Theoretical Computer Science 679 (2017) 69–82

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the semantics of regular expression parsing in the wild [☆]

Martin Berglund [a], Brink van der Merwe [b,*]

[a] Department of Computing Science, Umeå University, Sweden
[b] Department of Computer Science, Stellenbosch University, South Africa

ARTICLE INFO

Article history:
Received 30 November 2015
Received in revised form 7 September 2016
Accepted 11 September 2016
Available online 20 September 2016

Keywords:
Regular expression matchers
Capturing groups
Prioritized transducers

ABSTRACT

We introduce prioritized transducers to formalize capturing groups in regular expression matching in a way that permits straightforward modeling of capturing in Java's[1] regular expression library. The broader questions of parsing semantics and performance are also considered. In addition, the complexity of deciding equivalence of regular expressions with capturing groups is investigated.

1. Introduction

Many regular expression matching libraries perform matching as a form of parsing by using *capturing groups*, and thus output what subexpression matched which substring [9]. This form of regular expression matching requires theoretical underpinnings different from classical regular expressions as defined in formal language theory. A popular implementation strategy used for performing regular expression matching (or parsing) with capturing groups, used for example in Java, .NET and the PCRE library [14], is a worst-case exponential time depth-first search strategy. A formal approach to matching with capturing groups can be obtained by using finite state transducers that output annotations on the input string to signify what subexpression matched which substring [16]. A complicating factor in this approach is introduced by the fact that the matching semantics dictates a single output string for each input string, obtained by using rules to determine a "highest priority" match among the potentially exponentially many possible ones (in contrast, [6] discusses non-deterministic capturing groups).

The pNFA (prioritized non-deterministic finite automaton) model of [3] (a similar formalism was also introduced later in [14]) provides the right level of abstraction to model the matching time behavior of regular expression matchers (at least in Java), as established experimentally in [18]. Also, for matchers based on an input directed depth first search, adding output to pNFA to obtain pTr (prioritized transducers) provides a way of modeling matching with capturing groups.

A regular expression to transducer (with regular lookahead) construction, for regular expressions using a Perl matching strategy, is presented in [16]. Our approach permits an analysis of matching semantics of a subset of the regular expressions supported in Java, but also makes it possible to model alternative matching semantics as found in matchers such as RE2 [7]. In Section 4, where we discuss how to convert regular expressions to pTr, it will become clear that converting regular expressions to pTr is a natural generalization of the Thomson construction for converting regular expressions to

☆ This article is a revised and extended version of [4].
* Corresponding author.
E-mail addresses: mbe@cs.umu.se (M. Berglund), abvdm@cs.sun.ac.za (B. van der Merwe).
1 Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

http://dx.doi.org/10.1016/j.tcs.2016.09.006
0304-3975/© 2016 Elsevier B.V. All rights reserved.

non-deterministic finite automata. We also discuss a linear-time matching algorithm for pTr (i.e. determining the image of input strings), where in contrast [10] presents a parsing algorithm operating directly on regular expressions.

The title of our paper was chosen intentionally to be very similar to the title of a blogpost by Russ Cox [7], in which he describes the functionality and performance of the regular expression matcher RE2. Cox states, without giving any theoretical arguments, that the regular expression matcher RE2 demonstrates that it is possible to use automata theory to implement almost all the features of a modern backtracking regular expression matcher. He further claims that because RE2 is rooted in the theoretical foundation of automata, it provides stronger guarantees on execution time. The aim of our paper is to provide an automata-based theoretical foundation for the basic functionality of modern regular expression matchers (with a focus on the Java regular expression standard library), from which it will follow that the acceptable execution times mentioned by Cox are indeed attainable.

Our main contribution is in defining matching of strings by regular expressions in such a way to incorporate what subexpression matched which substring and in showing that for a given regular expression, a pTr can be constructed that provides exactly the same matching information. Our other contributions are in providing complexity bounds for determining the output string produced by a pTr for a given input string and for equivalence checking of regular expressions when using our more general definition of regular expression matching.

The outline of the paper is as follows. In the next section we define prioritized automata and transducers. After this we discuss, with examples, the regular expression matching in Java, and also the POSIX standard. This motivates the approach we follow in adapting the standard Thompson construction for converting regular expressions to non-deterministic finite automata, from [17], to the more general setting of converting regular expressions to prioritized transducers. The following section gives a normal form for prioritized transducers, the so-called flattened prioritized transducers that simplifies discussions in the next section on deciding equivalence of and parsing with pTr.

2. Definitions

Let $\mathrm{dom}(f)$ and $\mathrm{range}(f)$ denote the domain and range of a function f, respectively. When unambiguous let a function f with $\mathrm{dom}(f) = S$ generalize to S^* and $\mathcal{P}(S)$ element-wise, where $\mathcal{P}(S)$ denotes the power set of the set S. The cardinality of a (finite) set S is denoted by $|S|$. We denote by $\mathbb{N}$ the set of natural numbers, i.e. the set $\{1, 2, 3, \ldots\}$. The empty string is denoted ε. An alphabet Σ is a finite set of symbols with $\varepsilon \notin \Sigma$. We denote $\Sigma \cup \{\varepsilon\}$ by Σ^ε. For any string w let $\pi_S(w)$ be the maximal subsequence of w containing only symbols from S (e.g. $\pi_{\{a,b\}}(abcdab) = abab$).

If $w_1 \in \Sigma_1^*$ and $w_2 \in \Sigma_2^*$, with Σ_1 and Σ_2 disjoint alphabets, then $w \in (\Sigma_1 \cup \Sigma_2)^*$ is in the *shuffle of* w_1 *and* w_2 if $\pi_{\Sigma_1}(w) = w_1$ and $\pi_{\Sigma_2}(w) = w_2$. The shuffle of two languages L_1 and L_2, over disjoint alphabets, is the shuffle of all pairs of words from L_1 and L_2 respectively.

For sequences $s = (z_{1,1}, \ldots, z_{1,n}) \ldots (z_{m,1}, \ldots, z_{m,n}) \in (Z_1 \times \ldots \times Z_n)^*$, we denote by $\sigma_i(s)$ the subsequence of tuples obtained from s by deleting duplicates of tuples in s and only keeping the first occurrence of each tuple, where equality of tuples is based only on the value of the ith component of a tuple (e.g. $\sigma_1((1,a)(2,a)(1,b)(3,b)(2,c)) = (1,a)(2,a)(3,b)$). For each $k > 1$, we denote by B_k the alphabet of k types of brackets, which is represented as $\{[_1,]_1, [_2,]_2, \ldots [_k,]_k\}$. The Dyck language D_k over the alphabet B_k is the set of strings representing well balanced sequences of brackets over B_k.

As usual, a regular expression over an alphabet Σ (where $\varepsilon \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \,|\, E')$, $(E \cdot E')$, or (E^*), where E and E' are regular expressions. Some parentheses can be dropped with the rule that $*$ (Kleene closure) takes precedence over $\cdot$ (concatenation), which takes precedence over $|$ (union). Further, outermost parentheses can be dropped, and $E \cdot E'$ can be written as EE'. The language of a regular expression E, denoted $\mathcal{L}(E)$, is obtained by evaluating E as usual, where $\emptyset$ stands for the empty language and $a \in \Sigma \cup \{\varepsilon\}$ for $\{a\}$. The *size* of E, denoted $|E|$, is the number of symbols appearing in E. A *capturing group* is any parenthesized subexpression, e.g. (E). Brackets in regular expressions are used both for precedence and capturing. The precise matching and capturing semantics follow from Section 4.

For later constructions we require a few different kinds of automata and transducers. First (non-)deterministic finite automata (and their runs when applied to strings), followed by the prioritized finite automata from [3], which are used to model the capturing behavior of (some of the most popular) regular expression matching libraries.

Definition 1. A *non-deterministic finite automaton* (NFA) is a tuple $A = (Q, \Sigma, q_0, \delta, F)$ where:

- Q is a finite set of *states;*
- Σ is the *input alphabet;*
- $q_0 \in Q$ is the *initial state;*
- $\delta : Q \times \Sigma^\varepsilon \to \mathcal{P}(Q)$ is the *transition function; and*
- $F \subseteq Q$ is the set of *final states.*

A is *ε-free* if $\delta(q, \varepsilon) = \emptyset$ for all q. A is *deterministic* if it is *ε-free* and $|\delta(q, \alpha)| \le 1$ for all q and α. The *state size* of A is denoted by $|A|_Q$, and defined to be $|Q|$.

Definition 2. For an NFA $A = (Q, \Sigma, q_0, \delta, F)$ and $w \in \Sigma^*$, a *run* for w is a string $r = s_0\alpha_1 s_1 \cdots s_{n-1}\alpha_n s_n \in (Q \cup \Sigma)^*$, with $s_0 = q_0$, $s_i \in Q$ and $\alpha_i \in \Sigma^\varepsilon$ such that $s_{i+1} \in \delta(s_i, \alpha_{i+1})$ for $0 \le i < n$, and $\pi_\Sigma(r) = w$.

A run is accepting if $s_n \in F$. The language accepted by A, denoted by $\mathcal{L}(A)$, is the set $\{\pi_\Sigma(r) \mid r \text{ is an accepting run of } A\}$.

Now we define the prioritized NFA variant, as in [3], where an ordering is placed on ε-transitions at a given state, and where these ε-transitions are considered in the order specified to obtain what we refer to as the highest priority accepting path for a given input string, if the input string under consideration is accepted by the NFA.

