiv\texttt{()}$ where $\nexists z(x\xrightarrow{\bot}z)$ since $[\![\bot]\!]=\emptyset$.

3.5 Relation to Classical Regular Expressions and Derivatives

Recall the classical definition of the language $\mathcal{L}(R)\subseteq\Sigma^*$ of R *without anchors*: $\mathcal{L}(\texttt{()})=\{\epsilon\}$, $\mathcal{L}(\psi)=[\![\psi]\!]$, $\mathcal{L}(L\cdot R)=\mathcal{L}(L)\cdot\mathcal{L}(R)$, $\mathcal{L}(L|R)=\mathcal{L}(L)\cup\mathcal{L}(R)$, and $\mathcal{L}(R\star)=\mathcal{L}(R)^*$ that has the generalization $\mathcal{L}(R\{m\})=\mathcal{L}(R)\cdot\mathcal{L}(R\{m-1\})$ to finite loops.

[7] Many important optimizations are omitted here, such as $\mathit{MatchEnd}(\_,\bot)\overset{\text{DEF}}{=}\lightning$, $\mathit{MatchEnd}(s\langle\_\rangle,\top\star)\overset{\text{DEF}}{=}s\langle|s|\rangle$, $\mathit{MatchEnd}(x,\texttt{()})\overset{\text{DEF}}{=}x$, and $\mathit{MatchEnd}(x,A)\overset{\text{DEF}}{=}\mathit{Null}^{\lightning}_x(A)$ for $A\in⚓$.

Theorem 3.4. *If $R \in RE$ contains no anchors then $s \in \mathcal{L}(R) \Leftrightarrow \langle s\langle 0\rangle, s\langle |s|\rangle\rangle \in \mathcal{M}(R)$.*

Proof. If R is classical then $Der_{s\langle i\rangle}(R) = D_{s_i}(R)$ where $D_a(R)$ is essentially the *Brzozowski* derivative of R for $a \in \Sigma$. The statement then follows from [Brzozowski 1964] (provided that each predicate ψ in R is viewed equivalently as an alternation over all characters in $[\![\psi]\!]$). □

We are more focused here on the derivation relation rather than languages. Note that most classical properties fail for languages of $R \in RE$ when anchors are present. For example, if $R = \mathtt{a\backslash b}$ then $\mathcal{M}(R) = \{\langle s\langle i\rangle, s\langle i{+}1\rangle\rangle \mid s_i = \mathtt{a}, s_{i+1} \notin [\![\psi_{\backslash w}]\!]\}$ but $\mathcal{M}(R{\cdot}R) = \emptyset$ because `\b` is infeasible between two word-letters. While $\mathcal{M}(R)$ is *regular* for all $R \in RE$, because the derivation relation induces a finite set of derivatives (see Section 5), even for *IsMatch* it would be very complicated and for *MatchEnd not possible*, to convert $R \in RE$ into an equivalent regex without anchors.

Note on Loops. There is another *crucial* difference to the classical case of derivatives of loops. Let $L = R\{m, n\}$. When R contains no anchors, it holds that $Der_x(L) = Der_x(R){\cdot}(L - 1)$ because then either $Null_\forall(R) = \mathbf{true}$ or else $Null_\nexists(R) = \mathbf{true}$ – there is no "middle ground". In contrast, it would be *incorrect* to define $Der_x(L)$ as $Der_x(R){\cdot}(L - 1)$ when $Null_x(R) = \mathbf{true}$ and nullability depends on anchors, as Example 3.5 illustrates, while definition of $Der_x(L)$ as $Der_x(R{\cdot}(L-1))$ when $Null_x(R) = \mathbf{true}$ and both $m = 0$ and $n = \infty$, would be circular through the rule for concatenation and thus *not well-defined. So the two cases of loop derivatives are crucial for correctness.*

Example 3.5. Let $R = (\mathtt{a|\backslash b})$ and $L = R\{2\}$ and $x = \texttt{"abc"}\langle 0\rangle$. It is clear that L is meant to be equivalent to $R{\cdot}R$. Note that $Der_x(R){\cdot}R = R$, while $Der_x(R{\cdot}R) = Der_x(R){\cdot}R\,|\,Der_x(R) = R\,|\,()$ because $Null_x(R) = \mathbf{true}$. Therefore $Null_{x+1}(Der_x(R){\cdot}R) = Null_{x+1}(R) = \mathbf{false}$ because `\b` is not nullable in $x + 1$ while $Null_{x+1}(Der_x(R{\cdot}R)) = Null_{x+1}(R\,|\,()) = \mathbf{true}$ due to $()$. So $Der_x(R){\cdot}R \not\equiv Der_x(R{\cdot}R)$. ⊠

3.6 Reversal

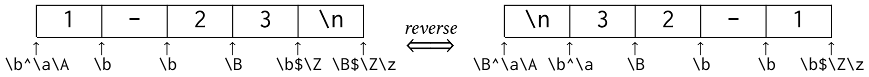
Reversal of regexes is used in the complete matching algorithm in order to locate the *beginning* of a match, where the original search pattern is used *backwards* from a previously found *ending* location. Details of the complete matching procedure are in Section 4.5. Here our focus is on reversal itself. The *reverse* R^r of $R \in RE$ is defined as follows.

$$\mathtt{\backslash A}^r \stackrel{\text{DEF}}{=} \mathtt{\backslash z} \quad \mathtt{\hat{}}^r \stackrel{\text{DEF}}{=} \mathtt{\$} \quad \mathtt{\backslash a}^r \stackrel{\text{DEF}}{=} \mathtt{\backslash Z} \quad \mathtt{\backslash b}^r \stackrel{\text{DEF}}{=} \mathtt{\backslash b} \quad \mathtt{\backslash z}^r \stackrel{\text{DEF}}{=} \mathtt{\backslash A} \quad \mathtt{\$}^r \stackrel{\text{DEF}}{=} \mathtt{\hat{}} \quad \mathtt{\backslash Z}^r \stackrel{\text{DEF}}{=} \mathtt{\backslash a} \quad \mathtt{\backslash B}^r \stackrel{\text{DEF}}{=} \mathtt{\backslash B}$$
$$\psi^r \stackrel{\text{DEF}}{=} \psi \quad ()^r \stackrel{\text{DEF}}{=} () \quad (R|S)^r \stackrel{\text{DEF}}{=} R^r|S^r \quad (R{\cdot}S)^r \stackrel{\text{DEF}}{=} S^r{\cdot}R^r \quad R\{m,n,b\}^r \stackrel{\text{DEF}}{=} R^r\{m,n,b\}$$

Lemma 3.6 is proved by induction over *RE*. Example 3.7 illustrates an instance of Lemma 3.6. Theorem 3.8 is proved by induction over location distances and uses Theorem 3.3 and Lemma 3.6.

Lemma 3.6. *For all $R \in RE$ and valid locations x: $Null_x(R) \Leftrightarrow Null_{x^r}(R^r)$.*

Example 3.7. Nullable anchor locations in the string "1-23\n" and its reverse:



⊠

Theorem 3.8 (Reversal). *For all $R \in RE$ and valid locations x and y: $x \xrightarrow{R} y \Leftrightarrow y^r \xrightarrow{R^r} x^r$.*

3.7 Lookarounds

Lookarounds are expressions in the form of *lookaheads* $(\texttt{?=}R)$ and $(\texttt{?!}R)$, and *lookbacks* $(\texttt{?<=}R)$ and $(\texttt{?<!}R)$, where R is a regex. Lookarounds are currently not implemented in the nonbacktracking engine but can *very transparently* be supported as follows. Let x be a valid location.

$$Null_x((\texttt{?=}R)) \stackrel{\text{DEF}}{=} IsMatch(x, R) \qquad Null_x((\texttt{?!}R)) \stackrel{\text{DEF}}{=} \mathbf{not}\, IsMatch(x, R)$$
$$Null_x((\texttt{?<=}R)) \stackrel{\text{DEF}}{=} IsMatch(x^r, R^r) \qquad Null_x((\texttt{?<!}R)) \stackrel{\text{DEF}}{=} \mathbf{not}\, IsMatch(x^r, R^r)$$
$$(\texttt{?=}R)^r \stackrel{\text{DEF}}{=} (\texttt{?<=}R^r) \quad (\texttt{?<=}R)^r \stackrel{\text{DEF}}{=} (\texttt{?=}R^r) \quad (\texttt{?!}R)^r \stackrel{\text{DEF}}{=} (\texttt{?<!}R^r) \quad (\texttt{?<!}R)^r \stackrel{\text{DEF}}{=} (\texttt{?!}R^r).$$

If ℓ is a lookaround and x a valid nonfinal location then $Der_x(\ell) \overset{\text{DEF}}{=} \bot$. In other words, *lookarounds are like anchors* but use the *full context* of x.[8] Observe that lookarounds are a *proper* generalization of anchors, we could have started with *RE* including lookarounds and *omitted* anchors, in which case all of the anchors can be defined in terms of lookarounds. E.g., observe that `\A` $\equiv$ `(?<!`$\top$`)`:

$$\begin{aligned} Null_x(\texttt{(?<!}\top\texttt{)}) &\Leftrightarrow \textbf{not}\, IsMatch(x^r, \top) \Leftrightarrow (Final(x^r)\ \textbf{or not}\, IsMatch(x^r{+}1, Der_{x^r}(\top))) \\ &\Leftrightarrow (Final(x^r)\ \textbf{or not}\, IsMatch(x^r{+}1, \texttt{()})) \Leftrightarrow (Final(x^r)\ \textbf{or not true}) \Leftrightarrow Initial(x) \end{aligned}$$

Where x^r is final iff x is initial. Similarly, `\z` $\equiv$ `(?!`$\top$`)`. The word-border anchor `\b` has also an elegant equivalent definition by using lookarounds `\b` $\equiv$ `(?<=`$\psi_{\backslash w}$`)`$\cdot$`(?!`$\psi_{\backslash w}$`)|(?<!`$\psi_{\backslash w}$`)`$\cdot$`(?=`$\psi_{\backslash w}$`)`. Similarly for `\B`. All the other anchors can be defined similarly, even using nested lookarounds. For example `(?=\n|\z)` $\equiv$ `$` and `(?<=\A|\A\n)` $\equiv$ `\a`. The downside of allowing unrestricted use of lookarounds would be that *input-linear* complexity of matching (Theorem 5.3) would not hold because nullability tests would no longer be independent of the length of the input.

