On the Semantics of Regular Expression Parsing in the Wild

Martin Berglund[1] and Brink van der Merwe[2](✉)

[1] Umeå University, Umeå, Sweden
mbe@cs.umu.se
[2] University of Stellenbosch, Stellenbosch, South Africa
abvdm@cs.sun.ac.za

Abstract. We introduce prioritized transducers to formalize capturing groups in regular expression matching in a way that permits straightforward modelling of and comparison with real-world regular expression matching library behaviors. The broader questions of parsing semantics and performance are discussed, and also the complexity of deciding equivalence of regular expressions with capturing groups.

1 Introduction

Few formal language research results have greater practical reach than regular expressions. As a result the practical implementations [5] have in many ways surged ahead of research, with new features which require underpinnings different from the original theory. Practical implementations perform matching as a form of parsing, using *capturing groups,* outputting what subexpression matched which substring. A popular implementation strategy, still very common [2], is a worst-case EXPTIME depth-first search for such parses. A more formal approach suggests using finite transducers, outputting annotations on the string to signify the nature of the match [9]. This is complicated by the matching semantics dictating a single output string for each input string, using rules to determine a "highest priority" match among the potentially exponentially many possible ones (for contrast e.g. [3] discusses non-deterministic capturing groups).

The pNFA (prioritized non-deterministic finite automaton) model of [2] (similar results were also published mere weeks later in [7]) provides the right level of abstraction to model the matching time behavior of regular expression matchers. However, for matchers based on an input directed depth first search, it does not provide an understanding of why practical regular expression matchers often (indirectly) use the pNFA model, and in particular, there is no notion of when one pNFA is equivalent to another. By adding output to pNFA to obtain pTr (prioritized transducers), we obtain a better understanding of the usefulness of the prioritized automata/transducer model, and we also have the notion of equivalence of pTr, which is defined in terms of equality of the underlying functions represented by the pTr. A regular expression to transducer construction is done in [9], but it is remarked that translating regular expression matching directly

F. Drewes (Ed.): CIAA 2015, LNCS 9223, pp. 292–304, 2015.
DOI: 10.1007/978-3-319-22360-5_24

into transducers is highly non-trivial. In Sect. 3, where we discuss conversion from regular expression to pTr, it will become clear that pTr are a perfect fit when converting regular expressions to transducers. We also discuss a linear-time matching algorithm for pTr (i.e. determining the image of input strings), generalizing e.g. [6] which operates directly on expressions, and mirroring work by Russ Cox [4] which is to a great extent not formally published.

The outline of the paper is as follows. In the next section, we define prioritized automata and transducers. After that, we show how to adapt the standard Thompson construction, from [10], for converting regular expressions to non-deterministic finite automata, to the more general setting of converting regular expressions to prioritized transducers. This is followed by a normal form for prioritized transducers, so called flattened prioritized transducers, that simplifies the following discussions on deciding equivalence of and parsing with pTr.

2 Definitions

Let dom(f) and range(f) denote the domain and range of a function f respectively. When unambiguous let a function f with dom(f) $= S$ generalize to S^* and $\mathcal{P}(S)$ element-wise. By $\mathcal{P}(S)$, we denote the power set of a 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. $\pi_{\{a,b\}}(abcdab) = abab$). 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 are 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 will be 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 $\varepsilon, \emptyset \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \mid E')$, $(E \cdot E')$, 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). The matching procedure will also produce information about which substring(s) are matched by each capturing group. Thus brackets in regular expressions are used both for precedence and capturing, and in Java[1] a

[1] Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

non-capturing subexpression E is indicated by $(?{:}E)$. The precise matching and capturing semantics follow from Sect. 3. When we say that E matches a string w we mean that all of w is read by E, as opposed to $vwv' \in \mathcal{L}(E)$, for $v, v' \in \Sigma^*$, as some implementations do. This substring matching can be simulated in our model with the expression $.^{*?}(R).^*$, where $E^{*?}$ and E^* denotes respectively the lazy and greedy (both to be defined in Sect. 3) Kleene closure of E.

