POSIX Regular Expression Parsing with Derivatives

Martin Sulzmann[1] and Kenny Zhuo Ming Lu[2]

[1] Hochschule Karlsruhe - Technik und Wirtschaft
`martin.sulzmann@hs-karlsruhe.de`
[2] Nanyang Polytechnic
`luzhuomi@gmail.com`

Abstract. We adapt the POSIX policy to the setting of regular expression parsing. POSIX favors longest left-most parse trees. Compared to other policies such as greedy left-most, the POSIX policy is more intuitive but much harder to implement. Almost all POSIX implementations are buggy as observed by Kuklewicz. We show how to obtain a POSIX algorithm for the general parsing problem based on Brzozowski's regular expression derivatives. Correctness is fairly straightforward to establish and our benchmark results show that our approach is promising.

1 Introduction

We consider the parsing problem for regular expressions. Parsing produces a parse tree which provides a detailed explanation of which subexpressions match which substrings. The outcome of parsing is possibly ambiguous because there may be two distinct parse trees for the same input. For example, for input string ab and regular expression $(a + b + ab)^*$, there are two possible ways to break apart input ab: (1) a, b and (2) ab. Either in the first iteration subpattern a matches substring a, and in the second iteration subpattern b matches substring b, or subpattern ab immediately matches the input string.

There are two popular disambiguation strategies for regular expressions: POSIX [10] and greedy [21]. In the above, case (1) is the greedy result and case (2) is the POSIX result. For the variation $(ab + a + b)^*$, case (2) is still the POSIX result whereas now the greedy result equals case (2) as well.

We find that greedy parsing is directly tied to the structure and the order of alternatives matters. In contrast, POSIX is less sensitive to the order of alternatives because longest matches are favored. Only in case of equal matches preference is given to the left-most match. This is a useful property for applications where we build an expression as the composition of several alternatives, e.g. consider lexical analysis.

As it turns out, POSIX appears to be much harder to implement than greedy. Kuklewicz [11] observes that almost all POSIX implementations are buggy which is confirmed by our own experiments. These implementations are also restricted in that they do not produce full parse trees and only provide submatch information. For example, in case of Kleene star only the last match is recorded instead of the matches for each iteration.

In this work, we propose a novel method to compute POSIX parse trees based on Brzozowski's regular expression derivatives [1]. A sketch of how derivatives could be applied to compute POSIX submatches is given in our own prior work [24]. The present work includes some significant improvements such as a rigorous correctness result, dealing with the more general parsing problem and numerous optimizations.

Specifically, we make the following contributions:

- We formally define POSIX parsing by viewing regular expressions as types and parse trees as values (Section 2). We also relate parsing to the more specific submatching problem.
- We present a method for computation of POSIX parse trees based on Brzozowski's regular expression derivatives [1] and verify its correctness (Section 3).
- We have built optimized versions for parsing as well as for the special case of submatching where we only keep the last match in case of a Kleene star. We conduct experiments to measure the effectiveness of our method (Section 4).

Section 5 discusses related work and concludes.

2 Regular Expressions and Parse Trees

Words:

$$\begin{array}{rll} w ::= & \epsilon & \text{Empty word} \\ \mid & l \in \Sigma & \text{Letter} \\ \mid & ww & \text{Concatenation} \end{array}$$

Regular expressions:

$$\begin{array}{rll} r ::= & l & \\ \mid & r^* & \text{Kleene star} \\ \mid & rr & \text{Concatenation} \\ \mid & r + r & \text{Choice} \\ \mid & \epsilon & \text{Empty word} \\ \mid & \phi & \text{Empty language} \end{array}$$

Parse trees:

$$\begin{array}{rl} v ::= & () \mid l \mid (v, v) \mid \mathit{Left}\ v \mid \mathit{Right}\ v \mid vs \\ vs ::= & [] \mid v : vs \end{array}$$

$$\boxed{\vdash v : r}$$

$$(\text{None}^*) \quad \vdash [] : r^*$$

$$(\text{Once}^*) \quad \frac{\vdash v : r \quad \vdash vs : r^*}{\vdash (v : vs) : r^*}$$

$$(\text{Pair}) \quad \frac{\vdash v_1 : r_1 \quad \vdash v_2 : r_2}{\vdash (v_1, v_2) : r_1 r_2}$$

$$(\text{Left}+) \quad \frac{\vdash v_1 : r_1}{\vdash \mathit{Left}\ v_1 : r_1 + r_2}$$

$$(\text{Right}+) \quad \frac{\vdash v_2 : r_2}{\vdash \mathit{Right}\ v_2 : r_1 + r_2}$$

$$(\text{Empty}) \quad \vdash () : \epsilon \qquad (\text{Lit}) \quad \frac{l \in \Sigma}{\vdash l : l}$$

