Formalising Boost POSIX Regular Expression Matching

Martin Berglund$^{1[0000-0002-3692-6994]}$, Willem Bester$^{2[0000-0002-0016-7623]}$, and Brink van der Merwe$^{2[0000-0001-5010-9934]}$

[1] Department of Information Science, Centre for AI Research (CSIR), University of Stellenbosch, Private Bag X1, Matieland 7602, South Africa
`pmberglund@sun.ac.za`

[2] Computer Science, Department of Mathematical Sciences, University of Stellenbosch, Private Bag X1, Matieland 7602, South Africa
`{whkbester,abvdm}@cs.sun.ac.za`
`http://www.cs.sun.ac.za/~{whkbester,abvdm}`

Abstract. Whereas Perl-compatible regular expression matchers typically exhibit some variation of leftmost-greedy semantics, those conforming to the POSIX standard are prescribed leftmost-longest semantics. However, the POSIX standard leaves some room for interpretation, and Fowler and Kuklewicz have done experimental work to confirm differences between various POSIX matchers. The Boost library has an interesting take on the POSIX standard, where it maximises the leftmost match not with respect to subexpressions of the regular expression pattern, but rather, with respect to capturing groups. In our work, we provide the first formalisation of Boost semantics, and we analyse the complexity of regular expression matching when using Boost semantics.

Keywords: Regular expression matching · POSIX · Boost

1 Introduction

In his "casual stroll across the regex landscape", Friedl [9] identifies two regular expression flavours with which the typical user must become acquainted, namely, those that are Perl-compatible, called PCRE [1], and those that follow the POSIX standard [4]. PCRE matchers follow a leftmost-greedy disambiguation policy, but POSIX matchers favour the leftmost-longest match. These flavours differ not only in terms of their syntax, but also, more crucially, in terms of their matching semantics. The latter is particularly noteworthy where *ambiguity* enters the picture, which is to say, where an input string "can be matched in more than one way" [24].

Through the standardisation of languages such as Perl, with native support for regular expressions, and libraries such as those defined by POSIX, new features became available, but initially, without much attention to the theoretic investigation of issues such as ambiguity. If, after the publication of Thompson's famous construction [25] in 1968, regular expressions were viewed as the perfect marriage between theory and practice, then by the 1980s, the state of the art and the state of the theory had parted ways. Since the 1990s, when the growth of the World Wide Web led to an interest in parsing for markup languages [7], the academic community has responded with vigour, as various features and flavours of regular expressions were studied and formalised (for example, see Kearns [12]).

To this, we now add the following contributions: We extend regular expressions to *capturing regular expressions,* which define forest languages instead of the usual string languages, in an effort to place the notion of parsing, as found in implementations, on a secure theoretical footing. We go on to provide a series of varied instructive examples, highlighting the similarities and differences between standards, implementations, and our formalisation of matching semantics. Finally, we formalise the matching semantics and investigate the matching complexity of the Boost variant of POSIX regular expressions, which has not been attempted before.

1.1 Related Work

In the documentation to their system regular expression libraries, which claim POSIX-compatibility, BSD Unices like OpenBSD [3] point to the implementations of Henry Spencer [11] as foundation. Recent versions of macOS [2], also in the BSD family, cite in addition the TRE library [18] by Laurikari, who used the notion of tagged automata to formalise full matching with submatch addressing for POSIX [17, 20]. Subsequently, Kuklewicz took issue with Laurikari's claims to efficiency and correctness [14, 16], resulting in the Regex-TDFA library for Haskell [13], which passes an extensive test suite [15] based on Fowler's original version [8].

Okui and Suzuki [22] formalised leftmost-longest semantics in terms of a strict order based on the yield lengths of parse tree nodes, ordered lexicographically by position strings. In contrast, Sulzmann and Lu [23], inspired by Frisch and Cardelli's work on the formalisation of greedy matching semantics [10], used a different approach, that of viewing regular expressions as types, and then treating the parse trees as values of some regular expression type, in the process also establishing a strict order of various parse trees with respect to a particular regular expression.

1.2 Paper Outline

The paper outline is as follows. In the next section, we state some definitions and properties of regular expressions and formal languages. Then, in Section 3, we present detailed examples, which serve to illustrate some of the issues and complexities of POSIX and Boost matching. In Section 4, we give a formal statement of Boost matching semantics, and also discuss the complexity of doing regular expression macthing with Boost. We then present some experimental results, and we end with concluding remarks.

2 Preliminaries

Denote by $\mathbb{N}$ the set of positive integers, let $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, and as usual, let $\leq$ ($<$) be the natural (strict) order on $\mathbb{N}_0$. We use $\perp$ and $\top$ to indicate undefined values, and assume that $\perp < n < \top$ for all $n \in \mathbb{N}_0$. Let Σ be a finite alphabet, denote by ε the empty string, and let Σ_ε be the set $\Sigma \cup \{\varepsilon\}$. For any string w over Σ with $\Sigma' \subseteq \Sigma$, let $\pi_{\Sigma'}(w)$ be the maximal subsequence of w that contains only symbols from Σ', and let $|w|$ be the length of w, which is to say, the number of symbols (from Σ) in w. In particular, $|\varepsilon| = 0$.

We denote the empty set by $\varnothing$. For any set A, let $\mathcal{P}(A)$ be power set of A. If $g : A \to B$ is a function, we also use g to denote the function from $\mathcal{P}(A)$ to $\mathcal{P}(B)$ defined by mapping $A' \subseteq A$ to $\{g(a) \mid a \in A'\} \subseteq B$.

Next we define the notion of forests, that is, the concatenation of trees. This is used to formalise the concept of which substring of an input string is matched (or captured) by which subexpression of a given regular expression.

Definition 1. *The* set of forests *over Σ and the index set I, denoted by $\mathcal{F}(\Sigma, I)$, is defined inductively as follows. We assume $\Sigma_\varepsilon \subseteq \mathcal{F}(F, I)$, and for $f_1, f_2 \in \mathcal{F}(\Sigma, I)$ and $i \in I$, we have that $f_1 f_2$ (that is, the concatenation of f_1 and f_2) and $[_i f]_i$ are elements in $\mathcal{F}(\Sigma, I)$. A* forest language *L over Σ and I is a subset of $\mathcal{F}(\Sigma, I)$, whereas a* string language *over Σ is a subset of Σ^*.*

In the sequel, we shall assume $I \subseteq \mathbb{N}_0$. Note that $\mathcal{F}(\Sigma, I)$ properly contains all strings over Σ if I is non-empty, and is otherwise precisely equal to the set of strings over Σ. A forest can be considered either as being a string over $\Sigma \cup \{[_i,]_i \mid i \in I\}$ or as a concatenation of ordered unranked trees over $\Sigma_\varepsilon \cup I$, where $[_i f]_i$ is a tree with root node labelled by i, having the forest f of zero or more trees as descendants.