For later constructions we require a few different kinds of automata and transducers. First (non-)deterministic finite automata (and runs for them), followed by the prioritized finite automata from [2], which are used to model the regular expression matching behaviors exhibited by typical software implementations.

Definition 1. *A* non-deterministic finite automaton *(NFA) is a tuple $A = (Q, \Sigma, q_0, \delta, F)$ where: (i) Q is a finite set of* states; *(ii) Σ is the* input alphabet; *(iii) $q_0 \in Q$ is the* initial state; *(iv) $\delta : Q \times \Sigma^\varepsilon \to \mathcal{P}(Q)$ is the* transition function; and *(v) $F \subseteq Q$ 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 $|\delta(q, \alpha)| \leq 1$ for all q and α. The* state size *of A is denoted by $|A|_Q$, and defined to be $|Q|$.*

Definition 2. *For a 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 $0 \leq i < n$, 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 subset $\{\pi_\Sigma(r) \mid r$ is an accepting run in $A\}$ of Σ^*.*

Now for the prioritized NFA variant, as defined in [2].

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

Remark 4. For a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ the *corresponding finite automaton* nfa(A) is given by nfa$(A) = (Q_1 \cup Q_2, \Sigma, q_0, \bar{\delta}, F)$, where $\bar{\delta}(q, \alpha) = \{\delta_1(q, \alpha)\}$ if $q \in Q_1$, and $\bar{\delta}(q, \varepsilon) = \{q_1, \ldots, q_n\}$ if $q \in Q_2$ with $\delta_2(q) = q_1 \ldots q_n$.

Next we define runs for pNFA. An accepting run for a string w in a pNFA A, is defined to be the highest priority accepting run of w in 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 in nfa(A) for a given input string w, when determining the highest priority path (referred to as a run in 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 5. *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 nfa(A), such that if $\alpha_i = \alpha_{i+1} = \ldots = \alpha_{j-1} = \alpha_j = \varepsilon$, with $i \leq j$, then $(s_{k-1}, s_k) = (s_{l-1}, s_l)$, with $i \leq k, l \leq j$, 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 $p \neq p'$, $\pi_\Sigma(p) = \pi_\Sigma(p')$ and either p' is a proper prefix of p, or if j is the first index such that $s_j \neq s'_j$, 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 $\{\pi_\Sigma(r) \mid r$ is an accepting run in $A\}$. Note that $\mathcal{L}(A) = \mathcal{L}(nfa(A))$.*

Our definition of pNFA, compared to the one in [2], is slightly less general, since we assume that $F \subseteq Q_1$, instead of $F \subseteq Q$. This restriction was introduced to simplify our definitions, and the more general pNFA can be converted to pNFA with $F \subseteq Q_1$, by introducing one new state $q_F \in Q_1$ and δ_2 transitions from the old accepting states $q \in Q_2$ to q_F, where we give the new δ_2 transitions for example the highest priority of all δ_2 transitions at q. In [7], an ordered NFA, very similar to our definition of pNFA, is defined, with a single set of states Q and a transition function $\delta : Q \times \Sigma \to Q^*$. We can simulate this with our pNFA, by decomposing $q \mapsto \delta(q, a)$ into $q \mapsto^{\delta_1(\_,a)} q_a \mapsto^{\delta_2(\_)} \delta(q, a)$, where $q \in Q_1$ and $q_a \in Q_2$. Note that we introduced pNFA (and runs in pNFA in Definition 5) mainly as an aid in defining runs in prioritized transducers in Definition 9 below.

Example 6. In Fig. 1(a), a Java based pNFA A for the regular expression $(a^*)^*$, constructed as described in [2], is given. The accepting run for the string a^n, in A, is $q_0 q_1 (q_2 a q_1)^n q_0 q_3$. Since there are for the input strings a^n, $n \geq 0$, exponentially many paths in A, a regular expression matcher using an input directed depth first search (without memoization as in Perl), such as the Java implementation, will take exponential time to attempt to match the strings $a^n x$, for $n \geq 0$.