Definition 3. A *prioritized non-deterministic finite automaton (pNFA)* is a tuple $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, where if $Q = Q_1 \cup Q_2$, we have:

- Q_1 and Q_2 are disjoint finite sets of *states;*
- Σ is the *input alphabet;*
- $q_0 \in Q$ is the *initial state;*
- $\delta_1 : Q_1 \times \Sigma \to Q$ is the *deterministic*, but not necessarily total, *transition function;*
- $\delta_2 : Q_2 \to Q^*$ is the *non-deterministic prioritized transition function;* and
- $F \subseteq Q_1$ are the *final states.*

Remark 1. For a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ the *corresponding finite automaton* $\mathrm{nfa}(A)$ is given by $\mathrm{nfa}(A) = (Q_1 \cup Q_2, \Sigma, q_0, \bar{\delta}, F)$, where

$$\bar{\delta}(q, \alpha) = \begin{cases} \{\delta_1(q, \alpha)\} & \text{if } q \in Q_1 \text{ and } \alpha \in \Sigma, \\ \{q_1, \ldots, q_n\} & \text{if } q \in Q_2,\ \alpha = \varepsilon, \text{ and } \delta_2(q) = q_1 \ldots q_n. \end{cases}$$

For a pNFA A and string $w \in \mathcal{L}(\mathrm{nfa}(A))$, we define an ordering on the accepting runs of w in $\mathrm{nfa}(A)$, by assigning to a run using the transition from q to q_i a higher priority than a run using the transition q to q_j, when $i < j$, $\delta_2(q) = q_1 \ldots q_n$, and assuming all transitions before the transitions going from q to q_i or q_j are the same in both runs. By using this ordering, an accepting run for a string w in a pNFA A, is defined to be the largest accepting run of w in $\mathrm{nfa}(A)$, not repeating the same ε-transition in a subsequence of consecutive ε-transitions. Prioritized NFA are thus on a conceptual level closely related to unambiguous NFA, since in an pNFA there is at most one accepting run for an input string. The repeated ε-transition restriction is made to ensure that we consider only finitely many of the runs of $\mathrm{nfa}(A)$ for a given input string w, when determining the highest priority path (referred to as a run of A) for w, and also to ensure that regular expression matchers based on pNFA/pTr do not end up in an infinite loop during (attempted) matching.

Definition 4. For a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, a *path* of $w \in \Sigma^*$ in A, is a run $s_0\alpha_1 s_1 \cdots s_{n-1}\alpha_n s_n$ of w in $\mathrm{nfa}(A)$, i.e. $s_i \in Q$ and $\alpha_i \in \Sigma^\varepsilon$, such that if $\alpha_i = \alpha_{i+1} = \ldots = \alpha_{j-1} = \alpha_j = \varepsilon$, with $i \le j$, then $(s_{k-1}, s_k) = (s_{l-1}, s_l)$, with $i \le k, l \le j$, implies $k = l$, i.e. a path is not allowed to repeat the same transition in a sequence of ε-transitions. For two paths $p = s_0\alpha_1 s_1 \cdots s_{n-1}\alpha_n s_n$ and $p' = s'_0\alpha'_1 s'_1 \cdots s'_{m-1}\alpha'_m s'_m$ we say that p is of higher priority than p', $p > p'$, if $p \ne p'$, $\pi_\Sigma(p) = \pi_\Sigma(p')$ and either p' is a proper prefix of p, or if j is the first index such that $s_j \ne s'_j$, then $\delta_2(s_{j-1}) = \cdots s_j \cdots s'_j \cdots$. An accepting run for A on w is the highest-priority path $p = s_0\alpha_1 s_1 \cdots \alpha_n s_n$ such that $\pi_\Sigma(p) = w$ and $s_n \in F$. The language accepted by A, denoted by $\mathcal{L}(A)$, is the subset of Σ^* defined by $\{\pi_\Sigma(r) \mid r \text{ is an accepting run of } A\}$.

From the previous definition we have that $\mathcal{L}(A) = \mathcal{L}(\mathrm{nfa}(A))$.

Note that we introduced pNFA (and runs of pNFA in Definition 4) mainly as an aid in defining runs of prioritized transducers in Definition 6 below.

Example 1. In Fig. 1(a), a pNFA A for the regular expression $(a^*)^*$, constructed as described in [3], is given. The accepting run for the string a^n, in A, is $q_0q_1(q_2aq_1)^nq_0q_3$. Since there are for the input strings a^n, $n \ge 0$, exponentially many paths in A, a regular expression matcher using an input directed depth first search (i.e. a search exploring runs of prefixes of an input string in an order of increasing priority), such as the Java implementation, will take exponential time to attempt to match the strings a^nx, for $n \ge 0$ (see [3] for more details on exponential matching time in Java).

Recall that a transducer T ([15]) is a tuple $(Q, \Sigma_1, \Sigma_2, q_0, \delta, F)$, where Q is a finite set of states, Σ_1 and Σ_2 are the input and output alphabets respectively, $\delta \subseteq Q \times \Sigma_1^\varepsilon \times \Sigma_2^* \times Q$ is the set of transitions, q_0 is the initial state and F is the set of final states. Accepting runs are defined as for NFA, but in a run when moving from state q to q' while reading input x and using the transition (q, x, y, q'), the string y is also produced as output. The *state size* of T, denote by $|T|_Q$, is the number of states in T, the *transition size*, $|T|_\delta$, is the sum of $(1 + |y|)$ over all transitions (q, x, y, q'), and the *size* of T, denoted by $|T|$, is $|T|_Q + |T|_\delta$. A transducer T defines a relation $\mathcal{R}(T) \subseteq \Sigma_1^* \times \Sigma_2^*$, containing all pairs (v, w) for which there is an accepting run reading input v and producing w as output while moving from the first to last state in the accepting run (i.e. v and w are the concatenation of the input symbols and output strings respectively, of all the transitions taken in the accepting

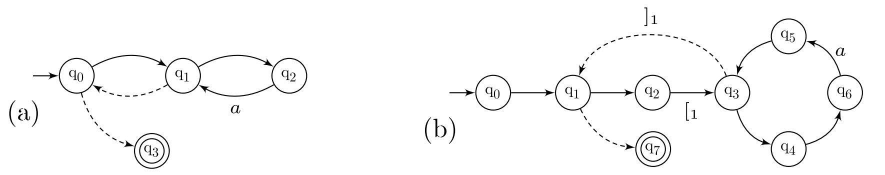

Fig. 1. (a) pNFA for the regular expression $(a^*)^*$, i.e., the pNFA $A = (\{q_2, q_3\}, \{q_0, q_1\}, \{a\}, q_0, \{(q_2, a, q_1)\}, \{[q_0, (q_1, q_3)], [q_1, (q_2, q_0)]\}, q_3)$. (b) pTr with $\Sigma_1 = \{a\}$ and $\Sigma_2 = \{[_1,]_1\}$, for $(a^*)^*$. Lower priority transitions are indicated by dashed edges.

run). As usual, we denote by $\text{dom}(T)$ the set $\{v \in \Sigma_1^* \mid (v, w) \in \mathcal{R}(T)\}$, and by $\text{range}(T)$ the set $\{w \in \Sigma_2^* \mid (v, w) \in \mathcal{R}(T)\}$. For functional transducers (see [15], Chapter 5), the relation $\mathcal{R}(T)$ is a function, and we write $T(v) = w$ if $(v, w) \in \mathcal{R}(T)$. Prioritized string transducers, defined next, also define relations, in this case contained in $\Sigma_1^* \times (\Sigma_1 \cup \Sigma_2)^*$, which are in fact functions, and the notation $\mathcal{R}(T)$, $\text{dom}(T)$, $\text{range}(T)$, and $T(v) = w$ if $(v, w) \in \mathcal{R}(T)$, will thus also be used.

Definition 5. A *prioritized non-deterministic finite transducer (pTr)* is a tuple $T = (Q_1, Q_2, \Sigma_1, \Sigma_2, q_0, \delta_1, \delta_2, F)$, where if $Q := Q_1 \cup Q_2$, we have:

- Q_1 and Q_2 are disjoint finite sets of *states;*
- Σ_1 is the *input alphabet;*
- Σ_2, disjoint from Σ_1, is the *output* alphabet;
- $q_0 \in Q$ is the *initial state;*
- $\delta_1 : Q_1 \times \Sigma_1 \to Q$ is the *deterministic*, but not necessarily total, *transition function;*
- $\delta_2 : Q_2 \to (\Sigma_2^* \times Q)^*$ is the *non-deterministic prioritized transition and output function*; and
- $F \subseteq Q_1$ are the *final states.*

The *state size* of T is $|T|_Q := |Q_1| + |Q_2|$, the δ_1 *transitions size* $|T|_{\delta_1} := \sum_{q \in Q_1, a \in \Sigma_1} |\delta_1(q, a)|$ where $|\delta_1(q, a)| = 1$ if $\delta_1(q, a)$ is defined and 0 otherwise, the δ_2 *transitions size* $|T|_{\delta_2} := \sum_{q \in Q_2} |\delta_2(q)|$ where $|\delta_2(q)|$ equals $\sum_i (1 + |w_i|)$ if $\delta_2(q) = (w_1, q_1) \dots (w_n, q_n)$ (and $|\delta_2(q)| = 0$ if $\delta_2(q) = \varepsilon$), the *transitions size* $|T|_\delta := |T|_{\delta_1} + |T|_{\delta_2}$, and finally, the *size* of T is $|T| := |T|_Q + |T|_\delta$.

Going forward, when discussing a pTr T without being specific on the tuple, we assume that $T = (Q_1, Q_2, \Sigma_1, \Sigma_2, q_0, \delta_1, \delta_2, F)$.