Since standard (theoretical) regular expressions do not formalise the parsing aspects related to regular expression matching, which is essential for our discussion on Boost and POSIX in general, we next extend the standard definition of regular expressions in a way suitable for our purpose.

Definition 2. *The set of* capturing regular expressions *over a finite alphabet Σ and an index set I, denoted by $\mathcal{R}(\Sigma, I)$, is defined inductively as follows:*

1. *the* empty language expression *$\varnothing$;*
2. *the* empty string expression *ε;*
3. *the* symbols *$a \in \Sigma$;*
4. *the* concatenation*, also known as the* product*, $(r_0 \cdot r_1)$ of $r_0, r_1 \in \mathcal{R}(\Sigma, I)$;*
5. *the* union*, also known as the* sum *or* alternation*, $(r_0 + r_1)$ of $r_0, r_1 \in \mathcal{R}(\Sigma, I)$;*
6. *the* (Kleene) closure *(r^*) of $r \in \mathcal{R}(\Sigma, I)$; and*
7. *the* capture group *$(_i r)_i$ for $r \in \mathcal{R}(\Sigma, I)$ and $i \in I$.*

Items 1 to 3 in Definition 2 are called the *atoms* of regular expressions. We assume the alphabet Σ is disjoint with the symbols used to define the operations in items 4 to 7. The parentheses around the expressions of items 4 to 6 are optional, and if some are left out, the operator precedence, from high to low, is (i) closure, (ii) concatenation, and (iii) union. In addition, we assume that any concatenation $r_0 \cdot r_1$ may be written by juxtaposition as $r_0 r_1$.

Remark 1. For a Boost capturing regular expression r, it is assumed that all opening parentheses are indexed from 1 onward, from left to right, with the corresponding closing parenthesis indexed correspondingly. A Boost matcher will also replace r by $({}_0 r)_0$, before starting the matching procedure. The Boost capturing regular expressions thus form a proper subset of $\mathcal{R}(\Sigma, I)$ (if $I \neq \varnothing$).

Next we define syntactic shortcuts used in practice and also in some of our examples.

Definition 3. *For $r \in \mathcal{R}(\Sigma, I)$, define the following additional iterative operators:*

1. *the* duplication $(r^{m,n})$ *abbreviates*
$$\big(\underbrace{r \cdots r}_{m\ times}\underbrace{(r+\varepsilon)\cdots(r+\varepsilon)}_{n-m\ times}\big)$$
for $m, n \in \mathbb{N}_0$ with $m \leq n$;
2. *the* option $(r?)$ *abbreviates $(r + \varepsilon)$;*
3. *the* positive closure (r^+) *abbreviates (rr^*).*

The second parameter of duplication is optional, and here we distinguish between two cases, namely, r^m being equivalent to $r^{m,m}$, and $r^{m,}$ being equivalent to $r^m r^*$. Again, we assume the set of symbols used to define the shortcut iteration operators in Definition 3 is disjoint from the underlying alphabet Σ.

Definition 4. *We define the following operations on forest, and thus also on string languages:*

1. *the* concatenation $L_0 \cdot L_1 = L_0 L_1 = \{w_0 w_1 \mid w_0 \in L_0$ *and* $w_1 \in L_1\}$ *for two languages L_0 and L_1;*
2. *the* union $L_0 \cup L_1 = \{w \mid w \in L_0$ *or* $w \in L_1\}$ *for two languages L_0 and L_1;*
3. *the* nth power *of the language L, where $n \in \mathbb{N}_0$, is*
$$L^n = \begin{cases} \{\varepsilon\} & \textit{if } n = 0, \\ L \cdot L^{n-1} & \textit{otherwise; and} \end{cases}$$
4. *the* closure $L^* = \bigcup_{n \in \mathbb{N}_0} L^n$ *of the language L.*

Finally, we are ready to define the forest languages described by the expressions in $\mathcal{R}(\Sigma, I)$.

Definition 5. *The* forest language *described by $r \in \mathcal{R}(\Sigma, I)$, and denoted by $\mathcal{L}(r)$, is defined inductively as follows: $\mathcal{L}(\varnothing) = \varnothing$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{\mathtt{a}\}$, $\mathcal{L}(r_0 \cdot r_1) = \mathcal{L}(r_0) \cdot \mathcal{L}(r_1)$, $\mathcal{L}(r_0 + r_1) = \mathcal{L}(r_0) \cup \mathcal{L}(r_1)$, and $\mathcal{L}(r^*) = \mathcal{L}(r)^*$, where $r, r_0, r_1 \in \mathcal{R}(\Sigma, I)$.*

Definition 6. *We define the string language described by $r \in \mathcal{R}(\Sigma, I)$ to be the set $\pi_\Sigma(\mathcal{L}(r))$.*

Remark 2. Note if r' is the regular expression obtained from $r \in \mathcal{R}(\Sigma, I)$ by replacing the capturing parenthesis with normal parenthesis, then $\mathcal{L}(r') = \pi_\Sigma(\mathcal{L}(r))$.

Definition 7. *A* forest disambiguation policy *$D(\Sigma, I)$ for $\mathcal{R}(\Sigma, I)$ is a set of functions $\{d_r : \mathcal{L}(r) \to \mathcal{L}(r) \mid r \in \mathcal{R}(\Sigma, I)\}$, such that for $f, f' \in \mathcal{L}(r)$, we have $\pi_\Sigma(d_r(f)) = \pi_\Sigma(f)$ and $\pi_\Sigma(f) = \pi_\Sigma(f')$ implies $d_r(f) = d_r(f')$.*

If for $f \in \mathcal{L}(r)$, we have $\pi_\Sigma(f) = w$, we refer to f as a *parse forest* for w. We can intuitively think of a regular expression matcher as not only deciding if a given string w is in $\pi_\Sigma(\mathcal{L}(r))$ or not, but also, if $w \in \pi_\Sigma(\mathcal{L}(r))$, to have a specification of which parse forest to associate with w, amongst the potentially many possible parse forests for w. However, for efficiency reasons, greedy, Boost, and other POSIX matchers do not have a forest disambiguation policy in general, but rather a disambiguation policy on matching information derived from parsing forests. In general, the corresponding forests can not uniquely be reconstructed from this derived information. In Section 4, we provide precise details on the structure of this derived information for Boost.