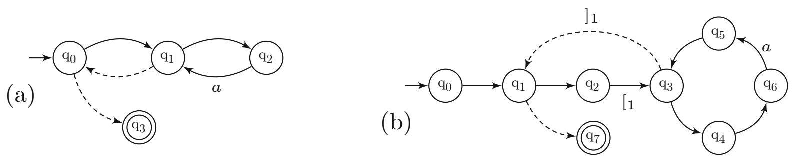

Fig. 1. (a) Java based pNFA for the regular expression $(a^*)^*$, i.e., the pNFA $A = (\{q_2, q_3\}, \{q_0, q_1\}, \{a\}, q_0, \{(q_2, a, q_1)\}, \{[q_0, (q_1, q_3)], [q_1, (q_2, q_0)]\}, q_3)$. (b) Java based pTr with $\Sigma_1 = \{a\}$ and $\Sigma_2 = \{[_1,]_1\}$, for $(a^*)^*$. Lower priority transitions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

Recall that a transducer T (see for example [11], Definition 3.1) 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 the input and output alphabets respectively, $\delta \subseteq Q \times \Sigma_1^\varepsilon \times \Sigma_2^* \times Q$ the set of transitions, q_0 the initial state and F 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 run). As usual, we denote by $\mathrm{dom}(T)$ the set $\{v \in \Sigma_1^* \mid (v, w) \in \mathcal{R}(T)\}$, and by $\mathrm{range}(T)$ the set $\{w \in \Sigma_2^* \mid (v, w) \in \mathcal{R}(T)\}$. For functional transducers (see [8], 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 $\mathcal{R}(T)$, $\mathrm{dom}(T)$, $\mathrm{range}(T)$, and $T(v) = w$ if $(v, w) \in \mathcal{R}(T)$, will thus also be used.

Definition 7. *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: (i)*Q_1 *and* Q_2 *are disjoint finite sets of* states; *(ii)* Σ_1 *is the* input alphabet; *(iii)*Σ_2*, disjoint from* Σ_1*, is the* group identifier *or* output *alphabet; (iv)*$q_0 \in Q$ *is the* initial state; *(v)*$\delta_1 : Q_1 \times \Sigma_1 \to Q$ *is the* deterministic*, but not necessarily total,* transition function; *(vi)*$\delta_2 : Q_2 \to (\Sigma_2^* \times Q)^*$ *is the* non-deterministic prioritized transition and output function; *and (vii)*$F \subseteq Q_1$ *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* $|T| := |T|_Q + |T|_\delta$*.*

Remark 8. It is only in Sect. 3, when we construct pTr from regular expressions, where the assumption $\Sigma_1 \cap \Sigma_2 = \emptyset$ is required.

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 make 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, which is a subset of Σ_1^*, to produce output by *decorating* the input string with symbols from Σ_2. For a pTr $T = (Q_1, Q_2, \Sigma_1, \Sigma_2, q_0, \delta_1, \delta_2, F)$, $\mathrm{pnfa}(T)$ is the pNFA $(Q_1, Q_2, \Sigma_1, q_0, \delta_1, \delta_2', F)$ obtained from T 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 in $\mathrm{pnfa}(T)$ 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 9. *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 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}) (and not necessarily at position $(i+1)$ of $\delta_2(s_i)$). 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 10. 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 $\delta_2(q)$.

Example 11. In Fig. 1(b), a pTr T for the regular expressions $(a^*)^*$, constructed by the procedure described in the next section, is given. 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, 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 sequence of ε transitions, and thus in the Java case 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 the lazy and greedy Kleene closure in Figs. 2(c) and (d) in the next section. 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 $T(a^n) = [_1 a]_1 [_2 a^{n-1}]_2 \neq [_1 a^{n-1}]_1 [_2 a]_2 = T'(a^n)$, 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 $T''(a^p b^q a^r) = [_1 a^p]_1 [_1 b]_1 \dots [_1 b]_1 [_1 a^r]_1$, for $p, q, r > 0$.