Next we define the semantics of pTr, which makes them define partial functions from Σ_1^* to $(\Sigma_1 \cup \Sigma_2)^*$. The pTr are viewed as devices which consume strings in their domain (a subset of Σ_1^*), to produce output by *decorating* the input string with symbols from Σ_2. For a pTr T (defined as above), let $\text{pnfa}(T)$ denote the pNFA obtained from T by letting $\text{pnfa}(T) = (Q_1, Q_2, \Sigma_1, q_0, \delta_1, \delta_2', F)$, with $\delta_2'(q) = q_1 \dots q_n$ if $\delta_2(q) = (w_1, q_1) \dots (w_n, q_n)$ for some $w_i \in \Sigma_2^*$. For a pTr T, the runs of $\text{pnfa}(T)$ determine the decorated output string, in $(\Sigma_1 \cup \Sigma_2)^*$, produced from a given input string in Σ_1^*. When applying the function δ_1 on (q, α), T produces α as output, where when using δ_2 on q with $\delta_2(q) = (w_1, q_1) \dots (w_n, q_n)$, one of the w_i's is produced as output.

Definition 6. Let $T = (Q_1, Q_2, \Sigma_1, \Sigma_2, q_0, \delta_1, \delta_2, F)$ be a pTr, and let Q denote $Q_1 \cup Q_2$ and Σ the set $\Sigma_1 \cup \Sigma_2$. An accepting run for $v \in \Sigma_1^*$ in T is a string $r = s_0 \alpha_1 s_1 \cdots s_{n-1} \alpha_n s_n \in (Q \cup \Sigma^\varepsilon)^*$, $s_i \in Q$ and $\alpha_i \in (\Sigma_1 \cup \Sigma_2)^*$, such that $\pi_{Q \cup \Sigma_1}(r)$ is an accepting run of v in $\text{pnfa}(T)$, and if $s_i \in Q_2$, with $i < n$, then the sequence of tuples defined by $\delta_2(s_i)$ contains (α_{i+1}, s_{i+1}). The pTr T defines a partial function from Σ_1^* to Σ^* by $T(v) = w$, if there is an accepting run r for v in T with $\pi_\Sigma(r) = w$.

Remark 2. Note that for pTr we may assume that $\delta_2(q) = (w_1, q_1) \dots (w_n, q_n)$ implies that the states $q_1, \dots, q_n$ are pairwise distinct, since if $q_i = q_j$ with $i < j$, then if the remainder of the input is not accepted from q_i, it will also not be accepted from q_j, and thus (w_j, q_j) may be removed from $\delta_2(q)$.

In the next example we use the concept of equivalence of pTr, where pTr T and T' are defined to be equivalent if they are equal as functions.

Example 2. In Fig. 1(b), a pTr T for the regular expressions $(a^*)^*$ is given (constructed as described in Section 4). In this case, only the substrings matched by the subexpression a^* are captured, and the captured substrings are enclosed by the pair of brackets in $B_1 = \{[_1,]_1\}$. Also, $\text{dom}(T)$ is a^*, and $T(a^n) = [_1 a^n]_1$ for $n \geq 0$. It should be pointed out that the Java regular expression matcher in fact only prohibits duplicates of the ε transitions $q_1 \to q_2$ and $q_3 \to q_4$ (in Fig. 1(b)) in a subsequence of ε transitions of consecutive ε transitions (of a run), and thus in the case of Java we have $T(a^n) = [_1 a^n]_1 [_1]_1$ for $n \geq 1$. In general, the Java matcher only prohibits duplicates of the ε transitions $f_1 \to q_1$ (in a subsequence of consecutive ε transitions of a run) in the lazy and greedy Kleene closure in Figs. 5(a) and (b) in Section 4.

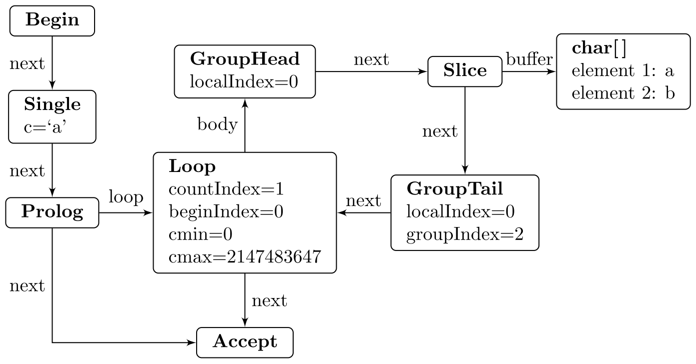

Fig. 2. The object graph for the regular expression $a(ab)^*$ in Java, with some irrelevant objects pruned. Each box is an instance of the class labeling it in bold (e.g. `Loop` is the class `java.util.regex.Pattern$Loop`), the arrows are references, and in the boxes the member variables are listed.

For the regular expressions $R = (a)(a^*)$ and $R' = (a^*)(a)$, we have $\mathcal{L}(R) = \mathcal{L}(R')$, but the corresponding pTr T and T' are not equivalent, since $T(a^n) = [_1a]_1[_2a^{n-1}]_2 \neq [_1a^{n-1}]_1[_2a]_2 = T'(a^n)$, for $n = 1$, or $n \geq 3$. Note that the same subexpression in a regular expression may capture more than one substring, for example, if T'' is a pTr for $(a^*|b)^*$, then $T''(a^pb^qa^r) = [_1a^p]_1[_1b]_1 \ldots [_1b]_1[_1a^r]_1$, for $p, q, r > 0$.

3. The semantics of practical regular expression parsing

A full formal characterization of the semantics of the regular expression matching in real systems is, unfortunately, not in the scope of this article. The problem is primarily one of sheer size, since the class in Java[2] which handles the internal representation of regular expressions (`java.util.regex.Pattern`) is 5856 lines of code, and interacts with several other classes. A pure logical characterization of the semantics of the implementation would consume vast amounts of space. The Java documentation also makes no great effort to explain the semantics. Still, the broad intents of the implementation can be summarized reasonably compactly. We first exemplify the implementation techniques applied in Java, and will in Section 4 explain the transducer model that is used to describe the Java capturing semantics.

3.1. An example from Java

We consider in Java the regular expression $a(ab)^*$. This is represented internally as in Fig. 2.

Java effectively performs a depth-first search on such a graph, but in practice different objects achieve the effect of the search in different ways. The initial "a" in the expression is represented by the `Single` class, which reads a single symbol, the member `c`, from the input and calls its `next` reference to match the remainder if successful. If it fails it returns to its parent (on the call stack), where matching may continue depth-first along some other path (though not in this case). `Slice` achieves the same purpose, matching a substring. There are in fact a total of twelve `Single` and `Slice` variants for different cases (including a Boyer-Moore implementation of `Slice` optimizing the case of regular expressions consisting mostly of a single long substring).

Of special note here are the `GroupHead` and `GroupTail` instances, representing the capturing group (ab). The `localIndex` member variable is an index into a global "locals" array, allocating index zero to store the string position where `GroupHead` last started matching. The old value is saved on the stack and restored when backtracking. `GroupTail` retrieves the value at index zero of the "locals" array and saves the beginning and end position of the match of the group into a global "group" array. The position saved in the "group" array is controlled by the `groupIndex` member variable, and since $(ab)^*$ is group 1 in the regular expression $a(ab)^*$, the position saved is 2, which instructs this instance of `GroupTail` to use position 2 and 3 (for starting and ending positions of group 1) in the global "group" array. This will get overwritten if the group is matched multiple times, but will get restored from the stack on backtracking. After a successful match, the "group" array is inspected to learn the position of the last matching of each capturing group.

`Loop` implements the Kleene closure, calling its `body` reference to match until either the maximum number of calls have been made ($2^{31} - 1$ times being the default, representing effective infinity), `body` backtracks, or `body` matches the empty string, at which point `Loop` calls `next` instead. There are several variants of `Loop` as well, each implementing these cycle termination semantics in different, but equivalent ways. For example, see Fig. 3, showing the representation of a simpler expression. In this context Java instead uses the class `Curly`, which achieves the same thing as `Loop`, but does not employ recursion, and instead matches the body in a loop. This improves efficiency while also abusing the depth-first search by not allowing the child `Single` to accept when it would otherwise be expected to do so.

[2] We use Java version 8u40 when not otherwise noted.

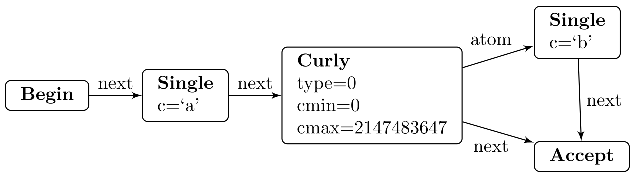

Fig. 3. The object graph for the regular expression ab^* in Java, with some irrelevant objects pruned.

These differences illustrate why a strict equivalence proof is difficult, but the semantics in themselves can be captured, and informally motivated.

3.2. *The semantics of POSIX regular expression parsing*

Next we briefly discuss why we decided against also considering POSIX regular expression matching in this paper. First it should be pointed out that implementations of the POSIX standard are a lot less common [9], making it a more academic target of study. The relevant part of the POSIX standard is specified as follows [11]:

> "Consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, shall match the longest possible string. For this purpose, a null string shall be considered to be longer than no match at all."