Example 1. Consider matching $w =$ "ab" with $E =$ /a?(ab)?b?/. The Okui–Suzuki disambiguation policy [22], modified here in terms of notation to align closer with our approach, first replaces E by an (almost) fully parenthesised expression $E' = /(_1\mathtt{a}?)_1(_2(_3(\mathtt{ab})?)_3(_4\mathtt{b}?)_4)_2/$, obtained by assuming concatenation is right-associative, and by numbering the opening parenthesis, in order, from left to right in E'. The following two forests are candidates for a full match:

$$t_0 = [_1\mathtt{a}]_1[_2[_3]_3[_4\mathtt{b}]_4]_2 \quad \text{and} \quad t_1 = [_1]_1[_2[_3\mathtt{ab}]_3[_4]_4]_2.$$

Using the natural order on $\mathbb{N}_0$ for the capture indices, this yields the two vectors $\langle 1, 1, 0, 1\rangle$ and $\langle 0, 2, 2, 0\rangle$ of lengths of captures for t_0 and t_1, respectively, by which t_0 is chosen when using a lexicographic order on these two vectors.

3 Examples

At this point, it is instructive to turn to some detailed examples. In particular, because POSIX [4] is a software engineering standard, its functional specification for regular expressions is written in English, without mathematical formalisms. As such, some parts of the specification might be open to multiple interpretations. Fowler [8] has argued that the specification is "surprisingly cavalier with terminology", and it is the implications of different readings of the standard that we now examine in detail.

Remark 3. An additional concern is that POSIX defines two different specifications for regular expressions, namely, *Basic Regular Expressions* (BREs) and

Extended Regular Expressions (EREs), which although they share many similarities, especially in the specifications of atoms, are incompatible. For example, BREs supports backreferences, "operations which bind a substring to a variable, allowing it to be matched again verbatim", a feature that is non-regular and that makes matching NP-complete [6]. EREs do not support backreferences, removes the need for group and iteration bounds delimiters to be escaped, and supports the union of subexpressions, which BREs do not. Therefore—and it also bears mentioning that some implementers of POSIX regular expressions consider BREs obsolete [2, 19]—we focus exclusively on EREs.

Remark 4. The specification language for regular expression patterns in POSIX differs from our Definitions 2 and 3. In particular, for the union $r + s$ of two expressions r and s, we would write `r|s` in POSIX, and the iterates r^*, r^m, $r^{m,}$, and $r^{m,n}$ are written as `r*`, `r{m}`, `r{m,}`, and `r{m,n}`, respectively. Until Section 5, we use the notation established in Section 2 exclusively.

Remark 5. In the POSIX standard for EREs [4, §9.4], each pair of parentheses, unless escaped or included in a *bracket expression*—where the escape sequences "`\(`" and "`\)`" match the literal opening and closing parenthesis characters, and whereas the bracket expression "`[(ab)]`" matches any one of the literal characters "`(`", "`a`", "`b`", or "`)`"—*always* automatically defines a *group*: They do not match literal occurrences of parentheses in the input string, but serve to override the default operator precedence, and also, allow the matcher to report which substring was *captured* (or matched) by which group. Consistent with Remark 1, groups are identified by positive integers, and internally, the matcher automatically numbers the pairs of parentheses from left to right, starting at 1; in addition, the entire regular expression is numbered as group 0. In our examples, we write group numbers explicitly, and depending on what is convenient, we either state group 0 explicitly, or do not indicate it at all.

Example 2. Consider matching the input string $w =$ "$\mathtt{aba}$" with the regular expression $E_0 = /({}_1\mathtt{ab} + \mathtt{ba} + \mathtt{a})_1^*/$. Matching w with E_0 is ambiguous, because two different forests in $\mathcal{L}(E_0)$ that correspond to a full match, where the entire input string is matched by the regular expression, are possible, namely, $f_0 = [_1\mathtt{ab}]_1[_1\mathtt{b}]_1$ and $f_1 = [_1\mathtt{a}]_1[_1\mathtt{ba}]_1$. The forest f_0 means the matcher used the $/\mathtt{ab}/$ subexpression of E_0 for the first iteration of the star, and the $/\mathtt{a}/$ subexpression for the second, whereas the forest f_1 means the matcher used the $/\mathtt{a}/$ subexpression of E_0 for the first iteration of the star, and the $/\mathtt{ba}/$ subexpression for the second. The bracketed subforests in f_0 and f_1 indicate which substrings were matched by group 1 during consecutive iterations. For E_0, both leftmost-greedy and leftmost-longest matchers will use the forest f_0.

However, now consider matching $w =$ "$\mathtt{aba}$" with $E_1 = /({}_1\mathtt{a} + \mathtt{ab} + \mathtt{ba})_1^*/$, and note that E_1 defines the same language as E_0, but that the order of the subexpressions inside the star has changed. Again, the two forests f_0 and f_1 correspond to a full match, but now, a leftmost-greedy matcher will use f_1, whereas a leftmost-longest matcher will still use f_0. □

Remark 6. It should be noted that matchers typically do not report forests, but only substrings matched (or captured) by some subexpressions, and specifically, in the case of a subexpression s^*, most matchers only report the last capture by the subexpression s. Thus it is in fact only the simplicity of the previous example that makes it possible to reverse engineer the parse forests.

Intuitively, whenever more than one match is possible for a particular subexpression, a greedy matcher will return the first match with respect to the order in which this subexpression's subexpressions are written in the regular expression. This is to say, when a subexpression admits several choices for matching the same substring of the input string, the leftmost choice will prevail. In contrast, a leftmost-longest matcher must seemingly consider all possible matches for that subexpression, starting as early as possible in the input string, where "early" means "leftmost", unless this choice causes the entire match to fail; if the leftmost policy is not enough to distinguish between two submatches, we give preference to longer submatches.

Example 3. Again, consider the input string w and the regular expressions E_0 and E_1 from Example 2, but now let us examine what happens when these examples are run on Haskell's Regex-TDFA [13] and Boost [21], which both claim to be POSIX-compliant—but see Remark 7 below—and hence, where we expect both to return $[_1\texttt{ab}]_1[_1\texttt{a}]_1$ for E_0 as well as E_1. However, whereas Regex-TDFA returns $[_1\texttt{ab}]_1[_1\texttt{a}]_1$ with E_0 and E_1, and thus performs as expected, Boost returns $[_1\texttt{a}]_1[_1\texttt{ba}]_1$, again for both E_0 and E_1. The disparity can be understood by realising that Regex-TDFA maximises all captures by a starred subexpression, from left to right, based on first considering the leftmost and secondly the length criteria, although it only reports the last match, while Boost does the maximisation only on the last submatch. □

A case can be made that Boost is in fact POSIX-compliant, albeit with a different reading than, for example, Regex-TDFA of the salient points of the POSIX matching policy [4, §9.1]:

> The search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found, where "first" is defined to mean "begins earliest in the string". If the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence is matched.... Consistent with the whole match being the longest of the leftmost matches, each subpattern[3] from left to right shall match the longest possible string.