3 Converting Regular Expressions into pTr

Next we give a Java based construction to turn a regular expression E into an equivalent pTr $\bar{J}^p(E)$ (refer to Fig. 2 for reference). If for a pTr T, we denote by $u(T)$ the string transducer obtained by ignoring the priorities in T, then for $w \in \mathcal{L}(E)$, $u(\bar{J}^p(E))(w)$ gives all possible ways in which E can match w with capturing information indicated, while $\bar{J}^p(E)(w)$ selects the highest priority

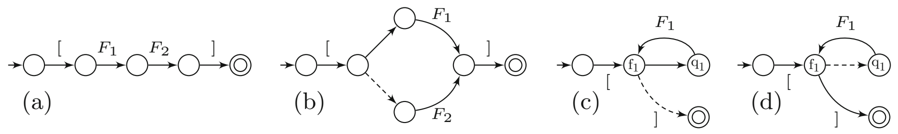

Fig. 2. Java based regular expression to pTr constructions for (a) (F_1F_2) (b) $(F_1 \,|\, F_2)$ (c) (F_1^*) and (d) $(F_1^{*?})$. Lower priority transitions are indicated by dashed edges. The pair of brackets [,] are used to indicate the substring or substrings captured by each of F_1F_2, $F_1 \,|\, F_2$, F_1^*, and $F_1^{*?}$ respectively.

match from $u(\bar{J}^p(E)(w))$. Due to space limitations, it is not possible to describe the matching semantics of regular expression with capturing groups in terms of a non-deterministic parser or other means, and show that the constructed pTr produces equivalent output, but we hope that it will at least be intuitively clear that this can be done. See also [2] for a thorough argument for the pNFA case, which may be extended to pTr with some effort. As indicated in Example 11, we opt to deviate from the Java matching semantics (and follow RE2 matching semantics [4]) in cases where the Java matcher follows a non-empty capture of a subexpression F (with F being part of a larger subexpression F^*), by an empty capture with F. Our construction is similar to the Java based regular expression to pNFA constructions given in [2] (and the classical Thompson construction [10]), with the additional detail of adding a group opening symbol on the transition leaving the initial state, and a group closing symbol on the transition incoming to the final state, for the pTr constructed for each subexpression of E. Where required, a new initial state and/or final state is added to the constructions from [2], so that there is only a single δ_2 transition from the initial state, and similarly, only a single incoming δ_2 transition to the final state of a pTr.

We denote the set of subexpressions of E by $\mathrm{SUB}(E)$. Assume $F_1, \ldots, F_k$ are the subexpressions in $\mathrm{SUB}(E)$, 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 if the same subexpression appears more than once in E, we regard these occurrences as distinct elements in $\mathrm{SUB}(E)$. Also, let $t : \mathrm{SUB}(E) \to \mathbb{N}$ be defined by $t(F_i) = i$. 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 is 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.

For a regular expression E we define a prioritized transducer $T := \bar{J}^p(E)$ such that $\mathrm{dom}(T) = \mathcal{L}(E)$, and $\mathrm{range}(T)$ is contained in the shuffle of $\mathrm{dom}(T)$ and the Dyck language D_k. In fact, taking some F which is a subexpression of E, and $v \in \mathrm{dom}(T)$, if $T(v)$ contains the substring $[_{t(F)} w]_{t(F)}$, where $[_{t(F)}$ and $]_{t(F)}$ are matching brackets, then $\pi_{\Sigma_1}(w) \in \mathcal{L}(F)$ (recall that $\pi_{\Sigma_1}(w)$ is obtained from w by deleting all brackets). Also, all output symbols from Σ_1 in $T(v)$, are between matching brackets.

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 [2] 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).