4 MATCHING WITH BACKTRACKING SEMANTICS WITHOUT BACKTRACKING

Here we introduce the top-level matching algorithm that preserves backtracking semantics. First we formally describe what we mean by backtracking based regex matching. We then introduce two key techniques: *high-nullability* and *pruning* that are fundamental in testing and restricting derivatives in such a way that backtracking semantics is preserved. We prove that the *tail-recursive* procedure *MatchEnd* works dually by *simulating backtracking*. Finally, we present the top-level algorithm *Match* that uses *MatchEnd* in two phases: a forward phase that simulates backtracking by pruning of derivatives to find an *end* location for a match, and a backwards phase that reverses the given search pattern to find the *start* location backwards from the previously found end location in nonbacktracking mode. All the theory presented in Section 3 is utilized here fully and formal correctness theorems are stated and proved for all the key statements.

4.1 Backtracking Based Regex Match End Search Semantics

We describe an abstract recursive backtracking based *match end* search procedure semantics $\lfloor R \rfloor_x$ for a regex R from a *start* location x in a string s by a recursive search through $\lceil R \rceil_x$ that returns the list of all end locations in order of backtracking priority, where $\oplus$ appends lists. Let $first([]) \overset{\text{DEF}}{=} ⚡$ and let $first(\ell)$ denote the first element of a nonempty list ℓ. In the following let $A \in ⚓$, $\psi \in \Psi$, $s \in \Sigma^*$, $x = s\langle i \rangle$ be a nonfinal location in s (where $0 \le i < |s|$), and $y = s\langle |s| \rangle$ be the final location in s.

$$\begin{array}{rcl@{\qquad}rcl} \lfloor R \rfloor_x & \overset{\text{DEF}}{=} & first(\lceil R \rceil_x) & \lceil A{\cdot}Z \rceil_x & \overset{\text{DEF}}{=} & \textbf{if } Null_x(A) \textbf{ then } \lceil Z \rceil_x \textbf{ else } [] \\ \lceil \texttt{()} \rceil_x & \overset{\text{DEF}}{=} & [x] & \lceil \psi{\cdot}Z \rceil_x & \overset{\text{DEF}}{=} & \textbf{if } s_i \in [\![\psi]\!] \textbf{ then } \lceil Z \rceil_{x+1} \textbf{ else } [] \\ \lceil (X\,|\,Y){\cdot}Z \rceil_x & \overset{\text{DEF}}{=} & \lceil X{\cdot}Z \rceil_x \oplus \lceil Y{\cdot}Z \rceil_x & \lceil X\{m\}{\cdot}Z \rceil_x & \overset{\text{DEF}}{=} & \lceil X{\cdot}X\{m{-}1\}{\cdot}Z \rceil_x \\ & & & \lceil R \rceil_y & \overset{\text{DEF}}{=} & \textbf{if } Null_y(R) \textbf{ then } [y] \textbf{ else } [] \end{array}$$

All concatenations are in right associative form and where Z may be `()` for the case when the regex is not a concatenation. We focus on *finite* loops only by interpreting ∞ as $|s|{+}1$ here. This assumption preserves the semantics but avoids infinite recursion of $\lceil R{*} \rceil_x$. We can therefore use the normal form of loops with a single counter (recall Section 3.1).[9] Below we extend the definition of derivatives accordingly to capture the semantics of backtracking. Intuitively, derivatives store the backtracking choice points in the regex itself by utilizing the *order* of alternatives where priority is always given to the *first* alternative of an alternation.

[8]Observe that, even if R itself would contain lookarounds then all the involved definitions would still remain mutually well-defined (by induction) because R is a smaller expression than the lookaround containing it.

[9]We are not concerned here with how backtracking is *implemented*, where highly optimized data structures are being used.

4.2 Prioritized Nullability

The central case is dealing with a concatenation $L{\cdot}R$ when L is nullable, that typically results in an alternation being created. In this case, if L is *high-priority nullable* or *high-nullable* then skipping L and continuing to match R takes priority. Formally, for $R \in RE$ and valid locations x:

$$\begin{aligned}
Null!_x(R) &\overset{\text{DEF}}{=} Null_x(R) \textbf{ if } R \in ⚓ \textbf{ or } R \in \Psi \textbf{ or } R = \texttt{()} \\
Null!_x(L \mid R) &\overset{\text{DEF}}{=} Null!_x(L) \\
Null!_x(L \cdot R) &\overset{\text{DEF}}{=} Null!_x(L) \textbf{ and } Null!_x(R) \\
Null!_x(R\{m, n, lazy\}) &\overset{\text{DEF}}{=} (lazy \textbf{ and } m = 0) \textbf{ or } Null!_x(R)
\end{aligned}$$

The intuition behind high-nullability of loops is that in a lazy loop the intent is to exit the loop as early as possible, while in an eager loop the intent is to exit the loop as late as possible. A lazy loop $R\{0,n\}?$ is equivalent to $\texttt{()}\,|\,R{\cdot}R\{0,n{\dot-}1\}?$ while an eager loop $R\{0,n\}$ is equivalent to $R{\cdot}R\{0,n{\dot-}1\}\,|\,\texttt{()}$. The rule for loops now follows from that of alternation and concatenation.[10]

Example 4.1. The regex `(abc|())` is nullable but not high-nullable because `()` comes second, while `(()|abc)` is nullable as well as high-nullable because `()` comes first. The regex `$|.*` is nullable but not high-nullable in `"a\na"`$\langle 0\rangle$ but it is high-nullable in `"a\na"`$\langle 1\rangle$. ⊠

The computation of derivatives is now updated as follows, by taking high-nullability into account.

$$Der_x(L{\cdot}R) \overset{\text{DEF}}{=} \begin{cases} Der_x(L){\cdot}R, & \textbf{if not } Null_x(L); \\ Der_x(R) \mid Der_x(L){\cdot}R, & \textbf{else if } Null!_x(L); \\ Der_x(L){\cdot}R \mid Der_x(R), & \textbf{otherwise}; \end{cases}$$

For example, $Der_{\texttt{"abba"}\langle 1\rangle}(\texttt{.*b}) = \texttt{.*b|()}$ but $Der_{\texttt{"abba"}\langle 1\rangle}(\texttt{.*?b}) = \texttt{()|.*?b}$.

4.3 Pruning

Pruning of a regex R in a valid location x, denoted by $Prune_x(R)$, removes those branches of R that are not used in *backtracking* in order to preserve backtracking semantics of the resulting derivatives. Intuitively, pruning mimics how backtracking chooses a path.

$$Prune_x(R) \overset{\text{DEF}}{=} \begin{cases} R & \text{, } \textbf{if } Null_x(R) = \textbf{false}; \\ \texttt{()} & \text{, } \textbf{else if } Null!_x(R) = \textbf{true}; \\ Prune_x(Z), & \textbf{else if } R = X{\cdot}Z \textbf{ and } Null!_x(X) = \textbf{true}; \end{cases}$$

otherwise $Null_x(R) = \textbf{true}$ and $Null!_x(R) = \textbf{false}$ and if $R = X{\cdot}Z$ then $Null!_x(X) = \textbf{false}$ – observe that if X is a loop with body B then $Null!_x(B) = \textbf{false}$. We proceed by case analysis over R. We focus on the key case of the normal form of loops using a single counter. Observe that the case $R = X{*}?{\cdot}Z$ is not possible below because $X{*}?$ is high-nullable.

$$\begin{aligned}
Prune_x((X{\cdot}Y){\cdot}Z) &\overset{\text{DEF}}{=} Prune_x(X{\cdot}(Y{\cdot}Z)) \\
Prune_x(X\,|\,Y) &\overset{\text{DEF}}{=} \textbf{if } Null_x(X) \textbf{ then } Prune_x(X) \textbf{ else } X \mid Prune_x(Y) \\
Prune_x((X\,|\,Y){\cdot}Z) &\overset{\text{DEF}}{=} Prune_x(X{\cdot}Z \mid Y{\cdot}Z) \\
Prune_x(X\{m\}{\cdot}Z) &\overset{\text{DEF}}{=} Prune_x(X{\cdot}X\{m{-}1\}{\cdot}Z) \\
Prune_x(X{*}{\cdot}Z) &\overset{\text{DEF}}{=} Prune_x(X){\cdot}X{*}{\cdot}Z \mid Prune_x(Z)
\end{aligned}$$

The main case is $Prune_x(X\,|\,Y)$ that prioritizes X if a solution exists (Y is forgotten). Concatenation $(X\,|\,Y)Z$ needs a special case of being treated as $XZ\,|\,YZ$ in order to preserve XZ as the first alternative in case XZ is not nullable and YZ ends up being pruned, essentially the alternation must be propagated to the top level prior to pruning. Certain optimizations (shortcuts) are also

[10]Typically the body of a loop is not nullable, in which case the loop is high-nullable iff its lower bound is 0 and it is lazy.

omitted for the case of $\mathit{Null}_\forall(R) = \textbf{true}$ in which case the counter need not be unfolded or when the loop body of an eager loop is never nullable. For example, `a{0,5}?c*` is pruned to `cc*|()` that is kept as `c*` because `c` is never nullable. A related optimization is that $\mathit{Prune}_x(X\{0,n\}) \stackrel{\text{DEF}}{=} X\{0,n\}$ if X is never nullable.

As Example 4.1 illustrated, R may be nullable in one location while not high-nullable in that location, but high-nullable in another location. So pruning is in general location dependent because of *anchors*. Pruning preserves backtracking semantics. It cuts off *all* alternatives that are not going to be used in backtracking. We will use the following lemma that relates backtracking to derivatives, and is proved by induction over R, using the definitions of pruning and high-nullability.

Lemma 4.2. *For all valid locations x and $R \in RE$, let $\lfloor R \rfloor_{x+1} \stackrel{\text{DEF}}{=} \lightning$ if x is final,*

(1) $\textbf{not}\, \mathit{Null}_x(R) \Rightarrow (\lfloor R\rfloor_x = \lfloor \mathit{Der}_x(R)\rfloor_{x+1}$ **and** $(\lfloor R\rfloor_x = \lightning \Rightarrow \mathit{AllMatchEnds}(x,R) = \emptyset))$.
(2) $\mathit{Null}_x(R) \Rightarrow \lfloor R\rfloor_x = \max(x, \lfloor \mathit{Der}_x(\mathit{Prune}_x(R))\rfloor_{x+1}) \in \mathit{AllMatchEnds}(x,R)$.