[3] Fowler [8] identifies the terms "subpattern" and "subexpression" as particular targets of abuse in the POSIX standard, especially since they are "central to the description of the matching algorithm". He goes on to note that, whereas "subpattern" is used but once, "subexpression" is used 70 times and always appears in the context of grouping.

Posix therefore requires full matching with *submatch addressing*, where "the position and extent of the substrings matched by given subexpressions must be provided" [20]. Contrast this with the classic automata-theoretic approach, where a matcher simply determines whether the entire input string was matched by the regular expression or not.

Although Boost applies a leftmost-longest policy, it considers what its documentation calls "marked subexpressions" [21], instead of arbitrary subexpressions, such that we now render the last quoted sentence as: "Consistent with the whole match being the longest of the leftmost matches, each *marked* subpattern from left to right shall match the longest possible string." Thus Boost applies its leftmost-longest disambiguation policy not by maximising arbitrary subexpressions of a regular expression, but instead, by maximising marked subexpressions—those subexpressions surrounded by parentheses.

Remark 7. Boost also supports PCRE syntax and semantics, which is, in fact, its default mode of operation. (For detail, see the discussion after Remark 9 on page 11.) When we refer to Boost in this and following sections, we exclusively mean Boost in its POSIX mode of operation.

Example 4. Consider matching w = "aa" with $E_2 = /({}_0\mathtt{a}^*({}_1\mathtt{a}^*)_1)_0/$ and $E_3 = /({}_0({}_1\mathtt{a}^*)_1({}_2\mathtt{a}^*)_2)_0/$. Although both expressions define the same language, the first $/\mathtt{a}^*/$ subexpression is a group in E_3, but not in E_2. For E_2, the forests $f_2 = [_0\mathtt{aa}[_1]_1]_0$, $f_3 = [_0\mathtt{a}[_1\mathtt{a}]_1]_0$, and $f_4 = [_0[_1\mathtt{aa}]_1]_0$ correspond to matching the entire input string w, and for E_3, we have the forests $f_5 = [_0[_1\mathtt{aa}]_1[_2]_2]_0$, $f_6 = [_0[_1\mathtt{a}]_1[_2\mathtt{a}]_2]_0$, and $f_7 = [_0[_1]_1[_2\mathtt{aa}]_2]_0$.

We consider matching with E_3 first: Boost and Regex-TDFA both prefer f_5, since all non-atomic subexpressions are parenthesised, and both matchers simply maximise the lengths of the substrings matched by the groups from left to right. However, for E_2, Regex-TDFA uses f_2, because this matcher maximises the lengths of *all* subexpressions from left to right, regardless of whether a subexpression is marked as a group. Boost, on the other hand, maximises groups—here, first with respect to group 0, and then with respect to group 1 (which is contained in group 0). Hence, Boost prefers f_4 to f_2. □

Since unescaped and unbracketed parentheses always define groups, it does not matter that parentheses might have been necessitated by issues of operator precedence: Unlike the typical PCRE matcher or the Java regular expression matcher, which support non-capturing groups—for which parentheses has no other influence, save possibly changing how that abstract syntax tree of the regular expression is constructed—the user of POSIX-compliant matchers, including Boost, has no choice in the matter of capturing groups. However, in our theoretical model of capturing regular expressions, we do in fact allow both capturing and non-capturing groups.

Example 5. Fowler gives the example $E_4 = /\mathtt{a}?({}_1\mathtt{ab})_1?\mathtt{b}?/$. Arguably, here the parentheses serve no other purpose except to delimit the subexpression to be matched by the second option operator. The two forests $f_8 = [_0\mathtt{ab}]_0$ and $f_9 =$

$[_0[_1\texttt{ab}]_1]_0$ correspond to a full match of the input string w = "ab". Here, f_8 represents the matcher using the first /a/ and the last /b/ subexpressions, and f_9 the case where the second option, of the /ab/ subexpression, is used. As is to be expected for leftmost-longest semantics, Regex-TDFA returns f_8, but Boost, since it will maximise the only marked subexpression, returns f_9. Contrast this with matching the same input string w with $E_5 = /(_1\texttt{a?})_1(_2\texttt{ab})_2?(_3\texttt{b?})_3/$, where both Regex-TDFA and Boost will prefer the forest $f_{10} = [_0[_1\texttt{a}]_1[_3\texttt{b}]_3]_0$ to the forest $f_{11} = [_0[_1]_1[_2\texttt{ab}]_2[_3]_3]_0$. □

Example 6. Consider the regular expression $E_6 = /(_1\texttt{a} + (_2\texttt{b}^*)_2)_1^*/$. Since all non-atomic subexpressions are parenthesised, for a given input string, matching with both Boost and Regex-TDFA succeed on the same forest. For the input strings w_1 = "abb", w_2 = "abba", and w_3 = "abbab", the respective forests $f_{12} = [_0[_1\texttt{a}]_1[_1[_2\texttt{bb}]_2]_1]_0$, $f_{13} = [_0[_1\texttt{a}]_1[_1[_2\texttt{bb}]_2]_1[_1\texttt{a}]_1]_0$, and $f_{14} = [_0[_1\texttt{a}]_1[_1[_2\texttt{bb}]_2]_1[_1\texttt{a}]_1[_1[_2\texttt{b}]_2]_1]_0$ are preferred.

The way Boost reports the result, however, differs from Regex-TDFA. Using Fowler's format of reporting [8], we express the output (with grouping) of running a matcher as a sequence of pairs, one for each group, starting at 0 for the entire regular expression. The first element of a pair gives the start index of the substring of the input that was matched by the group subexpression, and the second element is the end index plus one of the substring. For f_{13}, Boost reports `(0,4)(3,4)(1,3)`, but Regex-TDFA reports `(0,4),(3,4),(?,?)`, where `(?,?)` means the group subexpression did not participate in the match. □

How the information is reported depends on the implementer's reading of the matching function's specification [4, "System interfaces—`regcomp`"], which we summarise as: (1) If a subexpression is not contained within another subexpression, then if the subexpression participated in a match multiple times, the last such match must be reported, or else, if it did not participate in a match, then it must be reported as non-participating; (2) if a subexpression is contained within another subexpression, and the outer subexpression participated in a match, then the match or non-match of the inner subexpression must be reported according to Rule (1), but with respect to the substring matched by the outer subexpression and not the entire input string. "Participation" is defined negatively: A subexpression *does not participate* in a match when one of the choices in a union is not taken, or when the empty string is matched with an iterative operator by matching zero times with the associated subexpression.