When we construct $\bar{J}^p(F_1|F_2)$ from $\bar{J}^p(F_1)$ and $\bar{J}^p(F_2)$, and $\bar{J}^p(F_1^*)$ from $\bar{J}^p(F_1)$, the priorities of newly introduced δ_2-transitions require attention. We also consider the lazy Kleene closure $F_1^{*?}$. In the constructions (i) and (ii) below, we assume $\bar{J}^p(F_i)$ ($i \in \{1,2\}$) has initial state q_i and the final state f_i. Furthermore, δ_2 denotes the prioritized transition function in the newly constructed pTr $\bar{J}^p(F)$. All non-final states in $\bar{J}^p(F)$ that are in $\bar{J}^p(F_i)$ inherit their outgoing transitions from $\bar{J}^p(F_i)$. (i) If $F = F_1|F_2$ then $\bar{J}^p(F)$ is constructed by introducing new initial and final states $q_0 \in Q_2$ and $f_0 \in Q_1$, an additional new state $q' \in Q_2$, merging the states $f_1, f_2 \in Q_1$ into a state denoted by $f \in Q_2$, and defining $\delta_2(q_0) = ([_{t(F)}, q')$, $\delta_2(q') = (\varepsilon, q_1)(\varepsilon, q_2)$ and $\delta_2(f) = (]_{t(F)}, f_0)$. (ii) If $F = F_1^*$ then we add new initial and final states $q_0 \in Q_2$ and f_0 to Q_1, and change f_1 from being a state in Q_1, to be in Q_2. We define $\delta_2(q_0) = ([_{t(F)}, f_1)$ and $\delta_2(f_1) = (\varepsilon, q_1)(]_{t(F)}, f_0)$. The case $F = F_1^{*?}$ is the same, except that $\delta_2(f_1) = (]_{t(F)}, f_0)(\varepsilon, q_1)$. Thus $\bar{J}^p(F^*)$ tries F as often as possible whereas $\bar{J}^p(F^{*?})$ does the opposite.

Example 12. In Fig. 3(a), a pTr T for the regular expression $(\varepsilon \mid b)^*(b^*)$ 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 in [9]. In this example, the subexpression $(\varepsilon \mid b)^*$ 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 [4], uses different matching semantics in the problematic case. The problematic case is also present in regular expressions with no explicit ε symbols, such as $(a^* \mid b)^*(b^*)$. In Fig. 3(c) a pTr is given again for $(\varepsilon \mid b)^*(b^*)$, but this time obtained by using the modified greedy Kleene closure construction in Fig. 3(b). Note that $T'(b) = [_1b]_1[_2]_2$, corresponding to how RE2 only matches non-empty words with F in a subexpression F^*.

4 A Normal Form for Prioritized Transducers

To simplify later constructions, 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 Example 11 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 5. These δ_2 loops are found in pTr obtained from problematic regular expressions, as discussed in Examples 11 and 12.

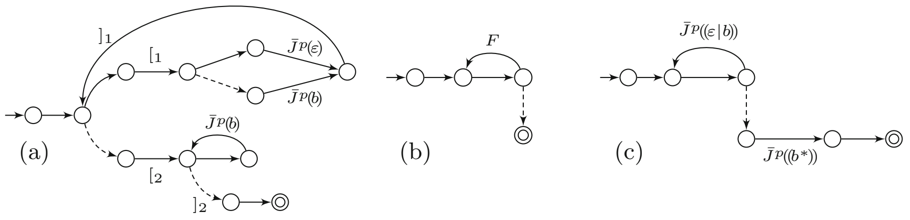

Fig. 3. (a) Java based pTr for $(\varepsilon|\, b)^*(b^*)$, (b) alternative F^* construction, and (c) pTr for $(\varepsilon|\, b)^*(b^*)$ using alternative F^* construction. Lower priority transitions are indicated by dashed edges.

Definition 13. *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 flat(T).

Theorem 14. *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 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 $p_1 = p'_1$, 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 5, 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 10, 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. □

Example 15. For the pTr T corresponding to the regular expression $(a^*)^*$ and discussed in Example 11, the pTr flat(T) is given in Fig. 4. 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 11, 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 14 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.