Consider for example the expression $((b^*)(ba^* \mid \varepsilon))a^*$ and input string *baa*. In POSIX, the overall group $((b^*)(ba^* \mid \varepsilon))$ would have the length of its match maximized, reading all of the input string with the $(ba^* \mid \varepsilon)$ subexpression, and letting the subexpressions b^* and the final a^* match the empty string. Also note that concatenation is not associative in POSIX matching, since when changing $((b^*)(ba^* \mid \varepsilon))a^*$ to $(b^*)((ba^* \mid \varepsilon)a^*)$, and still matching the input *baa*, b^* will match the first b in the input, and the final a^* the remaining a's. Matters get muddier in more complex cases however, as the definition of "subpattern" in the above quote from the standard is disputed. The popular Boost library [12] follows the lead of Glenn Fowler [8] and interprets this to mean that *capturing groups* should be maximized from left to right, whereas for example the Haskell regex-tdfa package, and e.g. [5] interprets the definition to apply to any subexpression, whether capturing or not. Even so, this still leaves a lot of room for interpretation in the case where one subexpression matches multiple substrings. For example, is a match where a subexpression matches only once, matching say a substring of length 2, preferred to a match where the same subexpression matches twice, first a substring of length 1 and then of length 3?

The lack of clarity in these matters make a thorough treatment difficult. Furthermore, the techniques of this article are hard to apply directly to POSIX matching (at least for some of its interpretations), as the next theorem shows.

Theorem 1. *The output language of Boost-style POSIX regular expression matching is not necessarily regular.*

Proof. Recall that the Boost library [12] interprets the POSIX standard to require that the *length of capturing strings* should be maximized, prioritizing the capturing groups left to right.

Consider the regular expression $E = a^*b^*(a^*b^*)a^*b^*$ on the subset of strings of the form a^kba^lb, and a machine M implementing this POSIX interpretation in the same sense as pTr, decorating the input according to captured groups:

$$M(a^kba^lk) = \begin{cases} [a^kb]a^lb & \text{when } k \geq l, \\ a^kb[a^ln] & \text{when } k < l. \end{cases}$$

That is, M will make the middle capturing group match whichever is the longer sequence of as. This makes for a non-regular output language. To see this, simply take a sufficiently large k and apply the pumping lemma to $[a^kb]a^kb$. The pumped substring can clearly not contain a b or a bracket, so it consists only of as. However, no a can be removed from the first part, nor any added to the second, since M would then bracket the second part. Thus no pumping can be done while remaining in the output language of M. □

As such, we leave POSIX aside for now, as different techniques would need to be devised to handle Boost-style POSIX semantics in a finite automata context.

4. Converting regular expressions into pTr

Next we get into the formal constructions, giving a Java based construction to turn a regular expression E into an equivalent pTr $\bar{J}^p(E)$. Before defining $\bar{J}^p(E)$, however, we give the capturing semantics of regular expressions in Definition 9, with some slight simplification. Due to a lack of formal specification of the capturing semantics of regular expressions in Java, this definition was derived and verified after extensive code review and experimentation with the Java regular expression matcher. After defining $\bar{J}^p(E)$, we show (in Theorem 2) that $\bar{J}^p(E)$ produces precisely the capturing semantics of E as given in Definition 9.

Consider a pTr T and let $u(T)$ denote the string transducer obtained by ignoring the priorities in T, then for $w \in \mathcal{L}(E)$, $u(\bar{J}^p(E))(w)$ produces as output all possible ways in which E can match w with capturing information indicated. This is equal to the set $M_E(w)$ of non-deterministic matches as given in Definition 7. The pTr $\bar{J}^p(E)$ selects the highest priority match from $M_E(w)$, when given w as input.

We denote the set of subexpressions of E by SUB(E). Assume $F_1, \ldots, F_k$ are the subexpressions in SUB(E), with the order obtained from a preorder traversal of the parse tree of E, or equivalently, ordered from left to right, based on the starting position of each subexpression in the overall regular expression. Note that if the same subexpression appears more than once in E, we regard these occurrences as distinct elements in SUB(E). Also, let $t : \mathrm{SUB}(E) \to \mathbb{N}$ be defined by $t(F_i) = i$.

Without stating it explicitly (in inductive definitions), we assume that the labels $t(F')$ of subexpression F' of a subexpression F of E, are renumbered when necessary, to ensure unique labels for subexpressions of E.

To simplify our exposition of the regular expression to pTr construction procedure, we assume that matches by all subexpressions are captured, and that the pair of brackets $[_i,]_i \in B_k$ indicates matches by the i-th subexpression. The more general case of placing brackets only around the substrings matched by subexpressions that are marked as capturing subexpressions is obtained by replacing some of the pairs of brackets by ε (in our pTr constructions) and renumbering the remaining brackets appropriately.

In addition to non-deterministic matches and a total ordering on prefixes of matches of a given string, the *lazy Kleene star* of an expression E, denoted by $E^{*?}$, is also defined in Definitions 7, 8 and 9. We have that $L(E^*) = L(E^{*?})$, and the distinction between $L(E^*)$ and $L(E^{*?})$ is only evident once we consider a highest priority overall match (as in Definition 9) of a string w by a regular expression E, where E contains subexpressions of the form $F^{*?}$.

Furthermore, to simplify our exposition, we also do not consider regular expressions E, with subexpressions F^* or $F^{*?}$, where $\varepsilon \in \mathcal{L}(F)$ (referred to as problematic regular expressions), in Definitions 7, 8, 9 and Theorem 2. Examples 2 and 3 show that there are more than one plausible way of defining capturing semantics for problematic regular expressions. This is also the subclass of regular expressions on which Java and RE2 capturing semantics differ (as verified experimentally). We thus leave a careful study of capturing semantics for problematic regular expressions as future work, but will indicate briefly in Remarks 3 and 4 the changes required to also cater for these cases.

Definition 7. A *non-deterministic match of a string* w *by a regular expression* E, with $w \in \mathcal{L}(E)$, is defined inductively on the number of subexpressions of the regular expression E, as follows.

- Let $E = \varepsilon$. Then $[_{t(E)}]_{t(E)}$ is a non-deterministic match for $w = \varepsilon$ by E.
- Let $E = a$ and $w = a$, with $a \in \Sigma$. Then $[_{t(E)}a]_{t(E)}$ is a non-deterministic match for $w = a$ by E.
- Let $E = F_1 \mid F_2$ and, for $i = 1$ *or* $i = 2$, let $w \in \mathcal{L}(F_i)$ and let $m_i(w)$ be a non-deterministic match for w by F_i. Then $[_{t(E)}m_i(w)]_{t(E)}$ is a non-deterministic match of w by E.
- Let $E = F_1 \cdot F_2$ and $w = w_1w_2$, then for $i \in \{1, 2\}$ let $w_i \in \mathcal{L}(F_i)$ and let $m(w_i)$ be a non-deterministic match for w_i by F_i. Then we have that $[_{t(E)}m(w_1)m(w_2)]_{t(E)}$ is a non-deterministic match of w by E.
- Let $E = F^*$ or $E = F^{*?}$, and $w = \varepsilon$. Then $[_{t(E)}]_{t(E)}$ is a non-deterministic match of w by E.
- Let $E = F^*$ or $E = F^{*?}$ and let $w = w_1 \ldots w_n$ be such that for all $i \in \{1, \ldots, n\}$ we have $w_i \in \mathcal{L}(F)$ and $m(w_i)$ is a non-deterministic match for w_i by F. Then we have that $[_{t(E)}m(w_1) \ldots m(w_n)]_{t(E)}$ is a non-deterministic match of w by E.

We denote by $M_E(w)$ the set of non-deterministic matches of the string w by the regular expression E. Also, $M_E(w) \neq \emptyset$ if and only if $w \in \mathcal{L}(E)$.

Remark 3. Note that modifying Definition 7 to permit subexpressions of the form F^* or $F^{*?}$, where $\varepsilon \in \mathcal{L}(F)$, is not very difficult, but needs to be done for each set of plausible set of semantics, and some of the details are elaborate. We would for example in Java allow only $w_i = \varepsilon$ in $[_{t(F)}m(w_1) \ldots m(w_n)]_{t(F)}$ when $i = n$, and we define $\bar{J}^p(F^*)$ (and $\bar{J}^p(F^{*?})$) to be even more restrictive, by allowing $w_i = \varepsilon$ only when $n = 1$.

The ordering $\succ$ on non-deterministic matches, defined next, is required in Definition 9.

Definition 8. We define a *total ordering* $\succ$ *on the non-deterministic matches*, of prefixes of the string w, by a regular expression E, by induction on the number of subexpressions of the regular expression E, as follows.

- Let $E = F_1 \mid F_2$, for each $i \in \{1, 2\}$ let w_i be a prefix of w, and take any $m(w_i) \in M_{F_i}(w_i)$. Then $[_{t(E)}m(w_1)]_{t(E)} \succ [_{t(E)}m(w_2)]_{t(E)}$.
- Let $E = F_1 \cdot F_2$, let $w_1w_2, w'_1w'_2$ be prefixes of w, and for all $i \in \{1, 2\}$ let $m(w_i) \in M_{F_i}(w_i), m(w'_i) \in M_{F_i}(w'_i)$. Then

$$[_{t(E)}m(w_1)m(w_2)]_{t(E)} \succ [_{t(E)}m(w'_1)m(w'_2)]_{t(E)}$$

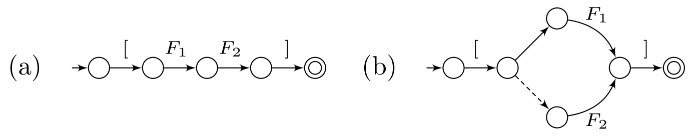

Fig. 4. Java based regular expression to pTr constructions for (a) (F_1F_2) (b) $(F_1 \mid F_2)$. Lower priority transitions are indicated by dashed edges. The pair of brackets [,] are used to indicate the substring or substrings captured by F_1F_2 and $F_1 \mid F_2$, respectively.