Essentially, Boost has elected to ignore Rule 2. Since group 2 is contained in group 1, and for the last match (by Rule 1) of group 1, group 2 did not participate in the match (by Rule 2), Regex-TDFA reported group 2 as `(?,?)` by Rule 2. This can be seen in the forest f_{13}, where the last subforest for group 1 does not contain a subforest for group 2. Boost, on the other hand, simply returns the last match information, regardless of whether one group is contained in another.

Incidentally, from the context in which the unqualified term "subexpression" is used in the POSIX specification for reporting submatches, it is clear that this term actually refers to parenthesised subexpressions, which is to say, groups.

Elsewhere [4, §9.4], the same term can refer to arbitrary or parenthesised subexpressions, the latter of which is sometimes referred to by "grouping". These inconsistencies illustrate Fowler's critical stance on the standard.

Example 7. To see how reporting differs for an empty match as opposed to a non-participating subexpression, consider $E_7 = /(_0(_1\mathtt{a} + (_2(_3\mathtt{b})^*_3)_2)^*_1)_0/$. Both Boost and Regex-TDFA prefer the forest $f_{15} = [_0[_1\mathtt{a}[_2[_3\mathtt{b}]_3[_3\mathtt{b}]_3]_2]_1[_1\mathtt{a}[_2]_2]_1]_0$, but Regex-TDFA reports `(0,4),(3,4),(4,4),(?,?)`, whereas Boost instead reports `(0,4),(3,4),(4,4),(2,3)`. Note that both report an empty match for group 2 (by having the same index for the start and end), since the iterative star operator is applied to group 3 inside group 2, and therefore, since the last match for group 2 is empty, group 3 did not participate in this match. Yet, although group 3 is inside group 2, Boost still reports the last participating match of group 3. □

4 Boost semantics and matching algorithm

We start this section by first providing three preliminary definitions, which is then used to formalise, in Definition 11, Boost semantics.

Definition 8. *The* capture history *for forests is the function $C : \mathcal{F}(\Sigma, I) \times I \to \mathbb{N}_0 \times \mathbb{N}_0$, defined as follows. Let $f \in \mathcal{F}(\Sigma, I)$, $j \in I$, and*

$$f(j) = \pi_{\Sigma \cup \{[_j,]_j\}}(f) = w_0[_j w_1]_j \cdots w_{2i}[_j w_{2i+1}]_j w_{2(i+1)} \cdots [_j w_{2k-1}]_j w_{2k}$$

where $w_i \in \Sigma^*$. Then

$$C(f, j) = \begin{cases} \{(|w_0 \cdots w_{2i}|, |w_{2i+1}|) \mid 0 \le i < k\} & \text{if } [_j \text{ appears in } f; \\ \varnothing & \text{otherwise.} \end{cases}$$

We assume that the tuples in $C(f, j)$ are always sorted by increasing first index. Also, $C_{\text{last}}(f, j)$ denotes the tuple in $C(f, j)$ with largest first index if $C(f, j)$ is non-empty, and $C_{\text{last}}(f, j) = (\top, \bot)$ otherwise.

Intuitively, Definition 5 allows us to express how substrings of an input string are captured by the capture groups of a regular expression, which is accomplished by decorating the input string with pairs of indexed brackets to delimit the substrings thus captured as matching proceeds. In turn, Definition 8 allows us to extract the capture history for a particular group, which yields a (possibly empty) set of pairs, where each pair gives the start index and the length of the captured substring. Note that we opted to record starting indices and length for captures in our formalisation in the previous definition, instead of starting and ending indices as is done typically by implementations.

Example 8. To illustrate the capture history, we revisit Example 5. For matching the input string w = "ab" by $E_5 = /(_1\mathtt{a}?)_1(_2\mathtt{ab})_2?(_3\mathtt{b}?)_3/$, we consider the forests $f_{10} = [_0[_1\mathtt{a}]_1[_3\mathtt{b}]_3]_0$ and $f_{11} = [_0[_1]_1[_2\mathtt{ab}]_2[_3]_3]_0$. They yield, first for f_{10}: $C(f_{10}, 0) = C([_0\texttt{ab}]_0) = \{(0, 2)\}$, $C(f_{10}, 1) = C([_1\texttt{a}]_1\texttt{b}, 1) = \{(0, 1)\}$, $C(f_{10}, 2) = C(\texttt{ab}, 2) = \{(\top, \bot)\}$, $C(f_{10}, 3) = C(\texttt{a}[_3\texttt{b}]_3, 3) = \{(1, 1)\}$; and then for f_{11}: $C(f_{11}, 0) = C([_0\texttt{ab}]_0) = \{(0, 2)\}$, $C(f_{11}, 1) = C([_1]_1\texttt{ab}, 1) = \{(0, 0)\}$, $C(f_{11}, 2) = C([_2\texttt{ab}]_2, 2) = \{(0, 2)\}$, and $C(f_{11}, 3) = C(\texttt{ab}[_3]_3, 3) = \{(2, 0)\}$. Note the difference between the empty captures such as $C(f_{11}, 1)$ and $C(f_{11}, 3)$, and a capture history in which a particular subexpression did not participate, such as $C(f_{10}, 2)$. □

Definition 9. *The* final capture history *for $f \in \mathcal{F}(\Sigma, I)$, denoted as $C_{\mathit{fin}}(f)$, is the set $\{(j, C_{\mathit{last}}(f, j)) \mid j \in I\}$.*

Remark 8. In the sequel, we abuse notation somewhat, and we write $C_{\text{fin}}(f)$ as a set of triplets instead of as a set of ordered pairs (of which each second element is also a pair).