Lemma 4.2(1) states that search may equivalently continue using the derivative from the next location when the current one is not a solution. Lemma 4.2(2) states that pruning eliminates *exactly* those alternatives of R that correspond to the choices that backtracking would never make, and the current location is the solution unless a later one exists. Lemma 4.2 paves a way to implement backtracking in a *tail-recursive* manner by using derivatives, as formalized next.

4.4 Finding Match End Locations with Backtracking Semantics

We extend the abstract syntax of regexes with an internal top-level marker indicating that, during derivation, the regex is to be interpreted in a mode that *simulates backtracking*: $RE_{\text{BT}} ::= RE \mid \mathtt{BT}(RE)$. Derivatives and nullability are extended to RE_{BT} where x is valid. We let $\mathtt{BT}(\bot) \stackrel{\text{DEF}}{=} \bot$. The marker is retained in derivatives in order to maintain the backtracking simulation mode, where R is now *pruned* before taking its derivative. (Let x be nonfinal in $\mathit{Der}_x(R)$.)

$$\mathit{Der}_x(\mathtt{BT}(R)) \stackrel{\text{DEF}}{=} \mathtt{BT}(\mathit{Der}_x(\mathit{Prune}_x(R))) \qquad \mathit{Null}_x(\mathtt{BT}(R)) \stackrel{\text{DEF}}{=} \mathit{Null}_x(R)$$

Observe that *MatchEnd* operates in *backtracking simulation mode* if the marker is present else in *nonbacktracking mode* where regexes are not pruned. Note that catastrophic backtracking is not possible here because pruning *simulates* backtracking semantics *without actual backtracking*. Lemma 4.3 establishes key properties used later, where 4.3(3) is proved by induction over locations distances using Lemma 4.2.

Lemma 4.3. *For $R \in RE$ and valid locations x and y:*

(1) $\mathit{MatchEnd}(x, \mathtt{BT}(R)) = y \neq \lightning \Rightarrow x \xrightarrow{R} y$;
(2) $\mathit{AllMatchEnds}(x,R) = \emptyset \Rightarrow \mathit{MatchEnd}(x, \mathtt{BT}(R)) = \lightning$;
(3) $\mathit{MatchEnd}(x, \mathtt{BT}(R)) = \lfloor R\rfloor_x$.

Example 4.4. Consider the regex `BT(.*?b)` that finds the location immediately after the *first occurrence* of `b` in an input. Let s = `"abba"`. We show main steps. Let **F** stand for *MatchEnd*.

$$\begin{aligned}&\mathbf{F}(s\langle 0\rangle, \mathtt{BT(.*?b)}) = \mathbf{F}(s\langle 1\rangle, \mathtt{BT(.*?b)}) = \mathbf{F}(s\langle 2\rangle, \mathtt{BT}(\mathit{Der}_{s\langle 1\rangle}(\mathtt{.*?b})))\\ &= \mathbf{F}(s\langle 2\rangle, \mathtt{BT(()|.*?b)}) = \max(s\langle 2\rangle, \mathbf{F}(s\langle 3\rangle, \mathtt{BT}(\mathit{Der}_{s\langle 2\rangle}(\mathit{Prune}(\mathtt{()|.*?b})))))\\ &= \max(s\langle 2\rangle, \mathbf{F}(s\langle 3\rangle, \mathtt{BT}(\mathit{Der}_{s\langle 2\rangle}(\mathtt{()})))) = \max(s\langle 2\rangle, \mathbf{F}(s\langle 3\rangle, \bot)) = \max(s\langle 2\rangle, \lightning) = s\langle 2\rangle\end{aligned}$$

The $s\langle 1\rangle$-derivative of the concatenation `.*?b` prioritizes skipping the lazy loop as opposed to staying in the loop. So pruning eliminates the second alternative of `()|.*?b`. Now consider the regex `BT(.*b)` that finds the location after the *last occurrence* of `b` in s. We get that

$$\begin{aligned}&\mathbf{F}(s, \mathtt{BT(.*b)}) = \mathbf{F}(s\langle 2\rangle, \mathtt{BT(.*b|())}) = \max(s\langle 2\rangle, \mathbf{F}(s\langle 3\rangle, \mathtt{BT(.*b|())}))\\ &= \max(s\langle 2\rangle, \max(s\langle 3\rangle, \mathbf{F}(s\langle 4\rangle, \mathtt{BT(.*b)}))) = \max(s\langle 2\rangle, \max(s\langle 3\rangle, \lightning)) = s\langle 3\rangle\end{aligned}$$

The derivative of the concatenation `.*b` prioritizes staying in the eager loop. Consequently, pruning keeps both alternatives of `.*b|()`. Note that $s\langle 4\rangle$ is final and `.*b` is not nullable. ⊠

4.5 Complete Matching

We are now ready to present the complete matching algorithm that given a string s and a regex $R \in RE$ finds the *earliest start* location x in s and the *end location* y in s such that $x \xrightarrow{R} y$ and *y is the backtracking end location.* A successful search result is the pair $\langle x, y\rangle$ and $\lightning$ represents failed search.

$$Match(s,R) \overset{\text{DEF}}{=} \textbf{if } y = MatchEnd(s\langle 0\rangle, \mathtt{BT}(\top{*}?{\cdot}R)) \neq \lightning \textbf{ then } \langle MatchEnd(y^r, R^r)^r, y\rangle \textbf{ else } \lightning$$

In the correctness proof we use symmetry of reversal that implies, by Theorem 3.8, that for any valid location y in a string s, $\min\{z \mid z \xrightarrow{R} y\} = (\max\{z \mid y^r \xrightarrow{R^r} z\})^r$. In other words, z is the *earliest* location in s such that $z \xrightarrow{R} y$ iff z^r is the *latest* location in s^r such that $y^r \xrightarrow{R^r} z^r$.

Theorem 4.5 (Correctness of *Match*). *For all $s \in \Sigma^*$ and $R \in RE$:*

(1) $Match(s,R) = \lightning \Rightarrow \nexists\, i\, j(s\langle i\rangle \xrightarrow{R} s\langle j\rangle)$.
(2) $Match(s,R) = \langle s\langle i\rangle, s\langle j\rangle\rangle \models R$ *such that*
  (a) $i = \min\{i \mid \exists z(s\langle i\rangle \xrightarrow{R} z)\}$
  (b) $s\langle j\rangle = \lfloor \top{*}?{\cdot}R\rfloor_{s\langle 0\rangle}$

Proof. Let $y = MatchEnd(s\langle 0\rangle, \mathtt{BT}(\top{*}?{\cdot}R))$.

No match exists: Assume $y = \lightning$. By using Lemma 4.3(2) it follows that $\nexists w(s\langle 0\rangle \xrightarrow{\top{*}?{\cdot}R} w)$. By Theorem 3.3(4), we get that $\nexists w(\exists z(s\langle 0\rangle \xrightarrow{\top{*}?} z \xrightarrow{R} w))$. But at the same time $s\langle 0\rangle \xrightarrow{\top{*}?} z$ for *all* valid locations z in s. It follows that there exist no locations x and y in s such that $x \xrightarrow{R} y$.

Match exists: Assume $y \neq \lightning$. Let $\mathbf{F}$ stand for *MatchEnd.* Then, by Lemma 4.3(1), $s\langle 0\rangle \xrightarrow{\top{*}?{\cdot}R} y$ and by Theorem 3.3(4) there exists z s.t. $s\langle 0\rangle \xrightarrow{\top{*}?} z \xrightarrow{R} y$. Let $z_{\min} = \min\{z \mid z \xrightarrow{R} y\}$. So $\mathbf{F}(y^r, R^r) = \max\{z \mid y^r \xrightarrow{R^r} z\}$ and by Theorem 3.8, $z_{\min} = \mathbf{F}(y^r, R^r)^r$. So 4.5(1) and 4.5(2a) follow, and 4.5(2b) follows from Lemma 4.3(3). □

Example 4.6. Let $R =$ `he|the|cat` and $s =$ `"I see the cat"`. Let $\mathbf{F} = MatchEnd$. Then[11]
$\mathbf{F}(s\langle 0\rangle, \mathtt{BT}(\top{*}?{\cdot}R)) = \mathbf{F}(s\langle 6\rangle, \mathtt{BT}(\top{*}?{\cdot}R)) = \mathbf{F}(s\langle 7\rangle, \mathtt{BT}(\mathtt{he}|\top{*}?{\cdot}R))$
$= \mathbf{F}(s\langle 8\rangle, \mathtt{BT}(\mathtt{e}|\mathtt{e}|\top{*}?{\cdot}R)) = \mathbf{F}(s\langle 9\rangle, \mathtt{BT}(()|()|\top{*}?{\cdot}R)) = \max(s\langle 9\rangle, \mathbf{F}(s\langle 10\rangle, \bot)) = s\langle 9\rangle$
where pruning removes all the other alternatives besides the first `()`, once the nullable location $s\langle 9\rangle$ has been found, and where $\bot = Der_{s\langle 9\rangle}(())$. Now $R^r =$ `eh|eht|tac` and $s^r =$ `"tac eht ees I"` and $s\langle 9\rangle^r = s^r\langle |s| - 9\rangle = s^r\langle 4\rangle$ where $(s^r)_4 =$ e. Then
$\mathbf{F}(s^r\langle 4\rangle, R^r) = \mathbf{F}(s^r\langle 5\rangle, \mathtt{h}|\mathtt{ht}) = \mathbf{F}(s^r\langle 6\rangle, ()|\mathtt{t}) = \max(s^r\langle 6\rangle, \mathbf{F}(s^r\langle 7\rangle, ())) = \max(s^r\langle 6\rangle, \max(s^r\langle 7\rangle, \lightning))$
So the result is $\langle s\langle 6\rangle, s\langle 9\rangle\rangle$, where $s\langle 6\rangle = s^r\langle 7\rangle^r$, with $s_{6,9-6} =$ `"the"` as the matched substring. ⊠

Observe that using $\mathtt{BT}(R^r)$ instead of R^r in the second pass would not work in general. In the above example the search would stop too early due to pruning in location $s^r\langle 6\rangle$ in s^r and therefore not reach the earliest location $s\langle 6\rangle$ in s.