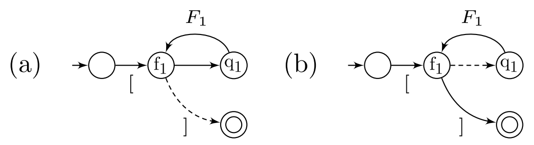

Fig. 5. Java based regular expression to pTr constructions for (a) (F_1^*) and (b) $(F_1^{*?})$. Lower priority transitions are indicated by dashed edges. The pair of brackets [,] are used to indicate the substring or substrings captured by F_1^* and $F_1^{*?}$ respectively.

if $m(w_1) \succ m(w'_1)$; or; $m(w_1) = m(w'_1)$ and $m(w_2) \succ m(w'_2)$.

- Let $w_1 \ldots w_m$ and $w'_1 \ldots w'_n$ be prefixes of w, and let $m(w_i) \in M_F(w_i)$ for all $i \in \{1, \ldots, m\}$, and $m(w'_i) \in M_F(w'_i)$ for all $i \in \{1, \ldots, n\}$.
  - If $E = F^*$, then
  $$[_{t(E)} m(w_1) \ldots m(w_m)]_{t(E)} \succ [_{t(E)} m(w'_1) \ldots m(w'_n)]_{t(E)}$$
  if $m(w'_1) \ldots m(w'_m)$ is a proper prefix of $m(w_1) \ldots m(w_m)$, or for the first index i such that $m(w_i) \neq m(w'_i)$, we have $m(w_i) \succ m(w'_i)$.
  - If $E = F^{*?}$, then
  $$[_{t(E)} m(w_1) \ldots m(w_m)]_{t(E)} \succ [_{t(E)} m(w'_1) \ldots m(w'_n)]_{t(E)}$$
  if $m(w_1) \ldots m(w_m)$ is a proper prefix of $m(w'_1) \ldots m(w'_m)$, or for the first index i such that $m(w_i) \neq m(w'_i)$, we have $m(w_i) \succ m(w'_i)$.
- Let w' be a prefix of w and take $m \in M_E(w')$ such that $m \neq [_{t(E)}]_{t(E)}$. Then
  - if $E = F^*$, $m \succ [_{t(E)}]_{t(E)}$;
  - if $E = F^{*?}$, $[_{t(E)}]_{t(E)} \succ m$.

Remark 4. For subexpressions of the form F^* or $F^{*?}$, where $\varepsilon \in \mathcal{L}(F)$, we need to add $[_{t(F^*)}[_F\ ]_F]_{t(F^*)} \succ [_{t(F^*)}\ ]_{t(F^*)}$ and $[_{t(F^{*?})}\ ]_{t(F^{*?})} \succ [_{t(F^{*?})}[_F\ ]_F]_{t(F^{*?})}$ to Definition 8.

Definition 9. Let E be a regular expression and take $w \in \mathcal{L}(E)$. Then a *match of* w *by* E, denoted by $\overline{m}_E(w)$, is the highest priority match in $M_E(w)$ when using $\succ$ as in Definition 8.

For a regular expression E we define a prioritized transducer $P := \bar{J}^p(E)$ such that $\mathrm{dom}(P) = \mathcal{L}(E)$, and $\mathrm{range}(P)$ is contained in the shuffle of $\mathrm{dom}(P)$ and the Dyck language D_k.

The classical Thompson construction converts the parse tree T of a regular expression E into an NFA, which we denote by $Th(E)$, by doing a postorder traversal on T. An NFA is constructed for each subtree T' of T, equivalent to the regular expression represented by T'. In [3] it was shown how to modify this construction to obtain a Java based pNFA denoted by $J^p(E)$, instead of the NFA $Th(E)$, from E. Here we take it one step further, and modify the construction of $J^p(E)$ to return a pTr, denoted by $\bar{J}^p(E)$, from E. Just as in the case of the constructions for $Th(E)$ and $J^p(E)$, we define $\bar{J}^p(E)$ recursively on the parse tree for E. For each subexpression F of E, $\bar{J}^p(F)$ has a single initial state with no incoming transitions and a single outgoing δ_2 transition, and a single final state with a single incoming δ_2 transition and no outgoing transitions. The constructions of $\bar{J}^p(\emptyset)$, $\bar{J}^p(\varepsilon)$, $\bar{J}^p(a)$, and $\bar{J}^p(F_1 \cdot F_2)$, given that $\bar{J}^p(F_1)$ and $\bar{J}^p(F_2)$ are already constructed, are defined as for $Th(E)$, splitting the state set into Q_1 and Q_2 in the obvious way. We also place the symbol $[_{t(F)} \in \Sigma_2$ on the δ_2 transition leaving the initial state of $\bar{J}^p(F)$ and $]_{t(F)}$ on the transition incoming to the final state of $\bar{J}^p(F)$ (adding a new initial and/or final state if required).

The constructions of $\bar{J}^p(F_1|F_2)$, $\bar{J}^p(F_1^*)$ and $F_1^{*?}$ are similar to those given in [3], but with output symbols added. These constructions are summarized in Figs. 4 and 5.

Theorem 2. *Let E be a regular expression with no subexpressions F^* or $F^{*?}$, such that $\varepsilon \in \mathcal{L}(F)$. Then $\bar{J}^p(E)(w) = \overline{m}_E(w)$ for $w \in \mathcal{L}(E)$.*

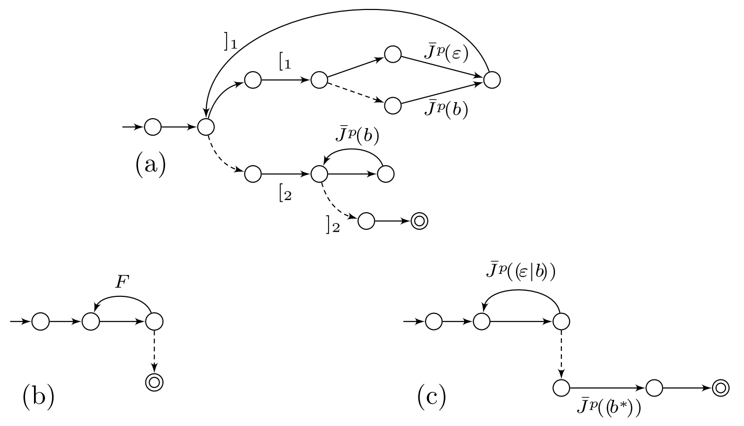

Fig. 6. (a) Java based pTr for $(\varepsilon\,|\,b)^*(b^*)$, (b) alternative F^* construction, and (c) pTr for $(\varepsilon\,|\,b)^*(b^*)$ using alternative F^* construction. Lower priority transitions are indicated by dashed edges.

Proof. First note that the condition that E has no subexpressions F^* or $F^{*?}$, with $\varepsilon \in \mathcal{L}(F)$, implies that $\bar{J}^p(E)$ contains no ε-loops, and thus when ignoring priorities of δ_2 transitions in $\bar{J}^p(E)$, we obtain by induction on the number of subexpressions in E, that $M_E(w)$ is the set of all possible outputs produced by $\bar{J}^p(E)$ on input w. Also, when ignoring the condition that $\bar{J}^p(E)$ needs to read all w before accepting and producing output (but still taking priorities of δ_2 transitions into account), we obtain by induction on the number of subexpressions in E, that $\bar{J}^p(E)$ produces as output, when considering all input paths in order of priority, the matches of prefixes of w, in the same order as defined in Definition 8. Thus once we take into account that $\bar{J}^p(E)(w)$ is the output produced by $\bar{J}^p(E)$ by the highest priority path of $\bar{J}^p(E)$ when reading all of w as input, we have that $\bar{J}^p(E)(w) = \overline{m}_E(w)$. $\square$

This theorem and proof may be adapted to the modifications discussed in Remarks 3 and 4 in a rather direct way. The induction will simply have to take into account the potential additional bracketed groups containing the empty string.

Example 3. In Fig. 6(a), a pTr T for the regular expression $(\varepsilon\,|\,b)^*(b^*)$ is given. This regular expression has a subexpression F^*, such that F matches ε. This is the so called problematic case in regular expressions matching, briefly discussed above and in [16]. In this example, the subexpression $(\varepsilon\,|\,b)^*$ will first match only ε, and will attempt to match more of the input string only if an overall match can not be achieved. Thus for the given pTr T, we have that $T(b) = [_1]_1[_2b]_2$. Regular expression matchers, such as RE2 [7], use different matching semantics in the problematic case. The problematic case is also present in regular expressions with no explicit ε symbols, such as $(a^*\,|\,b)^*(b^*)$. In Fig. 6(c) a pTr is given again for $(\varepsilon\,|\,b)^*(b^*)$, but this time obtained by using the modified greedy Kleene closure construction in Fig. 6(b). Note that $T'(b) = [_1b]_1[_2]_2$, which corresponds to how RE2 only matches non-empty words with F (where $F = (a^*\,|\,b)$ in our example) in a subexpression F^* (if F is used in a match at all).

5. A normal form for prioritized transducers

To simplify later constructions for parsing and arguments for equivalence checking, we introduce flattening for pTr in this section. The main simplification obtained by flattening is that δ_2 loops such as $q_1 \to q_2 \to q_3 \to q_1$ in Fig. 1(b) are removed, making it unnecessary to require that there are no repetition of the same δ_2 transition in a subsequence of transitions without δ_1 transitions, as in Definition 4. These δ_2 loops are found in pTr obtained from problematic regular expressions, as discussed in Examples 2 and 3.