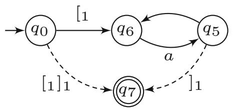

Fig. 4. flat(T) for T in Fig. 1(b). The dashed edges are lower priority.

Remark 16. 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 17. 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 $\delta : Q_1 \times \Sigma_1 \to (Q_1 \times \Sigma_2^*)^*$ (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.

5 Equivalence and Parsing with Prioritized Transducers

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 [8] it is shown that equivalence of functional transducers is decidable, but PSPACE-complete. In [9], 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 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 18. *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 (Sketch). Let $T = (Q_1, Q_2, \Sigma_1, \Sigma_2, q_0, \delta_1, \delta_2, F)$ and A the NFA obtained from T by ignoring output and priorities of T. Let Q_A be the states of A, which is of course just $(Q_1 \cup Q_2)$. The set of states of T_F is $(Q_1 \cup Q_2) \times 2^{Q_A}$. On the $(Q_1 \cup Q_2)$ part of the states of T_F, T_F behaves like T with priorities ignored, and the subsets of Q_A, which form the 2nd component of the states of T_F, is used to take priorities of δ_2 transitions in T into account. Each time we are at a state $q \in Q_2$ in T with $\delta_2(q) = (w_1, q_1) \cdots (w_k, q_k)$, T_F chooses non-deterministically a transition $q \to q_i$ (with output w_i) in the first component of states of T_F, and keep track with subsets from Q_A, that the input would not have been accepted if we took $q \to q_j$ for $j < i$. Note as we reach the next state on a path taken in T, we keep on tracking, in the 2nd component of the states of T_F, all states of T that could be reached, on the given input, from higher priority transitions we did not take at previous Q_2 states encountered. The accept states of T_F are the states (q, X) with $q \in F$ and $X \cap F = \emptyset$. Note T_F is a functional transducer, since the relation defined by T is a function. $\square$

Corollary 19. *(a) For prioritized transducers T_1 and T_2, equivalence can be decided in time $\mathcal{O}(\,(|T_1|2^{|T_1|_Q} + |T_2|2^{|T_2|_Q})^2\,)$. (b) 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. (a) For pTr T_1 and T_2, first check that $\text{dom}(T_1) = \text{dom}(T_2)$, which can obviously be done in the stated time complexity bound. Now convert T_1 and T_2 into functional transducers T_{F_1} and T_{F_2}, and use Theorem 1.1 in [8] 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 $\text{dom}(T_1) = \text{dom}(T_2)$), is quadratic in the number of transitions in $T_1 \cup T_2$. For (b) use (a) and the fact that for a regular expression R, $|\bar{J}^p(R)| \leq c|R|$, for some constant c. $\square$

Remark 20. 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 21. The transducer construction in the proof of Theorem 18 must be close to ideal, 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, $\text{dom}(T'_F)$ 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.

Some specifics of real-world matchers can be generalized away, such as the Σ_2 subsequences a real-world pTr outputs always forming a Dyck language (as in Sect. 3). 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 $\text{dom}(T)$, 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, notably it means that the amount of data output by a matching an input string w with T is bounded by $|\Sigma_2|\log(|w|)$.

Definition 22. *For a pTr T with $w \in dom(T)$, 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 w is a function $s^T : \Sigma_2 \to \{\perp, 0, \ldots, n\}$ such that for each $\beta \in \Sigma_2$ we have $\beta \in v_{f(\beta)}$, but $\beta \notin v_{f(\beta)+j}$ for any $j \in \mathbb{N}$. If $\beta \notin v_1 \cdots v_n$, $s^T(\beta) = \perp$.*