5 IMPLEMENTATION

Here we give a high level overview of how *MatchEnd* can be materialized into a practical implementation. The main concern is the cost of the calls to *Der* and *Null.* We first describe how *alphabet compression* can be used to reduce the large alphabet size of Unicode. We then show how an effective caching scheme can be produced using the compressed alphabet and the property that *Null* only depends on *anchor-contexts.*

[11] Duplicate later alternatives in an alternation are always removed from the alternation, here we include them for clarity.

Finally, we will address the most central concern for avoiding unnecessary cache misses: syntactically different regexes that are semantically equivalent. We describe a rewrite system that is used to simplify regexes that arise during derivation. Crucially, the rewrite system guarantees *finiteness* of the space of derivatives. In the following sections we consider a fixed search pattern $R_0 \in \mathit{RE}$.

5.1 Alphabet Compression

The following steps are taken to construct a compressed alphabet algebra $\mathcal{A}$ that is tailor-made for R_0. Initially, R_0 is traversed to extract the set Γ of all those predicates that R_0 depends on, using a *binary decision diagram* (BDD) based algebra $\mathcal{B}$ for Unicode. Predicates in Γ are BDDs.

Minterm Computation. A *minterm* of Γ is a predicate $(\bigwedge S) \wedge \neg\bigvee(\Gamma\backslash S)$ for some $S \subseteq \Gamma$. *Minterms*(Γ) denotes the set of all minterms of Γ, we use the algorithm from [D'Antoni and Veanes 2014]. All minterms of Γ are satisfiable and mutually disjoint, and each satisfiable predicate in Γ is equivalent to a disjunction of some of its minterms. Let $\mathit{Minterms}(\Gamma) = \bar{\beta} = (\beta_i)_{i=0}^{k-1}$ so that $\min([\![\beta_i]\!]) < \min([\![\beta_{i+1}]\!])$ by using the underlying fixed order of the characters (by their numeric code points) in Σ. This provides a *fixed* choice of the minterm order which does *not matter* for correctness but is important to avoid nondeterminism in this step. The Unicode code point order is fixed across all platforms and runtimes and independent of runtime culture (System.Globalization.CultureInfo) or using the RegexOptions.CultureInvariant option.

Bitvector Algebra Computation. Next, we construct a *bitvector* algebra $\mathcal{A}$ with k-bit bitvectors. The Boolean operations of $\mathcal{A}$ are bit-wise arithmetic operations with nonzero test being satisfiability. $\mathcal{A}$ has minterms $\bar{\alpha} = (2^i)_{i=0}^{k-1}$ and $[\![\alpha_i]\!]_{\mathcal{A}} \overset{\text{DEF}}{=} [\![\beta_i]\!]_{\mathcal{B}}$. Each BDD $\psi^{\mathcal{B}}$ in Γ is translated into $\psi^{\mathcal{A}}$ as the bitwise-OR of all α_i such that $\mathbf{SAT}_{\mathcal{B}}(\beta_i \wedge \psi^{\mathcal{B}})$, so $[\![\psi^{\mathcal{B}}]\!]_{\mathcal{B}} = [\![\psi^{\mathcal{A}}]\!]_{\mathcal{A}}$. The rest of the engine now works solely with R_0 modulo $\mathcal{A}$.

We also precompute a *minterm lookup dictionary* μ that maps each $a \in \Sigma$ to the unique minterm $\mu(a)$ such that $a \in [\![\mu(a)]\!]$ and subsequently we use this dictionary and satisfiability in $\mathcal{A}$ to implement *membership* $s_i \in [\![\psi]\!]$ in $\mathit{Der}_{s\langle i\rangle}(\psi)$ by $\mathbf{SAT}(\mu(s_i) \wedge \psi)$. If $a \in \Sigma$ is an ASCII character then $\mu(a)$ uses an array and else a multi-terminal BDD to perform the lookup.

For example, let $R_0 = \texttt{\^{}\textbackslash w+\textbackslash d}$ then $\Gamma = \{\psi^{\mathcal{B}}_{\texttt{\textbackslash n}}, \psi^{\mathcal{B}}_{\texttt{\textbackslash d}}, \psi^{\mathcal{B}}_{\texttt{\textbackslash w}}\}$ and $\bar{\beta} = (\psi^{\mathcal{B}}_{\texttt{[\textbackslash W-[\textbackslash n]]}}, \psi^{\mathcal{B}}_{\texttt{\textbackslash n}}, \psi^{\mathcal{B}}_{\texttt{\textbackslash d}}, \psi^{\mathcal{B}}_{\texttt{[\textbackslash w-[\textbackslash d]]}})$ that gives us $\bar{\alpha} = (0001_2, 0010_2, 0100_2, 1000_2)$ in $\mathcal{A}$. So for example $\psi_{\texttt{\textbackslash w}} = \alpha_2 \vee \alpha_3 = 1100_2$ in $\mathcal{A}$.

5.2 Caching Der and Null

The calls to $\mathit{Der}_{s\langle i\rangle}(R)$ made in *MatchEnd* are not trivially cacheable because i changes for every call. However, *Der* only accesses s_i directly and the *anchor-context* $\widehat{s\langle i\rangle}$ indirectly through *Null*. This observation leads to the following caching scheme.

The cache for *Der* is a map $\text{€}_{\mathit{Der}} : \mathit{KIND} \times \mathit{RE} \times \bar{\alpha} \to \mathit{RE}$. Each call to $\mathit{Der}_{s\langle i\rangle}(R)$ first checks if $\text{€}_{\mathit{Der}}[\boldsymbol{\kappa}(s\langle i{-}1\rangle), R, \mu(s_i)]$ is defined, and if so, immediately returns it. Otherwise, $\mathit{Der}_{s\langle i\rangle}(R)$ is computed normally and $\text{€}_{\mathit{Der}}[\boldsymbol{\kappa}(s\langle i{-}1\rangle), R, \mu(s_i)]$ is updated to the result.

The cache for *Null* is a map $\text{€}_{\mathit{Null}} : \mathit{KIND} \times \mathit{RE} \times \mathit{KIND} \to \mathbb{B}$. Calls to $\mathit{Null}_x(R)$ will first check the cache at $\text{€}_{\mathit{Null}}[\boldsymbol{\kappa}(x-1), R, \boldsymbol{\kappa}(x)]$ and update it as necessary.

The consistency of €_{Der} and €_{Null} follow from: 1) if $b \in [\![\mu(a)]\!]$ then a and b are indistinguishable in R_0 and in any derivative derived from it; 2) definition of $\mathit{Null}_x(R)$ depends only on $\langle\boldsymbol{\kappa}(x{-}1), \boldsymbol{\kappa}(x)\rangle$. Recursive calls to *Der* and *Null* also use the caches, which is important for achieving a complexity over all of the calls to *Der* and *Null* that is linear in the length of a pattern where any loops have been unrolled. See Example 5.4 for a demonstration.

Example 5.1. Consider $R_0 = \top{*}?\texttt{\textbackslash w+\textbackslash b}$. The two minterms in $\mathcal{A}$ are $(1, 2) = (01_2, 10_2)$ where $[\![1]\!] = \Sigma\backslash[\![\psi_{\texttt{\textbackslash w}}]\!]$ and $[\![2]\!] = [\![\psi_{\texttt{\textbackslash w}}]\!]$, so \w is represented by $\psi_{\texttt{\textbackslash w}} = 2$. Let $s =$ "-ABC$\cdots$Z-" and $|s| = 1000$. Note that $\mu(\texttt{-}) = 1$ and $\mu(a) = 2$ if a is a word-letter. We show how €_{Der} evolves during the run of $MatchEnd(s, R_0)$.

x	R	s_i	$\mu(s_i)$	R' in $MatchEnd(x+1, R')$	Cache update
$s\langle 0\rangle$	R_0	-	1	$R_0 = Der_{s\langle 0\rangle}(R_0)$	$\text{€}_{Der}[\boldsymbol{\varepsilon}, R_0, 1] := R_0$
$s\langle 1\rangle$	R_0	A	2	$R_1 = Der_{s\langle 1\rangle}(R_0) =$ `\w*\b|`R_0	$\text{€}_{Der}[\mathbf{o}, R_0, 2] := R_1$
$s\langle 2\rangle$	R_1	B	2	$R_1 = Der_{s\langle 2\rangle}(R_1) =$ `\w*\b|\w*\b|`R_0 = `\w*\b|`R_0	$\text{€}_{Der}[\mathbf{w}, R_2, 2] := R_1$
$s\langle 3\rangle$	R_1	C	2	$R_1 = \text{€}_{Der}[\mathbf{w}, R_2, 2]$	
...					
$s\langle 998\rangle$	R_1	Z	2	$R_1 = \text{€}_{Der}[\mathbf{w}, R_2, 2]$	
$s\langle 999\rangle$	R_1	-	1	$R_0 = Der_{s\langle 999\rangle}(R_1) = \bot \mid R_0 = R_0$	$\text{€}_{Der}[\mathbf{w}, R_1, 1] := R_0$
$s\langle 1000\rangle$	R_0	N/A			

Observe that the caches populate quickly, after which and the hot loop amounts to just reading €_{Der}. In the end $MatchEnd(s\langle 0\rangle, R_0)$ returns $s\langle 999\rangle$ because through the run only $Null_{s\langle 999\rangle}(R_1)$ is true. Although not shown, the behavior for €_{Null} would be similar to that of €_{Der}. ⊠

Example 5.1 implicitly used the *rewrite rules* that we will discuss next. The elimination of duplicate alternatives at $s\langle 2\rangle$ exhibits the most critical rule for guaranteeing that $|\text{€}_{Der}|$ eventually converges.