Definition 10. A pTr $T = (Q_1, Q_2, \Sigma_1, \Sigma_2, q_0, \delta_1, \delta_2, F)$ is *flattened* if $\delta_2(q) \in (\Sigma_2^* \times Q_1)^*$ for all $q \in Q_2$.

We denote by $r_T(Q_2)$ the subset of Q_2 defined by $Q_2 \cap (\{q_0\} \cup \{\delta(q,\alpha) \mid q \in Q_1, \alpha \in \Sigma_1\})$, i.e. all Q_2 states reachable from a Q_1 state in one transition, and also the state q_0 if it is in Q_2. We denote the flattened pTr constructed in the proof of the next theorem, and equivalent to T, by $\mathrm{flat}(T)$.

Theorem 3. $\mathrm{flat}(T)$ *can be constructed in time* $\mathcal{O}(|Q_1||\Sigma_1| + |r_T(Q_2)||T|_{\delta_2})$.

Proof. We start with some preliminaries required to define $\mathrm{flat}(T)$. For a pTr T, a sequence $p_1 \cdots p_n$ is a δ_2-*path* if $\delta_2(p_i) = \cdots (w_{i+1}, p_{i+1}) \cdots$ for all $1 \le i < n$ and $(p_i, p_{i+1}) = (p_j, p_{j+1})$ only if $i = j$. The string $w_2 \cdots w_n$, obtained from the definition of a δ_2-path, is denoted by $o_T(p_1 \cdots p_n)$. Let P_T be the set of δ_2-paths. For $p_1 \cdots p_n, p'_1 \cdots p'_m \in P_T$ having

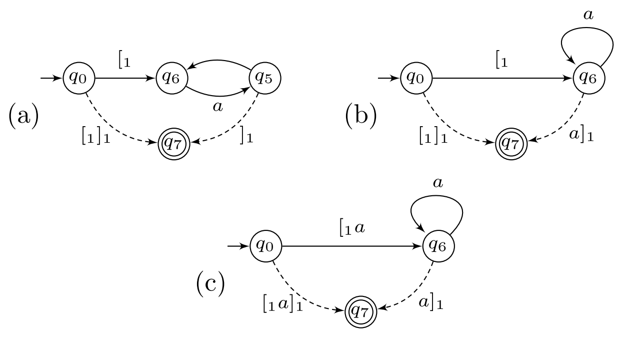

Fig. 7. (a) The flattened pTr flat(T) for T in Fig. 1b, (b) flat(T) with all but initial ε-transitions removed, and (c) flat(T) with all ε-transitions removed. The dashed edges have lower priority.

$p_1 = p'_1$, we define $p_1 \cdots p_n > p'_1 \cdots p'_m$ if and only if either (i) $p'_1 \cdots p'_m$ is a proper prefix of $p_1 \cdots p_n$, or (ii) the least i such that $p_i \neq p'_i$ is such that $\delta_2(p_{i-1}) = \cdots p_i \cdots p'_i \cdots$. Note this is similar to the definition of priorities of paths in Definition 4, but in this case restricted to δ_2-paths, and allowing any starting state in Q_2. Let $P_{q,q'} = \max\{p_1 \cdots p_n \in P_T \mid p_1 = q, p_n = q'\}$, that is, the highest priority δ_2-path from q to q' (if it exists).

We let flat(T) be $(Q_1, r_T(Q_2), \Sigma_1, \Sigma_2, q_0, \delta_1, \delta'_2, F)$, where δ'_2 is defined as follows. For $q \in Q'_2$, let $P_{q,q_1} < \cdots < P_{q,q_n}$ be all highest-priority δ_2-paths which end in a state $q_i \in Q_1$, ordered according to priority. We define $\delta'_2(q) := (o_T(P_{q,q_1}), q_1) \cdots (o_T(P_{q,q_n}), q_n)$. To compute δ'_2, with duplicate tuples removed as in Remark 2, in time $O(|r_T(Q_2)||T|_{\delta_2})$, repeat the following procedure for each $q \in r_T(Q_2)$: Determine the highest priority δ_2-path starting at q and ending in a state in Q_1. If the ending state is $q_1 \in Q_1$ (determining P_{q,q_1}), remove q_1 and all transitions going to or coming from q_1 from T, to obtain the pTr T_{q_1}. Repeat the procedure in T_{q_1}, successively finding all $P_{q,q'}$ with q fixed, in order. Note that computing $r_T(Q_2)$ takes $\mathcal{O}(|Q_1||\Sigma_1|)$ time. $\square$

Example 4. For the pTr T corresponding to the regular expression $(a^*)^*$ and discussed in Example 2, the pTr flat(T) is given in Fig. 7(a). Note that the flattening procedure removed the δ_2 loop $q_1 \to q_2 \to q_3 \to q_1$ from T. As noted in Example 2, Java matchers do not keep track of all δ_2 transitions in order to avoid repeated δ_2 transitions. When using this Java way of determining which paths are legal and which not, the flattened procedure in the proof of Theorem 3 can be modified, and when applied to T, we obtain an almost identical flattened pTr, but with output $]_1\ [_1\ ]_1$ on the transition from q_5 to q_7.

Remark 5. Note that $|\text{flat}(T)|_Q \leq |T|_Q$, $|\text{flat}(T)|_{\delta_1} = |T|_{\delta_1}$ and $|\text{flat}(T)|_{\delta_2} \leq |T|_{\delta_2}$, i.e. flat($T$) is of the same size or smaller than T.

Remark 6. All Q_2 states, with the exception of q_0 when $q_0 \in Q_2$, can be removed from a flattened pTr. To see this, redefine δ_2 to be the identity on Q_1 and let $\delta = \delta_2 \circ \delta_1$. Thus we can redefine a pTr to have a single transition function $\delta : Q_1 \times \Sigma_1 \to (Q_1 \times \Sigma^*)^*$, with $\Sigma := \Sigma_1 \cup \Sigma_2$ (except for the transitions from q_0 if $q_0 \in Q_2$), if we are willing to allow prioritized non-determinism on input from Σ_1. This procedure is applied to the pTr in Fig. 7(a), to obtain the equivalent pTr in Fig. 7(b). Since Σ_1 and Σ_2 are disjoint, it is enough to only denote the output on transitions as in Fig. 7(b). The input is obtained by deleting all symbols from Σ_2. In the situation where all Q_2 states have been removed, with the exception of $q_0 \in Q_2$, we can even consider q_0 to be in Q_1, at the expense of potentially not producing the correct output on input ε, by transitioning from q_0 to $\delta(\delta_2(q_0))$, with input determined by δ, and output by $\delta \circ \delta_2$. This procedure is applied to Fig. 7(b) in order to obtain Fig. 7(c).

6. Parsing with and equivalence of prioritized transducers

Some specifics of real-world matchers can be generalized away, for example, the property that the Σ_2 subsequences of real-world pTr outputs are always a member of a Dyck language (as in Section 4). One which we need to consider however, as including it saves memory in the parsing algorithm, is that the strings in range(T) are not output in practice, but rather matchers will walk through a string w in dom(T), and will *once the string has been accepted*, output for each symbol in $\alpha \in \Sigma_2$ in $T(w)$, the *index* of the *last* occurrence of α in $T(w)$. This limits the possible memory usage, since it implies that the space required by the output when matching an input string w with T, is bounded by $|\Sigma_2|\log(|w|)$.

Definition 11. For a pTr T with $w \in \text{dom}(T)$, let $T(w) = v_0\alpha_1 \cdots v_{n-1}\alpha_n v_n$, where $v_i \in \Sigma_2^*$ and $\alpha_i \in \Sigma_1$ for each i. Then the *slim parse output* of T on input w is the function $s^T : \Sigma_2 \to \{\bot, 0, \ldots, n\}$ such that for each $\beta \in v_0 \cdots v_n$ we have $\beta \in v_{s^T(\beta)}$, but $\beta \notin v_{s^T(\beta)+j}$ for $j \in \mathbb{N}$. If $\beta \notin v_0 \cdots v_n$, we let $s^T(\beta) = \bot$.

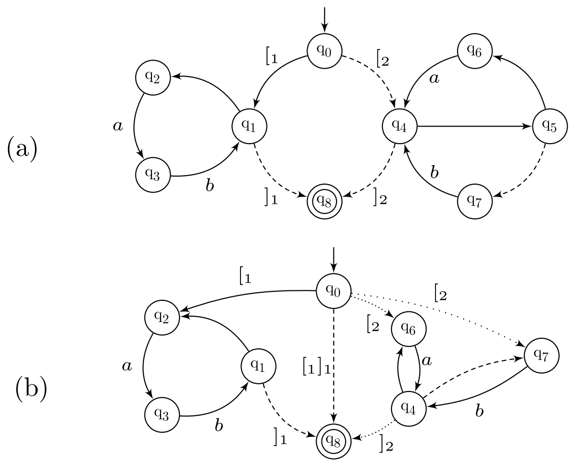

Fig. 8. (a) pTr and (b) Flattened pTr for $[_1(ab)^*]_1 \mid [_2(a|b)^*]_2$. The dashed edges are lower priority. Also, in (b), $\delta_2(q_0) = ([_1, q_2)\,([_1]_1, q_8)\,([_2, q_6)\,([_2, q_7)$ and $\delta_2(q_4) = (\varepsilon, q_6)\,(\varepsilon, q_7)\,(]_2, q_8)$.