For a pTr T and a string $w = \alpha_1 \cdots \alpha_n \in \mathrm{dom}(T)$, we next describe a linear (in the length of the input string) algorithm to compute the slim parse output of T, where T is flattened. Let $f, f', f_\perp : \Sigma_2 \to \mathbb{N} \cup \{\perp\}$. For $v \in \Sigma_2^*$ and $k \in \mathbb{N}$, define $f' := U(f, v, k)$ by letting $f'(\beta) = k$ for $\beta \in v$, and $f'(\beta) = f(\beta)$ otherwise. Also, $f_\perp(\beta) = \perp$ for all $\beta \in \Sigma_2$. Define $\Delta(q, f, i) = (q_1, U(f, \beta_1, i)) \cdots (q_n, U(f, \beta_n, i))$ when $q \in Q_2$ and $\delta_2(q) = (\beta_1, q_1) \cdots (\beta_n, q_n)$, and $\Delta(q, f, i) = (q, f)$ for $q \in Q_1$. Now for the steps in the algorithm. (i) Let $S_0 = \Delta(q_0, f_\perp, 0)$. (ii) Given $S_i = (q_1, f_1) \cdots (q_m, f_m)$, where $i < n$, then $S_{i+1} = \sigma_1(\Delta(q_1', f_1, i) \cdots \Delta(q_m', f_m, i))$, where $q_j' = \delta_1(q_j, \alpha_{i+1})$ for each j. (iii) If $S_n = (q_1, f_1) \ldots (q_m, f_m)$ and i is the smallest index such that $q_i \in F$, then $s^T(w) = f_i$. If no state q_i is in F, the string w is rejected.

Theorem 23. *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 $w \in dom(T)$, it returns the slim parse for T on w, and if $w \notin dom(T)$, it rejects the input w.*

Proof (Sketch). Since highest priority paths in a flattened pTr may be determined by DFS, a pTr can be translated into a deterministic stack machine with output. Each S_i can be associated with the stack content at a particular stage of the DFS, with the left-most tuple being the top element of the stack. It follows from the argument in Remark 10 that duplicates of tuples with the same state can be removed from the stack. In contrast to a stack machine, the given algorithm simply processes all stack elements in parallel. Clearly, from the description of the slim parsing algorithm, it runs in linear time in the length of the input string. □

6 Conclusions and Future Work

In this paper we brought together several different angles on regular expressions into one formal framework. This enables us to talk both about the matching behaviors of less than ideal real-world matchers as in [2], while allowing a modelling of the special features of those matchers without being tied to their algorithmic choices. Still, there is ample room for continued work. For example, there are a *lot* 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 [1], which interact deeply with the matching procedure. From a theoretical perspective, the next step should be to determine the precise complexity class for equivalence in Corollary 19.

References

1. Berglund, M., Björklund, H., Drewes, F., van der Merwe, B., Watson, B.: Cuts in regular expressions. In: Béal, M.-P., Carton, O. (eds.) DLT 2013. LNCS, vol. 7907, pp. 70–81. Springer, Heidelberg (2013)
2. Berglund, M., Drewes, F., van der Merwe, B.: Analyzing catastrophic backtracking behavior in practical regular expression matching. In: Ésik, Z., Fülöp, Z., (eds.) Proceedings of the 14th International Conference on Automata and Formal Languages, pp. 109–123 (2014)
3. Câmpeanu, C., Salomaa, K., Yu, S.: A formal study of practical regular expressions. Int. J. Found. Comput. Sci. **14**(6), 1007–1018 (2003)
4. Cox, R.: Implementing regular expressions (2007). http://swtch.com/rsc/regexp/. (Accessed 3 March 2015)
5. Friedl, J.: Mastering Regular Expressions, 3rd edn. O'Reilly Media Inc., Sebastopol (2006)
6. Frisch, A., Cardelli, L.: Greedy regular expression matching. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 618–629. Springer, Heidelberg (2004)
7. Rathnayake, A., Thielecke, H.: Static analysis for regular expression exponential runtime via substructural logics. CoRR, abs/1405.7058 (2014)
8. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, New York (2009)
9. Sakuma, Y., Minamide, Y., Voronkov, A.: Translating regular expression matching into transducers. J. Applied Logic **10**(1), 32–51 (2012)
10. Thompson, K.: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968)
11. Wang, J.: Handbook of Finite State Based Models and Applications. 1st edn. Chapman & Hall/CRC, Boca Raton (2012)