5.3 Rewrite Rules

Our system implements the following rewrite rules:

$$\bot R \longrightarrow \bot \quad R\bot \longrightarrow \bot \quad \bot|R \longrightarrow R \quad R|\bot \longrightarrow R \quad \top{*}|R \longrightarrow \top{*} \quad R() \longrightarrow R \quad ()R \longrightarrow R$$

As well as the following rules that we give names to for clarity:

$$\textsc{OptOpt}\text{: } R\{0,1,b_2\}\{0,1,b_1\} \longrightarrow R\{0,1,b_1 \lor b_2\} \qquad \textsc{AltAssoc}\text{: } (R \mid S) \mid T \longrightarrow R \mid (S \mid T)$$

$$\textsc{AltUni}\text{: } R_1 \mid \cdots \mid R_n \longrightarrow S_1 \mid \cdots \mid S_n \textbf{ where } S_i = (\textbf{if } \forall j{<}i(R_j \neq R_i) \textbf{ then } R_i \textbf{ else } \bot)$$

$$\textsc{AltSub}_{\lessapprox}\text{: } R \mid S \longrightarrow T \textbf{ if } R \lessapprox S \textbf{ and } T = FoldAlt(R,S) \neq \bot \qquad \textsc{AltSub}_{\gtrapprox}\text{: } R \mid S \longrightarrow R \textbf{ if } S \lessapprox R$$

These rules are applied in the constructors of the *RE* datatype, which ensures that regexes that are non-canonical under them cannot be constructed. The AltSub rules rely on a more involved notion of *subsumption* and will be discussed in Section 5.4. In addition to the rules above, we augment the AltSub rules to also handle cases of the form $R \mid (S \mid T)$ as follows: if a rule would rewrite $R \mid S \longrightarrow U$ then we will rewrite $R \mid (S \mid T) \longrightarrow U \mid T$. The rule AltUni played a key role in Example 5.1 to remove the duplicate alternative from `\w*\b|\w*\b|`R_0.

Example 5.2. To illustrate how the rewrite rules interact with derivation, consider the regex $R_0 = \top{*}?$`(she|he)` with an input s = "she". The first two derivatives required are expanded below with the rewrites applied shown on the right.

$$\begin{aligned} R_1 &= Der_{s\langle 0\rangle}(R_0) = Der_{s\langle 0\rangle}(\texttt{she|he})|Der_{s\langle 0\rangle}(\top{*}?)(\texttt{she|he}) \\ &= Der_{s\langle 0\rangle}(\texttt{s})\texttt{he}|Der_{s\langle 0\rangle}(\texttt{h})\texttt{e}|\top{*}?(\texttt{she|he}) \\ &= ()\texttt{he}|\bot\texttt{e}|\top{*}?(\texttt{she|he}) = \texttt{he}|\top{*}?(\texttt{she|he}) \end{aligned} \qquad ()\texttt{he} \to \texttt{he},\ \bot\texttt{e} \to \bot,\ \texttt{he}|\bot \to \texttt{he}$$

$$\begin{aligned} R_2 &= Der_{s\langle 1\rangle}(R_1) = Der_{s\langle 1\rangle}(\texttt{he})|Der_{s\langle 0\rangle}(\texttt{she|he})|\top{*}?\texttt{he} \\ &= \texttt{e}|\bot|\texttt{e}|\top{*}?(\texttt{she|he}) = \texttt{e}|\top{*}?(\texttt{she|he}) \end{aligned} \qquad \texttt{e}|\bot|\texttt{e} \xrightarrow{\textsc{AltUni}} \texttt{e}|\bot|\bot \to \texttt{e}$$

⊠

These rewrite rules ensure that from any R_0 a finite set of derivatives are reachable [Brzozowski 1964]. However, the problem of recognizing the equivalence of two regexes is still PSPACE-hard [Stockmeyer and Meyer 1973], which means that this rewrite system need and should not be complete. Rather, the rules should target the shapes of regexes that arise from derivation and recursive application of the rewrite rules. We view the task of selecting rewrite rules as a design problem that must balance the cost and power of rewriting.

Theorem 5.3. *The implementation of $Match(s, R)$ has $O(|s|)$ complexity for all $s \in \Sigma^*$ and $R \in RE$.*

Proof. *MatchEnd* runs at most twice over s. The sizes of $€_{Der}$ and $€_{Null}$ do not depend on $|s|$. $\square$

In effect, $€_{Der}$ and $€_{Null}$ form an automaton whose states are conditionally accepting based on the next location kind. Section 5.5 describes a *state caching* technique that further exploits this.

5.4 Subsumption for Reducing Alternations

The AltSub rules capture how *subsumption* can be used to eliminate alternatives. We write $S \leqq R$ when a regex R *subsumes* S (or S is *subsumed by* R), or formally $\mathcal{M}(S) \subseteq \mathcal{M}(R)$. Observe that $R \equiv S \Leftrightarrow R \leqq S \leqq R$. Let $R \equiv_{\text{BT}} S$ stand for $R \equiv S$ **and** $\mathsf{BT}(R) \equiv \mathsf{BT}(S)$ as a strong equivalence that preserves both backtracking as well as nonbacktracking semantics. Our system includes a sound but incomplete test $\lessapprox$ for subsumption implemented with the following inference rules. In the following let $X, Y, Z \in RE$. An important property of the rules is that they maintain $\equiv_{\text{BT}}$ in AltSub rules, which is the only context where they are being used. The standard notation for a regex R being *eagerly optional* is $R? \overset{\text{DEF}}{=} R\,|\,()$ and being *lazily optional* is $R?? \overset{\text{DEF}}{=} ()\,|\,R$.

$$\frac{}{\bot \lessapprox X} \quad \frac{}{X \lessapprox X} \quad \textsc{SubLazy}_1\!: \frac{X \lessapprox Y}{ZX \lessapprox Z??Y} \quad \textsc{SubLazy}_2\!: \frac{Null_\forall(Z)\ X \lessapprox Y}{Z??X \lessapprox ZY} \quad \textsc{SubNull}\!: \frac{Null_\forall(Z)\ X \lessapprox Y}{X \lessapprox ZY}$$

As an example of how subsumption enables the AltSub rewrites, by SubLazy$_1$ it holds that $ZY \lessapprox Z??Y$, which enables the rewrite $Z??Y\,|\,ZY \xrightarrow{\textsc{AltSub}_{\gtrapprox}} Z??Y$. To see that the rewrite is correct, observe that $Z\{0,1\}? \equiv_{\text{BT}} ()\,|\,Z$ and thus $Z??Y \equiv_{\text{BT}} (Y\,|\,ZY)$. The rewrite can be thought of as an implicit application of AltUni onto $Y \mid ZY \mid ZY$.

The SubNull rule is the workhorse for detecting *nullable prefixes* and would, for example, establish that $\mathtt{a} \lessapprox \mathtt{a?a}$ and thus $\mathtt{a?a|a}$ is rewritten to just $\mathtt{a?a}$ via AltSub$_{\gtrapprox}$. It would not, however be valid to rewrite $\mathtt{a|a?a}$ into $\mathtt{a?a}$ because the preference for match end locations would be altered. For example, $\mathtt{"aa"}\langle 0\rangle \xrightarrow{\mathsf{BT}(\mathtt{a|a?a})} \mathtt{"aa"}\langle 1\rangle$ while $\mathtt{"aa"}\langle 0\rangle \xrightarrow{\mathsf{BT}(\mathtt{a?a|a})} \mathtt{"aa"}\langle 2\rangle$.

The correct rewrite in this case is $\mathtt{a|a?a} \rightarrow \mathtt{a??a}$ where the right side's eager option $\mathtt{a?}$ has been "folded" into the left alternative as a lazy option. The AltSub$_{\lessapprox}$ rule handles such rewrites with the *FoldAlt* function, defined as follows:

$$FoldAlt(R, S) \overset{\text{DEF}}{=} \textbf{if } (R \lessapprox S \textbf{ and } P = SubsPrefix(R, S) \neq \bot) \textbf{ then } P??R \textbf{ else } \bot$$

$$SubsPrefix(R, S) \overset{\text{DEF}}{=} \textbf{if } S \lessapprox R \textbf{ then } () \textbf{ else } (\textbf{if } S = PT \textbf{ and } R \lessapprox T \textbf{ then } P\cdot SubsPrefix(R, T) \textbf{ else } \bot)$$

FoldAlt implements a rule that if S subsumes R due to there being a nullable prefix P such that $S = PT$ then $R|S$ can be rewritten to $P??R$. The SubLazy rules are designed to prove subsumption for the shapes of regexes resulting from *FoldAlt*.

These subsumption based rewrites proved critical for ensuring acceptable performance for patterns that include long concatenations of nullable regexes. The danger with such patterns is that if a pattern like $\mathtt{a?a?}\cdots\mathtt{a?}$ is allowed to evolve into an alternation of linear size, it becomes difficult to avoid quadratic behavior over multiple derivations. The following examples show how the AltSub rules work to avoid such blow-up.

Example 5.4. Consider the regex $R = \mathtt{a?a?a?}$ and input $s = \mathtt{"aaa"}$. For $x = s\langle 0\rangle$ it holds that:

$$\begin{aligned} Der_x(R) &= Der_x(\mathtt{a?})\mathtt{a?a?}\,|\,Der_x(\mathtt{a?a?}) \\ &= Der_x(\mathtt{a?})\mathtt{a?a?}\,|\,Der_x(\mathtt{a?})\mathtt{a?}\,|\,Der_x(\mathtt{a?}) \\ &= \mathtt{a?a?|a?|()} = \mathtt{a?a?|a?} = \mathtt{a?a?} \end{aligned} \qquad \Bigg| \qquad \mathtt{a?|()} \xrightarrow{\textsc{AltSub}_{\gtrapprox}} \mathtt{a?},\ \mathtt{a?a?|a?} \xrightarrow{\textsc{AltSub}_{\gtrapprox}} \mathtt{a?a?}$$

The AltSub$_{\gtrapprox}$ rule helped avoid alternations in the result. For longer concatenations of nullable regexes, this would avoid an $O(n^2)$ blow-up in the size of the result. Now consider the regex $S = \mathtt{a??a??a??}$ instead. For $x = s\langle 0\rangle$ it holds that:

$$\begin{array}{l|l} \begin{aligned} \mathit{Der}_x(S) &= \mathit{Der}_x(\texttt{a??})|\mathit{Der}_x(\texttt{a??})\texttt{a??}\,|\,\mathit{Der}_x(\texttt{a??})\texttt{a??a??} \\ &= \texttt{()}\,|\,\texttt{a??}\,|\,\texttt{a??a??} \\ &= \texttt{a??}\,|\,\texttt{a??a??} = \texttt{a??a??} \end{aligned} & \begin{array}{l} \texttt{()}\,|\,\texttt{a??} \xrightarrow{\textsc{AltSub}_{\lessapprox}} \texttt{(a??)??} \xrightarrow{\textsc{OptOpt}} \texttt{a??} \\ \texttt{a??}\,|\,\texttt{a??a??} \xrightarrow{\textsc{AltSub}_{\lessapprox},\textsc{OptOpt}} \texttt{a??a??} \end{array} \end{array}$$

Similar quadratic blow-up is avoided here as above. ⊠

Depth Limit. The inference rules for subsumption are realized by implementing $\lessapprox$ as a function that tries to apply each rule in turn by checking their conditions *with recursive calls to itself when required.* This, however, may lead to deep recursions that are ultimately unproductive. For example, consider the regexes $R = \texttt{a?}\cdots\texttt{a?}$ and $S = \texttt{a?}R$. For a query $S \lessapprox R$ the SubNull rule would be tried until a query $S \lessapprox \texttt{a?}$ is made, at which point SubNull can no longer apply due to the left side not being a concatenation. To limit this kind of unproductive work we introduce a *recursion depth limit* for any subsumption query triggered by a rewrite rule. If the limit is reached $\lessapprox$ returns with failure. In our implementation this depth limit is set to 50, which we found to be sufficient to cover realistic patterns while limiting the impact for malicious patterns.