Example 9. To illustrate the final capture history, we use the forest $f_{14} = [_0[_1\texttt{a}]_1[_1[_2\texttt{bb}]_2]_1[_1\texttt{a}]_1[_1[_2\texttt{b}]_2]_1]_0$ from Example 6. Recall, for this forest, we matched the input string $w_3 =$ "abbab" by the regular expression $E_6 = /(_1\texttt{a} + (_2\texttt{b}^*)_2)_1^*/$. From f_{14}, we extract the capture histories

$$C(f_{14}, 0) = C([_0\texttt{abbab}]_0, 0) = \{(0, 5)\},$$
$$C(f_{14}, 1) = C([_1\texttt{a}]_1[_1\texttt{bb}]_1[_1\texttt{a}]_1[_1\texttt{b}]_1, 1) = \{(0, 1), (1, 2), (3, 1), (4, 1)\}, \text{and}$$
$$C(f_{14}, 2) = C(\texttt{a}[_2\texttt{bb}]_2\texttt{a}[_2\texttt{b}]_2, 2) = \{(1, 2), (4, 1)\}.$$

Therefore, $C_{\text{fin}}(f_{14}) = \{(0, 0, 5), (1, 4, 1), (2, 4, 1)\}$. □

Definition 10. *We define the* Boost partial order*, denoted as $\prec_B$, on $\{C_{\mathit{fin}}(f) \mid f \in \mathcal{F}(\Sigma, I)\}$ as follows. Assume $\pi_\Sigma(f_1) = \pi_\Sigma(f_2)$, then $C_{\mathit{fin}}(f_1) \prec_B C_{\mathit{fin}}(f_2)$ if for the smallest element $j \in I$ such that $(j, s_1, \ell_1) \neq (j, s_2, \ell_2)$, where $(j, s_i, \ell_i) \in C_{\mathit{fin}}(f_i)$, we have $s_2 < s_1$, or $s_1 = s_2$ but $\ell_1 < \ell_2$.*

Definition 11. *For $r \in \mathcal{R}(\Sigma, I)$ and $w \in \pi_\Sigma(\mathcal{L}(r))$, the* Boost captures *of matching w with r, denoted as $B(r, w)$, is defined to be the largest element in $\{C_{\mathit{fin}}(f) \mid f \in \mathcal{L}(r), \pi_\Sigma(f) = w\}$ determined by $\prec_B$.*

Remark 9. It should be noted that $\prec_B$ is a total order on the finite set $\{C_{\text{fin}}(f) \mid f \in \mathcal{L}(r), \pi_\Sigma(f) = w\}$ used in the previous definition, and thus $B(r, w)$ is well-defined.

Example 10. To illustrate Boost partial order and captures, we continue Example 5. We match $w =$ "ab" with $E_4 = /\texttt{a}?(_1\texttt{ab})_1?\texttt{b}?/$, and we consider the forests $f_8 = [_0\texttt{ab}]_0$ and $f_9 = [_0[_1\texttt{ab}]_1]_0$. By Definition 8, we have the capture histories $C(f_8, 0) = \{(0, 2)\}$, $C(f_8, 1) = \varnothing$, $C(f_9, 0) = \{(0, 2)\}$, and $C(f_9, 1) = \{(0, 2)\}$, whence by Definition 9, $C_{\text{fin}}(f_8) = \{(0, 0, 2), (1, \top, \bot)\}$ and $C_{\text{fin}}(f_9) = \{(0, 0, 2), (1, 0, 2)\}$. At $j = 1$, we find $s_8 = \top$ and $s_9 = 0$, so that $s_9 < s_8$, and therefore, by Definition 10, $f_8 \prec_B f_9$. Finally, by Definition 11, $B(E_4, \text{"ab"}) = f_9$.

For matching w with $E_5 = /(_1\texttt{a}?)_1(_2\texttt{ab})_2?(_3\texttt{b}?)_3/$, and for the forests $f_{10} = [_0[_1\texttt{a}]_1[_3\texttt{b}]_3]_0$ and $f_{11} = [_0[_1]_1[_2\texttt{ab}]_2[_3]_3]_0$, we calculate, by way of Example 8,

$$C_{\text{fin}}(f_{10}) = \{(0,0,2),(1,0,1),(2,\top,\bot),(3,1,1)\} \text{ and}$$
$$C_{\text{fin}}(f_{11}) = \{(0,0,2),(1,0,0),(2,0,2),(3,2,0)\}.$$

At $j = 1$, we find $s_{10} = s_{11} = 0$, $\ell_{10} = 1$, and $\ell_{11} = 0$, so that $\ell_{11} < \ell_{10}$. Therefore, $f_{11} \prec_B f_{10}$, and $B(E_5, w) = f_{10}$. □

The actual implementation of POSIX matching in Boost is implemented in a very straightforward way, in that it is a small modification of *another* matching engine. Boost contains a very complete implementation of PCRE/Java-style semantics, implemented by depth-first backtracking search on what is in effect an automaton constructed from the expression. See Berglund and Van der Merwe [5] for a complete discussion both of these semantics and the details of such search implementations; this paper applies fully to the PCRE-style mode in Boost, including the potential for very poor performance for some regular expressions [26] in instances where a significant amount of backtracking is necessary. The POSIX mode is derived from this engine as follows:

1. Apply the PCRE-style matching engine to the input, and record the resulting parse tree t. If the engine rejects the string then it is rejected (as the modes agree on simple membership though not on capturing semantics[4]).
2. Apply the PCRE-style matching engine to the input, and *each* time it *would* accept with a parse tree t':
   (a) if $t \prec_B t'$, set $t \leftarrow t'$, with $\prec_B$ defined precisely as in Definition 10,
   (b) reject, as if the search had failed, causing the engine to backtrack.
3. Output the final t as the POSIX-style match result.

In effect the PCRE-style engine is simply made to explore every possible parse tree by triggering its backtracking. Unfortunately there are some edge cases where this does not quite work, as the PCRE-style engine fails to explore some trees which are from the PCRE perspective not useful candidates, but which are clearly more correct from a POSIX perspective—more on this follows in Section 5—but we view these instances as plain bugs rather than as intended semantics.

The larger issue with this implementation technique is that there may be exponentially many parse trees, and exploring them all may cause very poor performance. For example, with default settings, the Boost POSIX matcher will refuse to attempt to match the string "aaaaaaaaaaaaa" with $/(\texttt{a}^*)^*/$, issuing a warning that the expression should be refactored to avoid "eternal" matching; remove one "a", however, and the match will succeed. Again, see Weideman et al. [26] for a full treatment of this type of matching issues.

[4] The matching engine should also reject on syntax or operators not permitted, as not all PCRE-style features make sense in the POSIX context. The parsing and validation of the expression is not within the scope of this discussion however.

Depending on the application, this may be a rather severe issue, but fortunately, the problem of computing the correct Boost match does not actually require exponential time, as we will see next.