Let T be a flattened pTr and $w = \alpha_1 \cdots \alpha_n \in \mathrm{dom}(T)$. We next describe a linear time (in the length of the input string) algorithm to compute the slim parse output of T.

For $f : \Sigma_2 \to \mathbb{N} \cup \{\bot\}$, $v \in \Sigma_2^*$ and $k \in \mathbb{N}$, define $F_{f,v,k} : \Sigma_2 \to \mathbb{N} \cup \{\bot\}$ by letting $F_{f,v,k}(\beta) = k$ if $\beta \in v$, and $F_{f,v,k}(\beta) = f(\beta)$ otherwise. Also, for $q \in Q_1 \cup Q_2$, $f : \Sigma_2 \to \mathbb{N} \cup \{\bot\}$ and $i \in \{0, \dots, n\}$, let

$$\Delta(q, f, i) = \begin{cases} (q, f) & \text{if } q \in Q_1; \\ (q_1, F_{f,\beta_1,i}) \cdots (q_n, F_{f,\beta_n,i}) & \text{if } q \in Q_2, \delta_2(q) = (\beta_1, q_1) \cdots (\beta_n, q_n). \end{cases}$$

The steps in the parsing algorithm are given next.

1. Let $S_0 = \Delta(q_0, f_\bot, 0)$.
2. Given $S_i = (q_1, f_1) \cdots (q_m, f_m)$, where $i < n$, then S_{i+1} is constructed as follows:
   (a) Let $q'_j = \delta_1(q_j, \alpha_i)$ for each j.
   (b) Then let $S_{i+1} = \sigma_1(\Delta(q'_1, f_1, i) \cdots \Delta(q'_m, f_m, i))$.
3. Let $S_n = (q_1, f_1) \dots (q_m, f_m)$ and j the smallest index such that $q_j \in F$. Then f_j is the slim parse output. If no state q_i is in F, then the string w is rejected.

Example 5. In this example we illustrate the slim parsing algorithm on the regular expression $((ab)^*)|((a|b)^*)$ with input *aba*. Since we capture input matched by the subexpressions $(ab)^*$ and $(a|b)^*$, we rewrite the regular expressions as $[_1(ab)^*]_1 \mid [_2(a|b)^*]_2$. The pTr, constructed as in Section 4, but with some irrelevant δ_2 transitions removed, and equivalent flattened pTr, for this regular expression, is given in Figs. 8(a) and (b). The slim parsing algorithm is applied on the flattened pTr. The functions $f_{i,j} : \{[_1,]_1, [_2,]_2\} \to \{\bot, 0, 1, 2, 3\}$, for $i, j \in \{0, 1, 2, 3\}$, correspond to the functions f_i given in the equality $S_j = (q_1, f_1) \cdots (q_m, f_m)$. Next we give the values S_0, S_1, S_2 and S_3 for this particular example.

$$\begin{array}{lcl} S_0 &=& \Delta(q_0, f_\bot, 0) &=& (q_2, f_{0,0})(q_8, f_{0,1})(q_6, f_{0,2})(q_7, f_{0,3}) \\ S_1 &=& \sigma_1(\Delta(q_3, f_{0,0}, 1)\Delta(q_4, f_{0,2}, 1)) &=& (q_3, f_{1,0})(q_6, f_{1,1})(q_7, f_{1,2})(q_8, f_{1,3}) \\ S_2 &=& \sigma_1(\Delta(q_1, f_{1,0}, 2)\Delta(q_4, f_{1,2}, 2)) &=& (q_2, f_{2,0})(q_8, f_{2,1})(q_6, f_{2,2})(q_7, f_{2,3}) \\ S_3 &=& \sigma_1(\Delta(q_3, f_{2,0}, 3)\Delta(q_4, f_{2,2}, 3)) &=& (q_3, f_{3,0})(q_6, f_{3,1})(q_7, f_{3,2})(q_8, f_{3,3}) \end{array}$$

In S_3, q_8 is the only accepting state in $(q_3, f_{3,0})(q_6, f_{3,1})(q_7, f_{3,2})(q_8, f_{3,3})$. Also, $f_{3,3}([_1) = f_{3,3}(]_1) = \bot$, $f_{3,3}([_2) = 0$ and $f_{3,3}(]_2) = 3$, indicating that the subexpression $(a|b)^*$ captured all of the input. Also, note that $f_{3,0}([_1) = 0$ and $f_{3,0}(]_1) = f_{3,0}([_2) = f_{3,0}(]_2) = \bot$, which is a description of the capturing information related to the subexpression $(ab)^*$, which goes unused with input *aba*, but which will be updated and used if another b is added to the input *aba*, in order to indicate that the input *abab* is captured completely by the subexpression $(ab)^*$.

Theorem 4. *The slim parsing algorithm runs in linear time in the length of the input string, and is correct, i.e. with input a pTr T and $w \in \mathrm{dom}(T)$, it returns the slim parse for T on w, and if $w \notin \mathrm{dom}(T)$, it rejects the input w. Additionally, the algorithm uses $\mathcal{O}(|Q_1||\Sigma_2|\log|w|)$ space.*

Proof. The reasoning behind the algorithm is straightforward enough that excessive detail is unnecessary. It is obvious that the depth-first search approach is correct, and memoization can solve the performance issues.

The highest-priority accepting path in a flattened pTr may be determined by depth first search by simply in each step attempting the options in order of priority, attempting lower priority choices only when backtracking. Matching input with a flattened pTr can be done with a machine similar to a deterministic stack machine with output. In contrast to a stack machine, the given algorithm simply processes all stack elements in parallel. Each S_i can be associated with the stack content at a particular stage of the depth first (when all stack elements are processed in parallel), with the left-most tuple (of S_i) being the top element of the stack. It follows from the argument in Remark 2 that duplicates of tuples with the same state can be removed from the stack, and thus the number of elements on the stack is bounded by $|Q_1|$. Clearly, from the description of the slim parsing algorithm, it runs in linear time in the length of the input string.

A map is attached to each state on the stack, recording the string position at which each symbol from Σ_2 was last output (overwriting old values when a symbol is output repeatedly). At the end of the algorithm, the first map from the top, belonging to an accepting state, is produced as output of the slim parsing algorithm. Storing the states and maps accounts for the $\mathcal{O}(|Q_1||\Sigma_2|\log|w|)$ space usage (at most $|Q_1|$ states on the stack, and each state contains at most $|\Sigma_2|$ string positions). $\square$

By way of comparison, depth-first search takes exponential time in the worst-case [3], and uses $\mathcal{O}(|w|)$ space (or more precisely, $\mathcal{O}(|w|\log|Q|)$ if we also take the space required to store an individual state into account) for storing the stack (for flattened pTr) of states on the currently explored path. The capturing positions in the input string, for each subexpression, can be derived from the states on the stack once the input string is accepted.

Remark 7. It is not difficult to modify the slim parsing algorithm to produce "full" parsing output, recording instead the string positions of *all* instances of output being produced, but obviously at the expense of increased memory usage. This functionality is for example available in the regular expression matching library of `C#`, by using the `CaptureCollection` class [13].

Specifically, the space requirement is increased to $\mathcal{O}(|Q_1||\Sigma_2||w|)$, since to each state on the stack is attached a table $t : \Sigma_2 \times \{0, \dots, |w|\} \to \{\mathit{true}, \mathit{false}\}$, where $t(\alpha, i)$ is true if and only if α is output at position i (in practice this table is likely to be sparse, making more efficient storage possible). This does make parsing more expensive in terms of space usage, but parsing time stays linear in the length of the input string.

Regular expressions R and R' are equivalent if the pTr $\bar{J}^p(R)$ and $\bar{J}^p(R')$ are equivalent. In general, deciding equivalence of string transducers is undecidable, but in [15] it is shown that equivalence of functional transducers is decidable, but PSPACE-complete. In [16], the equivalence of regular expressions through transducers, is approached by first formulating the semantics of regular expression matching as a non-deterministic parser, then transforming the parser into first a transducer with regular lookahead, and then into a functional transducer without lookahead. For non-problematic (a subset of expressions without some complicating properties) regular expressions R, a functional transducer of size $2^{\mathcal{O}(|R|)}$ is obtained. Thus to decide equivalence of regular expressions with capturing groups, equivalence is decided on the corresponding functional transducers. We obtain a similar result for a larger class of regular expressions and regular expression matching semantics, through equivalence of pTr.

Theorem 5. *A pTr T can be converted into an equivalent functional transducer T_F with $|T_F|_Q = |T|_Q 2^{|T|_Q}$, and $|T_F|$ in $\mathcal{O}(|T|2^{|T|_Q})$.*

Proof. The construction used in this proof relies on the realization that prioritization can be simulated using regular lookahead. That is, we construct a transducer which on each transition additionally looks ahead to verify that no alternative transition of higher priority would lead to an accepting run. As such we construct the transducer T_F by taking the product of T and the power set of nfa(pnfa(T)), using the latter to effectively look-ahead with respect to all higher-priority choices ignored.

Take $T = (Q_1, Q_2, \Sigma_1, \Sigma_2, q_0, \delta_1, \delta_2, F)$ and let $A = (Q_A, \Sigma_1, q_{0,A}, \delta_A, F_A)$ be the NFA nfa(pnfa(T)) (recalling the constructions in Section 2). Then we construct $T_F = (Q, \Sigma_1, \Sigma_2, q'_0, \delta, F')$ as follows.