Subsumption Hinting. Note that in the example of the paragraph above, $R \lessapprox S$ does hold and, furthermore, between the $\textsc{AltSub}_{\gtrapprox}$ and $\textsc{AltSub}_{\lessapprox}$ rules both directions of subsumption will be checked if the alternation $R|S$ is constructed. This is particularly relevant for $\mathit{Der}_x(L \cdot R)$ when L is nullable (see Section 4.2). The order of the alternation between $\mathit{Der}_x(R)$ and $\mathit{Der}_x(L) \cdot R$ varies depending on the high-nullability of L, such that $\textsc{AltSub}_{\lessapprox}$ is more likely to apply when L is high-nullable and $\textsc{AltSub}_{\gtrapprox}$ when L is not. To take advantage of this our implementation passes a hint from *Der* to the alternation constructor to indicate which rule should be tried first.

Note on Extensibility. We have found $\lessapprox$ and *FoldAlt* to serve as useful points of extensibility for the rewriting system. Though the rules presented above mainly deal with subsumption due to nullable prefixes, we have on-going work on enabling regex subcapture matching that introduces new types into *RE*. The $\lessapprox$ and *FoldAlt* functions can be extended to handle these new types in a modular way. Loop subsumption optimizations in the style of [Saarikivi et al. 2019] but corrected for backtracking semantics are another viable extension, e.g., the rule $R\{0,m\}?\,|\,R\{0,n\}? \rightarrow R\{0, \max(m,n)\}?$.

5.5 Other Implementation Considerations

Equality Checks. The caching and rewrite rules described in the previous sections rely on knowing when regexes are structurally equal. To make this cheap, our implementation *interns* all regexes such that pointer equality coincides with structural equality.

State Caching. As shown is Example 5.1, for sufficiently long inputs the vast majority of calls to *Der* and *Null* will immediately hit the $€_{\mathit{Der}}$ and $€_{\mathit{Null}}$ caches. We implement a *state graph* construction that further optimizes these top-level cache lookups.

A set Q of *states seen so far* is maintained, where each state is a pair $\langle\kappa, R\rangle \in \mathit{KIND} \times \mathit{RE}$ of the kind κ of the previous character that transitioned into this state and its derivative R. The parameter R of *MatchEnd* is replaced by a state in Q with $\langle\boldsymbol{\varepsilon}, R_0\rangle$ as the initial state. Top-level lookups into $€_{\mathit{Der}}$ resolve instead through a lookup table $£_{\mathit{Der}} : Q \times \bar{\alpha} \rightarrow Q$. For each entry $€_{\mathit{Der}}[\kappa, R, \mu(a)] = R'$ resulting from a top-level call to *Der*, the lookup table will have an entry $£_{\mathit{Der}}[\langle\kappa, R\rangle, \mu(a)] = \langle\boldsymbol{\kappa}(a), R'\rangle$. A similar lookup table $£_{\mathit{Null}} : Q \times \mathit{KIND} \rightarrow \mathbb{B}$ is introduced for *Null*.

By associating each state with an index, these lookup tables can be implemented as flat arrays that are grown on demand as new states are encountered. Furthermore, because $£_{\mathit{Der}}$ no longer depends on the kind of the previous character, $\mathit{MatchEnd}(s\langle i\rangle, R)$ performs just one read into s_i per iteration. Altogether, $£_{\mathit{Der}}$ and $£_{\mathit{Null}}$ greatly improve the performance of the matching loop.

NFA Mode. When the size of Q reaches a certain threshold[12] the search engine switches into *NFA mode* and the current state $s \in Q$ is converted into an ordered set S such that $\forall i(\pi_1(s) = \pi_1(S_i))$ and $\pi_2(s) \equiv |_{i=1}^{|S|}\pi_2(S_i)$. Essentially, alternation inside the state is broken up into separate states. Derivatives are computed for each S_i separately, broken up again into ordered sets and inserted into the successor S' in order. One subtlety that arises is that the NFA mode must reimplement part of *Prune*. When computing derivatives for S, if $\pi_2(Q_{S_i}) = \mathtt{BT}(T)$ for some i and T is nullable in the current context, then all S_j for $j > i$ are ignored. This mirrors how alternations are pruned.

Observe that the NFA mode is a space-time tradeoff, since iterating S takes more time but uses less space since only the components of S are cached. The classical analogue here is that each S_i corresponds to a partial derivative [Antimirov 1995] in contrast to a (full) derivative $|_{i=1}^{|S|}\pi_2(S_i)$ [Brzozowski 1964]. To minimize the performance impact, we implement S as the sparse ordered set data structure for small integers described in [Briggs and Torczon 1993], which allows for $O(1)$ ordered set insertion and ordered iteration over contiguous memory. The NFA mode also uses a separate lookup table $£_{Der}^{NFA} : Q \times \bar{\alpha} \longrightarrow Q^*$ that directly caches the ordered sets of target states.

Fixed Length Matches. The second call to *MatchEnd* in *Match* can be avoided when the match is found in a fixed length fragment of R_0. With R_0 = `ab|a+b` and s = `"abc"` the derivative at $s\langle 2\rangle$ would be `()` | R_0, which has lost track of the fact that the match happened through `ab`. To overcome this we tag R_0 with fixed-length markers $\mathtt{()}_n$ that act like epsilon in *Der* and *Null*, e.g., $R_0 = \mathtt{ab()}_2 \mid \mathtt{a+b}$. With this the derivative at $s\langle 2\rangle$ would instead be $\mathtt{()}_2 \mid \mathtt{()} \mid R_0$, which lets us resolve the whole match as $\langle s\langle 0\rangle, s\langle 2\rangle\rangle$ without having to call $MatchEnd(s\langle 2\rangle^r, R_0^r)^r$. This optimization roughly doubles asymptotic throughput for a common class of patterns that are alternations of fixed strings.

Subcaptures. The matching algorithm uses *tagged derivatives* with tags recording start and end locations of sub-patterns, during a *third phase*, run only when *grouping constructs* [Microsoft 2021c] are present, and results in incrementally creating a variant of a *tagged NFA* [Laurikari 2000].

6 .NET INTEGRATION

Our implementation is integrated as a new backend for the System.Text.RegularExpressions library in the .NET runtime and may be triggered with the RegexOptions.NonBacktracking flag. The feature is available in .NET7 – released in November 2022.

The implementation is pure C#, but integrates with existing *prefix optimizations* for other .NET regex engines. These optimizations target patterns like `abc\w+`, where the fixed string `"abc"` can be efficiently searched for with vectorized string search procedures. We also employ a number of low-level optimization tricks to ensure that the hot matching loop is as fast as possible, such as using local variables to avoid read/writes to ref/out values, and having *MatchEnd* be a generic function parameterized by a DFA or NFA state handler constrained to a `struct` with a static interface, which allows inlining of state handling logic for both modes. We encourage interested readers to study the open-source implementation in [Microsoft 2022].

The integration of NonBacktracking into .NET runtime went through extensive testing, involving thousands of regexes covering multiple standard test suites, with tests that verify expected and mutually equal match results for all the .NET regex engines (NonBacktracking, Compiled, and None) including all supported RegexOptions and covering all platforms supported by .NET. The tests helped uncover multiple bugs in both implementation and theory. An early bug in NonBacktracking was related to case-insensitivity in combination with complement in character classes where case-insensitivity was applied *after* complement (as in SRM), while the Unicode standard is to apply case-insensitivity *before* complement. E.g., `(?i:[^B])` ≡ `[^Bb]` and

[12]This threshold is 10000 by default in .NET7.

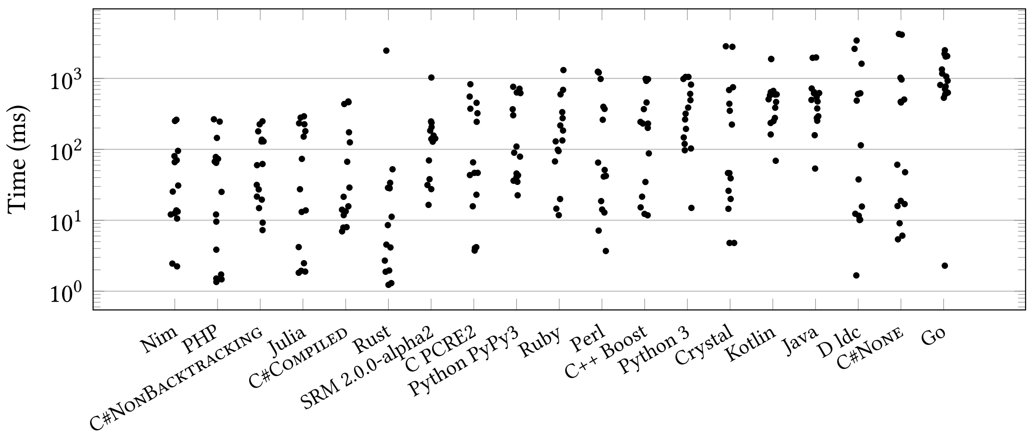


Fig. 1. Various matchers on 15 patterns for the works of Mark Twain, ordered by average matching time.

`(?i:[\0-AC-\uFFFF])` ≡ `[\0-\uFFFF]`[13] although `[^B]` ≡ `[\0-AC-\uFFFF]`. Certain rewrites by the regex parser, e.g., rewrites to *atomic* regexes `(?>...)`, are not supported in NonBacktracking and had to be omitted. We found several bugs in early versions of pruning rules, such as $Prune_x(R\{m\}) \longrightarrow Prune_x(R)\{m\}$ being *incorrect* when R contains anchors, and in rewrite rules adopted from SRM that violated backtracking semantics, such as the loop rewrite rule mentioned in Section 1. These findings motivate further rigor: we intend to formalize our theory using a proof assistant to provide even more confidence for the current implementation and also to serve as a platform for verifying future optimizations.