Theorem 1. *Boost captures $B(r, w)$, where $r \in \mathcal{R}(\Sigma, I)$ and $w \in \Sigma^*$, can be computed in time $\mathcal{O}(k|w||r| \log |w|)$, where k is the number of distinct capturing indices used in r.*

Proof. Without loss of generality, assume $I = \{1, \ldots, k\}$, and let $T(r)$ be a transducer, obtained via a modified Thompson construction, which on input w, outputs all matching forests of w; see Berglund and Van der Merwe [5] for a detailed description of such a construction. We associate with each $i \in I$ the sets of transitions O_i and C_i, from $T(r)$, that outputs $(_i$ and $)_i$, respectively. Next, we determine the capturing information for each $i \in I$ in order of priority, so starting with $i = 1$, we use binary search, in conjunction with a modified on-the-fly subset construction, on $T(r)$, to first find the leftmost position in w where we can use a transition, from O_i, for the last time, while matching w from left to right with $T(r)$. That is, the binary search proceeds by stating "the last leftmost position is to the right of position p", then we simulate T on w by keeping track of all states reachable, verifying this assumption. If we succeed, we attempt a smaller p; if we fail, we attempt a larger one, until the precise leftmost last position possible is identified. To make this more precise, the condition is verified by up to position p simulating T, adding to each state reached a flag annotating whether it has been reached on a path which used some transition from O_1 at least once (if the same state is reached with and without using a transition from O_1, the flag is kept). When position p is reached, all states which have not used a transition from O_1 are discarded, and the simulation of T continues, but now *no* transition from O_1 may be used for the remainder of w.

Once we have this first position for capture i fixed—that is, either every scan suggested from here on should again verify this condition on the paths they consider, most easily achieved by a small annotation on w and on T, but as this only *constrains* the possible paths in T it has no negative impact on the matching performance—we again use the same search procedure to determine the rightmost position in w where we can use a transition from C_1, for the last time, while matching w from left to right with $T(r)$.

Combining this modified on-the-fly subset construction—which is to say, tracking of reachable states fulfilling the additional conditions placed by the capturing order—with binary search, allows us to determine the starting position of the capture on index 1 in time $\mathcal{O}(|w||r| \log |w|)$. This is the case as $|T| \in \mathcal{O}(r)$, and checking if T matches w can be done in $|T||w|$, even with the added modifications, as the restrictions only *remove* paths which a full simulation would have to consider.

We now repeat this search index $i = 2$, but while doing the search for starting and ending position of this capture, we use the additional restrictions that transitions from O_1 and C_1 has to be taken at (and not after) the opening and closing positions of the capture on index 1. Repeating this procedure for each index gives us an $\mathcal{O}(k|E||w|log|w|)$ algorithm to compute $B(w, r)$. □

Remark 10. Theorem 1 does establish that matching consistent with the Boost semantics can be performed in polynomial time, which improves greatly on the exponential worst-case of Boost itself. However, this construction is primarily given for illustrative purposes, and it is clearly not the most efficient approach possible: The binary search proposed to optimize the moment when the state machine last uses the transitions corresponding to the captures can be replaced by a more complicated but more efficient linear scan which determines the correct placement outright. The details of such an algorithm are non-trivial, however, so we leave the construction and correctness proof details as future work.

5 Experimental Results

To test our formalism experimentally, we developed two applications in Python: (1) a small testing framework for existing matchers, and (2) a larger, extensible framework that allows us, given a regular expression $r \in \mathcal{R}(\Sigma, I)$ and an input string $w \in \Sigma^*$, to generate the forests $f \in \mathcal{L}(r)$, and then to apply the Boost or POSIX disambiguation policy, for the latter of which we used the Okui–Suzuki approach as proxy. For the sake of simplicity, we limited Σ to alphabetic characters, we did not implement the more involved POSIX regular expression atoms such as bracket expressions and collating elements, and beyond the barest minimum, we did not attempt to make forest generation and matching efficient in any way.

Our main source for test cases was the 93 examples Fowler [8] designed specifically to tease out POSIX compliance: We retained the 49 ERE examples from `interpretation.dat`, removing a further three for containing bracket expressions. Of these, Boost was able to return matches, without resorting to partial matching, for 37 test cases; see Remark 11 for a discussion. We also wrote 19 additional test cases, designed to show the difference between Boost and POSIX disambiguation.

The implementation of our Boost formalism passed all of our own test cases with respect to what the Boost matcher returns. For the 37 test cases, our Boost formalism failed two, which we now discuss.

Example 11. Our formalism disagrees with the Boost matcher on Fowler's test case 10, matching "`x`" with the regular expression `(.?){2}`, where the dot operator indicates a match with any character. Here, we get the forests $f_{16} = [_0[_1]_1[_1\mathtt{x}]_1]_0$ and $f_{17} = [_0[_1\mathtt{x}]_1[_1]_1]_0$. From Definitions 8 and 9, we get $C_{\text{fin}}(f_{16}) = \{(0,0,1),(1,0,1)\}$ and $C_{\text{fin}}(f_{17}) = \{(0,0,1),(1,1,0)\}$, and hence, by Definition 10, we have $f_{17} \prec_B f_{16}$, so that our formalism selects f_{16}. However, the Boost matcher prefers f_{17}, which is to say, it returns `(0,1)(1,1)` instead of the expected `(0,1)(0,1)`.

Running Regex-TDFA on the same example also returns `(0,1)(1,1)`. Therefore, we refer to the POSIX standard, which specifies that duplication "shall match what *repeated consecutive* occurrences" [emphasis added] would match [4, §9.4.6]. This would seem to suggest that `(.?){2}` is equivalent to the literal expansion `(.?)(.?)`. Both Boost and Regex-TDFA now return `(0,1)(0,1)(1,1)` as expected, but note that we had no choice in the second pair of parentheses automatically defining a new group. We might now posit that (1) Boost has the internal forest representations $[_0[_1\mathtt{x}]_1[_2]_2]_0$ and $[_0[_1]_1[_2\mathtt{x}]_2]_0$, (2) it selects the former by our Boost formalism, but then (3) reports this choice as $[_0[_1\mathtt{x}]_1[_1]_1]_0$.