- Let $Q = (Q_1 \cup Q_2) \times 2^{Q_A}$. The first component of a state in Q represents a state in T, while the second component is the set of higher priority states which must not accept the input string.
- Let $q'_0 = (q_0, \emptyset)$. That is, start in the initial state of T with no higher priority untaken paths yet.
- $F' = F \times \{X \mid X \subseteq (Q_A \setminus F)\}$. Accept if a state in T accepts, and none of the higher-priority ones do (i.e. this is where priority is checked).
- The transition function δ is defined next, where $\overline{U}$ denotes the epsilon closure when $U \subseteq Q_A$.
  - For the δ_1 transition, for all $(q, P) \in Q$ and $\alpha \in \Sigma_1$ with $\delta_1(q, \alpha) = q'$, we have $((q, P), \alpha, \varepsilon, (q', \overline{(\delta_A(P, \alpha))})) \in \delta$. Thus on δ_1 transitions, all states are simply stepped forwards accordingly, where the function δ_A is extended to be defined on sets of states contained in Q_A, in the obvious way.
  - For $(q, P) \in Q$ with $\delta_2(q) = (w_1, q_1) \cdots (w_n, q_n)$, and each $i \in \{1, \dots, n\}$, we have $((q, P), \varepsilon, w_i, (q_i, P')) \in \delta$, where $P' = P \cup \overline{\{q_1, \dots, q_{i-1}\}}$. That is, the transducer may pick any of the states in a δ_2 transition, but records all higher-priority choices to prove they do not accept.

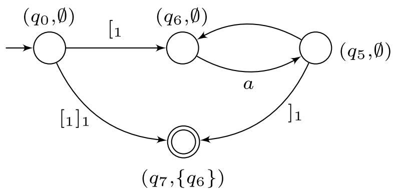

Fig. 9. The functional transducer T_F obtained by applying the procedure described in the proof of Theorem 5 to the pTr flat(T) in Fig. 7(a).

The proof that T and T_F defines the same relation (which is in fact a partial function), is obtained by induction on the length of an input string. Also recall from Theorem 3 that we may assume that T is flattened. Whenever T is in state q on an accepting run, T_F would be in (q, P), where P represents all states reachable on the input up to that point on paths that have higher priority than the one chosen. Note that T_F is a functional transducer, since the relation defined by T is a partial function. The size claims follows directly from the construction of T_F from T. □

Example 6. In Fig. 9 we show the functional transducer T_F obtained by applying the procedure described in the proof of the previous theorem to the pTr flat(T) in Fig. 7(a).

Corollary 1. *For prioritized transducers T_1 and T_2, equivalence can be decided in time $\mathcal{O}((|T_1|2^{|T_1|_Q} + |T_2|2^{|T_2|_Q})^2)$.*

Proof. For pTr T_1 and T_2, first check that dom$(T_1) =$ dom(T_2), which can done in time $\mathcal{O}(|\Sigma_1|(2^{|T_1|_Q} + 2^{|T_2|_Q}))$, by determinizing the NFA obtained when ignoring the output and priorities of T_1 and T_2, and then testing equivalence of the obtained DFAs (see [1] for a discussion of the Hopcroft and Karp algorithm for testing equivalence of DFA).

Now convert T_1 and T_2 into functional transducers T_{F_1} and T_{F_2}, and use Theorem 1.1 in [15] that states that the complexity of deciding if the transducer $T_1 \cup T_2$ is functional (and thus that T_1 and T_2 are equivalent, since dom$(T_1) =$ dom(T_2)), is quadratic in the number of transitions in $T_1 \cup T_2$. □

Corollary 2. *Equivalence of regular expressions R_1 and R_2 with capturing groups, can be decided in time $2^{\mathcal{O}(|R_1|+|R_2|)}$.*

Proof. Follows from Corollary 1 combined with the fact that for a regular expression R, $|\bar{J}^p(R)| \le c|R|$, for some constant c. □

Remark 8. Note that deciding equivalence of pTr and regular expressions with capturing groups is at least PSPACE-hard, since it is PSPACE-complete already to check if the domains of pTr are equal.

Remark 9. The transducer construction in the proof of Theorem 5 must be close to optimal, as the worst-case state complexity of a transducer T_F equivalent to a pTr T is bounded from below by $2^{|T|_Q}$. This is so since one can for any NFA A, construct a pTr T, with $\Sigma_2 = \{\beta, \beta'\}$, having $T(w) = \beta w$ for $w \in \mathcal{L}(A)$ and $T(w) = \beta' w$ for all $w \notin \mathcal{L}(A)$ (a similar example is obtained by constructing a pTr for the regular expression $(R)|(\Sigma_1^*)$, with R corresponding to A). Simply let $\delta_2(q_0) = (\beta, q_A)(\beta', q_{\Sigma_1^*})$, with q_0 the initial state of T, q_A the initial state of A, and $q_{\Sigma_1^*}$ a sink accept state. Now consider the class of NFA A, for which the complement of $\mathcal{L}(A)$ can only be recognized by NFA with at least $2^{|A|_Q}$ states. Then if T_F' is obtained from T_F by removing transitions having β as output, dom(T_F') will be equal to the complement of $\mathcal{L}(A)$. Thus T_F' and also T_F, will require at least $2^{|A|_Q}$ states.

7. Conclusions and future work

In this paper we brought together several different angles on regular expressions into one formal framework. The concept of prioritized transducers allows the modeling of some of the special features of real-world regular expression matchers (especially the Java matcher), without being tied to their algorithmic choices. Still, there is ample room for continued work. For example, there is a plethora of additional operators in regex libraries that should be analyzed. A special example is pruning operators, such as atomic subgroups, and the cut operator of [2], which interact deeply with the matching procedure. While POSIX holds lesser promise, a more thorough treatment of the most popular interpretations of the POSIX semantics, in the same context, would also be of interest.

Acknowledgements

We thank the reviewers for their constructive input.

References

[1] Marco Almeida, Nelma Moreira, Rogério Reis, Testing the equivalence of regular languages, J. Autom. Lang. Comb. 15 (1/2) (2010) 7–25.

[2] Martin Berglund, Henrik Björklund, Frank Drewes, Brink van der Merwe, Bruce Watson, Cuts in regular expressions, in: Marie-Pierre Béal, Olivier Carton (Eds.), Developments in Language Theory, in: Lecture Notes in Computer Science, vol. 7907, Springer, 2013, pp. 70–81.

[3] Martin Berglund, Frank Drewes, Brink van der Merwe, Analyzing catastrophic backtracking behavior in practical regular expression matching, in: Zoltán Ésik, Zoltán Fülöp (Eds.), Automata and Formal Languages, in: Electronic Proceedings in Theoretical Computer Science, vol. 151, 2014, pp. 109–123.

[4] Martin Berglund, Brink van der Merwe, On the semantics of regular expression parsing in the wild, in: Frank Drewes (Ed.), Implementation and Application of Automata, in: Lecture Notes in Computer Science, vol. 9223, Springer, 2015, pp. 292–304.

[5] Angelo Borsotti, Luca Breveglieri, Stefano Crespi-Reghizzi, Angelo Morzenti, From ambiguous regular expressions to deterministic parsing automata, in: Frank Drewes (Ed.), Implementation and Application of Automata, in: Lecture Notes in Computer Science, vol. 9223, Springer, 2015, pp. 35–48.

[6] Cezar Câmpeanu, Kai Salomaa, Sheng Yu, A formal study of practical regular expressions, Internat. J. Found. Comput. Sci. 14 (6) (2003) 1007–1018.

[7] Russ Cox, Implementing regular expressions, http://swtch.com/~rsc/regexp/, 2007, accessed November 9, 2015.

[8] Glenn Fowler, An interpretation of the POSIX regex standard, http://www2.research.att.com/~astopen/testregex/re-interpretation.html, 2003, accessed November 9, 2015.

[9] Jeffrey Friedl, Mastering Regular Expressions, third edition, O'Reilly Media, Inc., Sebastopol, CA, USA, 2006.

[10] Alain Frisch, Luca Cardelli, Greedy regular expression matching, in: Josep Díaz, Juhani Karhumäki, Arto Lepistö, Donald Sannella (Eds.), International Colloquium on Automata, Languages and Programming, in: Lecture Notes in Computer Science, vol. 3142, Springer, 2004, pp. 618–629.

[11] The IEEE and The Open Group, The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html, 2013, accessed November 9, 2015.

[12] John Maddock, Boost.Regex 5.0.1, http://www.boost.org/doc/libs/1_59_0/libs/regex/doc/html/boost_regex/background_information/faq.html, 2013, accessed November 9, 2015.

[13] Microsoft Developer Network, CaptureCollection class, https://msdn.microsoft.com/en-us/library/system.text.regularexpressions.capturecollection(v=vs.110).aspx, 2015, accessed November 9, 2015.

[14] Asiri Rathnayake, Hayo Thielecke, Static analysis for regular expression exponential runtime via substructural logics, Computing Research Repository, arXiv:1405.7058, 2014.

[15] Jacques Sakarovitch, Elements of Automata Theory, Cambridge University Press, New York, NY, USA, 2009.

[16] Yuto Sakuma, Yasuhiko Minamide, Andrei Voronkov, Translating regular expression matching into transducers, J. Appl. Log. 10 (1) (2012) 32–51.

[17] Ken Thompson, Regular expression search algorithm, Commun. ACM 11 (6) (1968) 419–422.

[18] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, Bruce Watson, Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA, in: Yo-Sub Han, Kai Salomaa (Eds.), Implementation and Application of Automata, in: Lecture Notes in Computer Science, vol. 9705, Springer, 2016, pp. 322–334.