7 EXPERIMENTS

In this section we evaluate the new engine against the other regular expression matchers, including .NET's other main backends and the SRM library [Saarikivi et al. 2019]. We refer to the .NET engines by the name of the `RegexOptions` enum member that triggers them, i.e., None, Compiled and NonBacktracking. All experiments below use .NET 7 Release Candidate 2.

When matching "well behaved" patterns (i.e. no catastrophic backtracking or state space explosion), performance is dominated by 1) the pre/post/infix *search optimizations* available and 2) the byte-to-byte *transition logic* in the innermost matching loops. The search optimizations are largely orthogonal to the matching approach, while both derivative- and automata-based engines will have very similar innermost loops implementing transitions in a cached/lazily constructed DFA. Therefore, the most interesting aspect to compare across regex engines is their performance with potential outliers and this is what our evaluation focuses on.

7.1 Comparison with Standard Library Matchers

First we compare NonBacktracking with standard matchers for 16 different programming languages. We took a popular cross language benchmark [Juárez 2020] and to add potential outliers we replaced its dataset[14] with the "Twain" benchmark [Herczeg 2015], consisting of 15 patterns for the collected works of Mark Twain. We excluded several redundant matchers as well as ones

[13]The concrete syntax `(?i:`R`)` means that R is evaluated locally with RegexOptions.IgnoreCase, thus treating all character classes that occur in R as *case-insensitive*. Case-insensitivity of some Unicode characters, such as the Turkish İ, moreover depend on the *runtime culture* (System.Globalization.CultureInfo), that was another source of subtle bugs.
[14]The original dataset is three patterns for parsing emails, URIs and IPs, which are well behaved in all engines.

missing support for case-insensitive groups and added Compiled, NonBacktracking and SRM version 2.0.0-alpha2. We verified that all of the .NET engines produce the same number of matches for each pattern. The measurements were performed on an Azure Dsv4-series VM running an Ubuntu 18.04 Docker image with 2 cores of an Intel Xeon 8272CL processor and 8 GB of memory. Figure 1 presents the results.

C#'s None and Compiled have similar performance profiles due to using the same matching logic, but the code generation helps speed up all patterns and already places C# near the top of the pack. NonBacktracking further improves the performance and places it 3rd in average matching time among the included matchers. Compared to both Compiled and SRM 2.0.0-alpha2 the improvement is mainly from avoiding outliers and having a more consistent performance profile. The average/best cases seem largely equal between NonBacktracking and Compiled. General search optimizations are shared between all the engines. In the hot (DFA mode) matching loop, NonBacktracking uses additional optimizations that are specific to the algorithms discussed in Section 5, such as dead-end state detection for early search failure.

Incidentally, the result for Rust further highlights the importance of avoiding outliers, as it would be a clear winner if not for the pattern `[a-q][^u-z]{13}x`. The crux of the pattern is that `[a-q]` denotes a *subset* of `[^u-z]` much like in the classical example `a.{13}`. This pattern is challenging for DFA based engines, as the loop can be entered multiple times, leading to a 2^{13} factor in number of states. While lazy exploration of the state space with derivatives helps, this pattern is still the slowest one for SRM 2.0.0-alpha2. Optimizations in NonBacktracking have made this pattern no longer visible as an outlier. SRM cached states in a fixed-size array and a dictionary for overflow. NonBacktracking instead grows the array on-demand, avoiding expensive dictionary lookups and improving performance for patterns that create many states.

Even though the Go matcher in Figure 1 supports the RE2 regex dialect it is a reimplementation in Go. We view the Rust engine as a better version of RE2, which implements the same techniques but with many more micro-optimizations.

7.2 Case Study: Word Phrase Matching

We extracted a dataset of 184 patterns from an industrial word phrase matching use case. They use the `\b` anchor extensively and include large alternations. The data is 5 MB of English dialog extracted from the MultiWOZ dataset [Budzianowski et al. 2018].

We measured matching time for each pattern using the .NET Performance tool [Microsoft 2021b], which we modified to use the new patterns and data. The experiment ran on an Intel Core i7-1185G7 machine with 32 GB of memory running Windows 11. Figure 2 presents the results as a scatter plot.

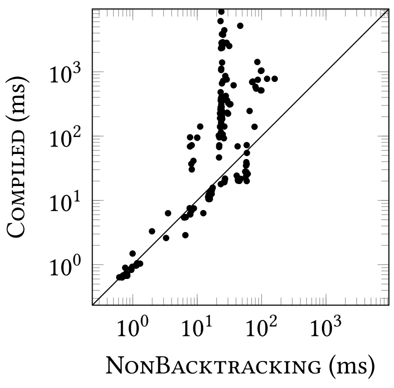


Fig. 2. Log-scale matching time of 184 patterns from an industrial word phrase matching dataset on 5 MB of English dialog. Points above the line are faster with NonBacktracking.

The results highlight how in NonBacktracking any number of alternations can be handled with little extra cost, while backtracking engines match each option separately. The geometric mean speedup is 4.7× and the difference in total matching time is even larger at 24×. For applications that depend on regular expression matching, this kind of performance differential can be critical. Furthermore, the high variability in performance may lead to inconsistent user experiences: for NonBacktracking the slowest pattern takes only 0.26 seconds longer than the fastest one, while for Compiled the difference is 13.6 seconds.

7.3 Case Study: Credential Scanning

CredScan [Microsoft 2021a] is a tool that scans source code and other cloud datasets for leaked credentials and other sensitive content. It uses a large set of regular expressions to identify potential leaks. We modified the tool to use NonBacktracking and compare end-to-end scanning time against None and Compiled in .NET 7. Figure 3 presents the results from scanning 188 production repositories comprising 15 GB of data. Each point in the two scatter plots represents a single repository and the time it took to scan it with the baseline and NonBacktracking.

NonBacktracking gives overall speedups of 26% and 9% against None and Compiled, respectively. This is despite a lack of severe outliers for the backtracking engines, which were the source of the massive speedups in Section 7.2: the best speedups for any single repository are 2.5× and 1.5× against None and Compiled, respectively.

The speedups are still driven by NonBacktracking's more consistent performance profile, which is visible as lower scanning times on larger repositories. For small repositories None, especially, is faster than NonBacktracking. This is due to higher initialization overhead for NonBacktracking of the hundreds of patterns CredScan includes. We believe the alphabet compression step described in Section 5.1 to be the main contributor. Against Compiled the difference for the small repositories is smaller, as the compilation it performs results in similar per-pattern overheads. We are investigating ways to reduce the initialization overhead of NonBacktracking through techniques such as code generation.

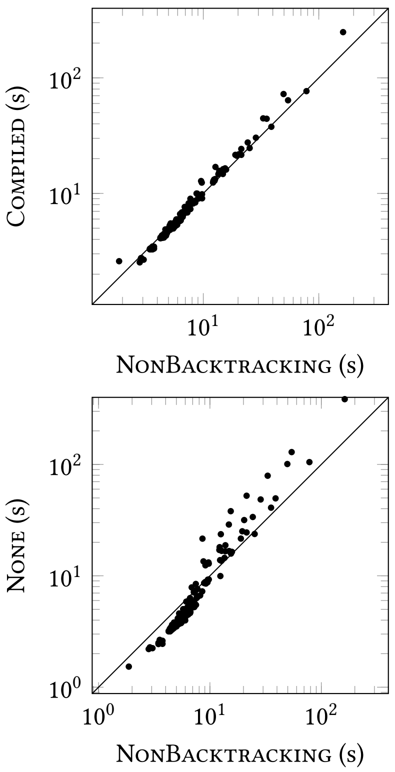


Fig. 3. Log-scale end-to-end scanning time of CredScan with different regex engines applied to 188 source code repositories. Points above the line are faster with NonBacktracking.

8 RELATED WORK

The discussion here is limited to *regular* features of regexes and to derivative and automata based techniques. The complete regex language of ECMAScript, covering non-regular features such as backreferences and balancing groups (see [Loring et al. 2019]) is out of scope of the work here.

Derivative Foundations. Derivatives were introduced in [Brzozowski 1964] for DFAs and reformulated in [Antimirov 1995] as *partial derivatives* for NFAs. Derivatives for symbolic regexes were studied in [Keil and Thiemann 2014]. These works do not study anchors or lookarounds.

Alphabet Compression in Regex Matchers. Use of *minterms* in our case is key to fast and straightforward state transition memoization while incrementally maintaining a state transition graph both for DFAs and NFAs by keeping the size of the out-degree of transitions as small as possible, in a *regex dependent* manner. Use of minterms as an alphabet compression technique for regexes was first observed in [Hooimeijer and Veanes 2011]. Other tools like RE2 apply different techniques where the input is converted to UTF8, *independent of regex*, and a specialized automaton is used to recognize UTF8 encoding during reading of the input [Cox 2010]. In .NET this would not be possible in general due to, e.g., UTF16 surrogates being valid characters with no UTF8 encoding.

Derivative Based Matching. Derivatives were originally studied in [Fischer et al. 2010; Owens et al. 2009] for *IsMatch*. The work in [Sulzmann and Lu 2012] studies *MatchEnd* using Antimirov derivatives and POSIX (leftmost longest) semantics. Subsequently in [Ausaf et al. 2016] that work is improved and elegantly formalized in Isabelle/HOL and also extended for Brzozowski derivatives. This work is very inspiring for us as we also intend to formalize our algorithm using a proof assistant. The key similarity to our work is that ordering of alternatives arising from derivatives of concatenations $R{\cdot}S$ is based on a variant of high-nullability of R that determines the order in which the resulting derivatives are then processed to maintain POSIX semantics. The key difference to our work, is that nullability of regexes in these works is not context dependent and anchors and counting are not considered. For regexes with anchors the classical language properties $L(R{\cdot}S) = L(R){\cdot}L(S)$ and $L(R^*) = L(R)^*$ (axioms (4) and (6) in [Ausaf et al. 2016]) *do not hold.*

Anchors have so far been treated in ad-hoc fashion. In *match generation* anchors are a key ingredient defining the boundaries of a match and therefore cannot be eliminated by preprocessing. To this end, and to the best of our knowledge, the theory presented here is novel. In [Wingbrant 2019] anchors are treated as imaginary characters in the input using classical derivatives. This approach does not preserve backtracking semantics, which we learned the hard way – *this was also our initial approach.* Moreover, there are critical semantic differences compared to the classical treatment of loops in the foundations of derivatives (recall Section 3.5) when anchors are used.