This postulation does not extend to Fowler's test case 17, matching "`xxx`" with `(.?.?){3}`, where Boost returns `(0,3)(2,3)`, capturing the last "`x`" with group 1—unlike Regex-TDFA, which returns `(0,3)(3,3)`; when expanded to `(.?.?)(.?.?)(.?.?)`, both return `(0,3)(0,2)(2,3)(3,3)`. Our point is this: For sensible options of *internal* representation—non-capturing groups, group number reuse and reordering—we can cook up counterexamples, so that the same proposed representation does not work over all test cases. We believe this to be a bug in the Boost matcher: During code inspection, we found code that limits the forests to be explored, an optimisation that short-circuits a duplication when it first matches an empty string, which is fine for PCRE semantics—recall that Boost's POSIX matcher is a modified PCRE engine—but prevents all possibilities from being considered for POSIX semantics. □

Remark 11. In our test setup, Boost only looks for full matches, that is, where the entire input string is matched by the regular expression. *Partial matching* allows a matcher to match a substring of the input string with a regular expression. Because Boost maximises groups (as opposed to subexpressions) from left to right, it is possible to simulate partial matching by prepending and appending `.*` to the regular expressions involved (and if necessary, surrounding the original expression with parentheses). For example, to allow Fowler's test case 28, matching "`ababa`" by `(aba|a*b)`, to succeed, we rewrite the regular expression as `.*(aba|a*b).*`. Doing so allows Boost to return the partial match for the nine Fowler test cases that failed originally, and the results correspond to those returned by our own Boost ordering.

The same construction does not in general return correct results for a classic POSIX matcher set up to return full matches. It will match the first and last "`a`" in "`aba`" with the first and last `.*` of `.*(aba|a*b).*`, respectively, and "`b`" with group 1. A *lazy star*, which consumes as few symbols as possible, is necessary for the construction to work for POSIX matchers [5], but is not supported by the standard.

6 Future Work and Conclusion

Although we focused in this paper mostly on Boost semantics of regular expression matching, the overarching theme of this research is the more general notion of providing users of matching libraries the freedom to specify their own orders (or disambiguating policies) that can be used by more generic regular expression matching libraries. Thus, instead of being locked into the unclear semantics provided by current greedy and POSIX implementations, users can then specify their own policies, such as for example longest-leftmost, instead of the current leftmost-longest policy. Certainly, it might often be of more interest to find a longest submatch rather than a leftmost one. Given that comparators made generic sorting algorithms widely applicable, why not by analogy provide a generic way to specify classes of disambiguating policies to be used by a matcher, while still keeping the matching procedure efficient?

References

1. PCRE – Perl compatible regular expressions, `https://www.pcre.org/`, last accessed, 2018-05-26
2. Regex(3) BSD Library Functions Manual (September 2011), as available on macOS 10.11.6.
3. Regular expression routines – OpenBSD Library Functions Manual (May 2016), `http://man.openbsd.org/regexec`
4. IEEE standard for information technology – portable operating system interface (POSIX) base specifications, issue 7. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (2017). https://doi.org/10.1109/IEEESTD.2008.4694976
5. Berglund, M., van der Merwe, B.: On the semantics of regular expression parsing in the wild. Theoretical Computer Science **679**, 69–82 (2017)
6. Berglund, M., van der Merwe, B.: Re-examining regular expressions with backreferences. In: Holub, J., Žd'árek, J. (eds.) Proceedings of the Prague Stringology Conference (PSC2017). pp. 30–41 (2017)
7. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Information and Computation **140**(2), 229–253 (1998). https://doi.org/10.1006/inco.1997.2688, `http://www.sciencedirect.com/science/article/pii/S0890540197926882`
8. Fowler, G.: An interpretation of the POSIX regex standard. Tech. rep., AT&T Research, Florham Park, NJ (2003), `http://gsf.cococlyde.org/download`
9. Friedl, J.E.F.: Mastering Regular Expressions. O'Reilly, Sebastopol, CA, third edn. (2006)
10. Frisch, A., Cardelli, L.: Greedy regular expression matching. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) Automata, Languages and Programming. pp. 618–629. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27836-8_53
11. Houston, G.: Henry Spencer's regular expression libraries. Git repositories, `https://garyhouston.github.io/regex/`, last accessed, 2018-05-26
12. Kearns, S.M.: Extending regular expressions with context operators and parse extraction. Software: Practice and Experience **21**(8), 787–804 (1991). https://doi.org/10.1002/spe.4380210803, `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380210803`
13. Kuklewicz, C.: Regex-tdfa, `https://hackage.haskell.org/package/regex-tdfa`, last accessed, 2018-05-26
14. Kuklewicz, C.: Summoned – response to blog entry. On Lambda the Ultimate, a programming blog (February 2007), `http://lambda-the-ultimate.org/node/2064`, last accessed, 2018-05-26
15. Kuklewicz, C.: regex-posix-unittest (2009), `https://hackage.haskell.org/package/regex-posix-unittest`, last accessed, 2018-05-26

16. Kuklewicz, C.: Regex Posix. Haskell Wiki (March 2017), `https://wiki.haskell.org/Regex_Posix`, last accessed, 2018-05-26
17. Laurikari, V.: NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In: Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000. pp. 181–187 (2000). https://doi.org/10.1109/SPIRE.2000.878194
18. Laurikari, V.: TRE – the free and portable regex matching library. Git repository, `https://github.com/laurikari/tre/`, last accessed, 2018-05-26
19. Laurikari, V.: TRE documentation, `https://laurikari.net/tre/documentation/regex-syntax/`, last accessed, 2018-05-26
20. Laurikari, V.: Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology (November 2001)
21. Maddock, J.: Boost.Regex (2013), `https://www.boost.org/doc/libs/1_67_0/libs/regex/doc/html/index.html`, last accessed, 2018-05-26
22. Okui, S., Suzuki, T.: Disambiguation in regular expression matching via position automata with augmented transitions. In: Domaratzki, M., Salomaa, K. (eds.) Implementation and Application of Automata. pp. 231–240. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18098-9_25
23. Sulzmann, M., Lu, K.Z.M.: POSIX regular expression parsing with derivatives. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming. pp. 203–220. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_13
24. Sulzmann, M., Lu, K.Z.M.: Derivative-based diagnosis of regular expression ambiguity. In: Han, Y.S., Salomaa, K. (eds.) Implementation and Application of Automata. pp. 260–272. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-40946-7_22
25. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (Jun 1968). https://doi.org/10.1145/363347.363387, `http://doi.acm.org/10.1145/363347.363387`
26. Weideman, N., van der Merwe, B., Berglund, M., Watson, B.: Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In: Implementation and Application of Automata. pp. 322–334 (2016). https://doi.org/10.1007/978-3-319-40946-7_27, `http://dx.doi.org/10.1007/978-3-319-40946-7_27`