While some aspects of NonBacktracking build on SRM [Saarikivi et al. 2019], the current work is based on a fundamental redesign of the foundations to support anchors and backtracking semantics. Moreover, the top-level matcher of SRM is less efficient because it needs *three* passes over the input to locate a match, instead of two.

Automata Based Matching. Modern automata based regular expression matchers such as RE2 [Cox 2010] and grep [GNU 2020] also use state graph memoization similarly to NonBacktracking. One can roughly classify these matchers as highly optimized variants of [Thompson 1968] enhanced with DFA state caching, or in the case of Hyperscan [Intel Co 2021] based on [Glushkov 1961]. As far as we know, there are no analogues of our Correctness theorem for these engines, that would be difficult to state since anchors and backtracking are outside classical automata theory.

A key contribution of our work is the set of rewrite rules in Section 5.3, which gives powers that we believe are unavailable to automata-caching engines that maintain a lazy DFA construction even if the minimal DFA might be small, since the upfront cost of DFA construction and minimization is undesirable for the target use case. Derivatives allow DFA-minimizing optimizations to be applied on-the-fly. Relating exactly which rewrite rules go beyond what engines like RE2/Rust do, is difficult, but in [Owens et al. 2009, Table 1] it is shown that derivatives with good rewrite rules often provide minimal DFAs. Similar conclusions are drawn in [Sulzmann and Lu 2012, Section 5.4] relating NFA sizes arising from Thompson's and Glushkov's, versus Antimirov's constructions. The subsumption-based rules in Section 5.3 are certainly beyond what is easily possible in automata-caching, because in automata the subsumption checks would require global analysis.

The backtracking semantics and anchors support cover necessary modern features, while the alphabet compression and caching scheme are necessary for good performance. *Greedy* matching algorithm for backtracking (PCRE) semantics was originally introduced in [Frisch and Cardelli 2004], based on ϵ-NFAs, while maintaining matches for eager loops. We do not believe an extension to anchors is straightforward in this work, because anchors are context conditions with no direct semantics in classical automata. In particular, [Frisch and Cardelli 2004, Proposition 2] assumes the axiom $L(R{\cdot}S) = L(R){\cdot}L(S)$ that fails with anchors. Derivatives also avoid the issue of ϵ-transitions in [Frisch and Cardelli 2004] that do not arise with derivatives, that is also true in [Ausaf et al. 2016;

Sulzmann and Lu 2012]. Even if successfully extended with modern features the constructions in [Cox 2010; Frisch and Cardelli 2004], result in an NFA that loses the local meaning of states, which derivatives do maintain and thus enable further rewriting-based optimizations.

Our proof of correctness helped us find bugs, and gives confidence for future extensions. Our work to modernize derivatives will enable features out of reach for engines like RE2, such as support for *intersection* and *complement*, which SRM has but we cut for now due to lack of support in .NET7's regex dialect. Further minimizations with rewrite rule-based optimizations, such as, a backtracking compatible version of SRM's loop-subsumption rule, is ongoing work.

The two phases of the top-level matching algorithm in .NET NonBacktracking – *forward phase* to find the match end location and *backward phase* to find the match start location – correspond to those in RE2 [Cox 2010]. The preliminary *third phase* for *sub-capture* support, which is also derivative based, is different. In [Cox 2010] the third phase will sometimes fall back to a backtracking NFA execution, losing input-linearity. In our case, the third phase is also a linear pass over the matched substring that uses *tagged derivatives* with tags recording start and end locations of sub-patterns to be captured, resulting in a variant of a *tagged NFA* [Laurikari 2000].

When switching to NFA mode (recall Section 5.5) the overall effect is similar to [Cox 2010], where mapping from DFA states to sets of NFA states has to be maintained, but in our case Brzozowski-style derivatives can switch to Antimirov-style derivatives without any prior bookkeeping.

Derivative Based Analysis. SMT solvers that support regexes via derivatives are Z3 [de Moura and Bjørner 2008] that now uses a generalization of derivatives called *transition regexes* [Stanford et al. 2021], and CVC4 [CVC4 2020; Liang et al. 2015] that uses Antimirov derivatives. In a recent study [Turoňová et al. 2020], derivatives were used to extend NFAs with counting arising from Antimirov-style exploration of standard regexes supporting finite loops. Kleene algebras with tests [Kozen 1997] have also been used to work with derivatives of symbolic classical regexes where predicates are encoded by BDDs [Pous 2015]. An interesting direction for future work would be to extend location based derivatives with *intersection* and *complement* and to also support *lookarounds* and *MatchEnd* in SMT solvers, in addition to *IsMatch* over extended regular expressions.

Match Generation Semantics. The two most well-known standards for matching are PCRE and POSIX with a formalization of POSIX based on tagged automata [Laurikari 2000]. There is also a Boost variant of POSIX [Berglund et al. 2021]. PCRE semantics of NonBacktracking is needed in .NET for compatibility with the other regex backends Compiled and None. PCRE is also the more widely adopted semantics. POSIX finds the longest leftmost match, while PCRE stops on the first match (according to backtracking). E.g., consider the regex `(a|ab)*` and the input string `"abab"`, where the PCRE match is the prefix `"a"` of the input while the POSIX match is the whole input.

9 CONCLUSIONS

The new nonbacktracking regex backend for .NET delivers significant speedups for real-world use cases: 4.7× speedup on an internal word phrase matching task and 9% better *end-to-end* throughput on a benchmark of CredScan [Microsoft 2021a]. We believe it can both enable new use cases and significantly reduce resource requirements for users of .NET regular expressions. The correctness of the matching algorithm has not only allowed us to integrate our work into .NET with confidence but also experimentally confirmed mutual semantic consistency among all the backends. The framework itself is open source and available for use as a research platform to explore new ideas.

ACKNOWLEDGMENTS

We thank all our reviewers and shepherd *Konstantinos Mamouras* for valuable feedback during the review process. We particularly appreciated the suggestions that lead to a more detailed comparison with related work, and a clearer positioning of the evaluation results.

DATA AVAILABILITY STATEMENT

The implementation is available in .NET Runtime 7 under the `RegexOptions.NonBacktracking` flag. Source code is available in the .NET Runtime repository [Microsoft 2022]. Additionally, an artifact with instructions for reproducing the results in Section 7.1 is available [Moseley et al. 2023b]. The artifact does not cover Sections 7.2 and 7.3, as both of them concern proprietary datasets that could not be made available.

REFERENCES

Valentin Antimirov. 1995. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science* 155 (1995), 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 (LNCS, Vol. 9807)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer, 69–86. https://doi.org/10.1007/978-3-319-43144-4_5

Adam Baldwin. 2016. *Regular Expression Denial of Service affecting Express.js*. http://web.archive.org/web/20170116160113/https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43.

Martin Berglund, Willem Bester, and Brink van der Merwe. 2021. 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

Preston Briggs and Linda Torczon. 1993. An Efficient Representation for Sparse Sets. *ACM Lett. Program. Lang. Syst.* 2, 1–4 (mar 1993), 59–69. https://doi.org/10.1145/176454.176484

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

Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Ultes Stefan, Ramadan Osman, and Milica Gašić. 2018. MultiWOZ - A Large-Scale Multi-Domain Wizard-of-Oz Dataset for Task-Oriented Dialogue Modelling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://doi.org/10.18653/v1/D18-1547

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

CVC4. 2020. https://github.com/CVC4/CVC4.

Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. *ACM SIGPLAN Notices – POPL'14* 49, 1 (2014), 541–553. https://doi.org/10.1145/2535838.2535849

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 *TACAS'08 (LNCS)*. Springer, 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) (LNCS, Vol. 3142)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, 618–629. https://doi.org/10.1007/978-3-540-27836-8_53

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

GNU. 2020. grep. https://www.gnu.org/software/grep/.

Google. 2021. RE2. https://github.com/google/re2.

John Graham-Cumming. 2019. *Details of the Cloudflare outage on July 2, 2019*. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

Zoltan Herczeg. 2015. Performance comparison of regular expression engines. https://zherczeg.github.io/sljit/regex_perf.html.

Pieter Hooimeijer and Margus Veanes. 2011. An Evaluation of Automata Algorithms for String Analysis. In *VMCAI'11 (LNCS, Vol. 6538)*. Springer, 248–262. https://doi.org/10.1007/978-3-642-18275-4_18

Intel Co. 2021. Hyperscan. https://github.com/intel/

Mario Juárez. 2020. Languages Regex Benchmark. https://github.com/mariomka/regex-benchmark.

Matthias Keil and Peter Thiemann. 2014. Symbolic Solving of Extended Regular Expression Inequalities. In *FSTTCS'14 (LIPIcs)*. 175–186. https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175

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

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

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 (LNCS, Vol. 9322).* Springer, 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 *PLDI'19.* ACM, 425–438. https://doi.org/10.1145/3314221.3314645

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

Microsoft. 2021b. .NET Performance. https://github.com/dotnet/performance.

Microsoft. 2021c. 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.

Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023a. *Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics.* Technical Report MSR-TR-2023-15. Microsoft Research. https://www.microsoft.com/en-us/research/publication/derivative-based-nonbacktracking-real-world-regex-matching-with-backtracking-semantics/

Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023b. Artifact for "Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics". https://doi.org/10.5281/zenodo.7709500.

OWASP. 2020. Regular expression Denial of Service - ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS

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

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

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

Henry Spencer. 1994. Software Solutions in C. Academic Press Professional, Inc., San Diego, CA, USA, Chapter A Regular-expression Matcher, 35–71. http://dl.acm.org/citation.cfm?id=156626.184689

Stack Exchange. 2016. *Outage Postmortem.* http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *PLDI'21.* ACM, 620–635. https://doi.org/10.1145/3453483.3454066

L. J. Stockmeyer and A. R. Meyer. 1973. Word Problems Requiring Exponential Time(Preliminary Report). In *Fifth Annual ACM Symposium on Theory of Computing, STOC'73.* ACM, 1–9. https://doi.org/10.1145/800125.804029

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

K. Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

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. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 218 (Nov. 2020). https://doi.org/10.1145/3428286

Ola Wingbrant. 2019. Regular languages, derivatives and finite automata. https://doi.org/10.48550/ARXIV.1907.13577

Received 2022-11-10; accepted 2023-03-31