Flattening:

$$\begin{array}{llll} |()| = \epsilon & |l| \quad = l & |\mathit{Left}\ v| \ = |v| & |v : vs| = |v||vs| \\ |[]| \ = \epsilon & |(v_1, v_2)| = |v_1||v_2| & |\mathit{Right}\ v| = |v| & \end{array}$$

Fig. 1. Regular Expressions and Parse Trees

We follow [8] and phrase parsing as a type inhabitation relation. Regular expressions are interpreted as types and parse trees as values of some regular expression type. Figure 1 contains the details which will be explained in the following.

The syntax of regular expressions r is standard. As it is common, concatenation and alternation is assumed to be right associative. The example $(a+b+ab)^*$ from the introduction stands for $(a+(b+ab))^*$. Words w are formed using letters l taken from a finite alphabet Σ. Parse trees v are represented via some standard data constructors such as lists, pairs, left/right injection into a disjoint sum etc. We write $[v_1, ..., v_n]$ as a short-hand for $v_1 : ... : v_n : []$.

Parse trees v and regular expressions r are related via a natural deduction style proof system where inference rules make use of judgments $\vdash v : r$. For example, rule (Left+) covers the case that the left alternative r_1 has been matched. We will shortly see some examples making use of the other rules.

For each derivable statement $\vdash v : r$, the parse tree v provides a proof that the word underlying v is contained in the language described by r. That is, $L(r) = \{|v| \mid \vdash v : r\}$ where the flattening function $|\cdot|$ extracts the underlying word. In general, proofs are not unique because there may be two distinct parse trees for the same input.

Recall the example from the introduction. For expression $(a+(b+ab))^*$ and input ab we find parse trees $[\mathit{Left}\ a, \mathit{Right}\ \mathit{Left}\ b]$ and $[\mathit{Right}\ \mathit{Right}\ (a,b)]$. For brevity, some parentheses are omitted, e.g. we write $\mathit{Right}\ \mathit{Left}\ b$ as a short-hand for $\mathit{Right}\ (\mathit{Left}\ b)$. The derivation trees are shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash a : a \qquad \vdash b : b}{\vdash (a,b) : ab}}{\vdash \mathit{Right}\ (a,b) : b+ab}}{\vdash \mathit{Right}\ \mathit{Right}\ (a,b) : a+(b+ab)} \qquad \vdash [] : (a+(b+ab))^*}{\vdash [\mathit{Right}\ \mathit{Right}\ (a,b)] : (a+(b+ab))^*}$$

$$\dfrac{\dfrac{\vdash a : a}{\vdash \mathit{Left}\ a : (a+(b+ab))^*} \quad \dfrac{\dfrac{\dfrac{\vdash b : b}{\vdash \mathit{Left}\ b : b+ab}}{\vdash \mathit{Right}\ \mathit{Left}\ b : a+(b+ab)} \qquad \vdash [] : (a+(b+ab))^*}{\vdash [\mathit{Right}\ \mathit{Left}\ b] : (a+(b+ab))^*}}{\vdash [\mathit{Left}\ a, \mathit{Right}\ \mathit{Left}\ b] : (a+(b+ab))^*}$$

To avoid such ambiguities, the common approach is to impose a disambiguation strategy which guarantees that for each regular expression r matching a word w there exists a unique parse tree v such that $|v| = w$. Our interest is in the computation of POSIX parse trees. Below we give a formal specification of POSIX parsing by imposing an order among parse trees.

Definition 1 (POSIX Parse Tree Ordering). *We define a POSIX ordering $v_1 >_r v_2$ among parse trees v_1 and v_2 where r is the underlying regular* *expression. The ordering rules are as follows*

$$\text{(C1)}\ \frac{v_1 = v_1' \quad v_2 >_{r_2} v_2'}{(v_1, v_2) >_{r_1 r_2} (v_1', v_2')} \qquad \text{(C2)}\ \frac{v_1 >_{r_1} v_1'}{(v_1, v_2) >_{r_1 r_2} (v_1', v_2')}$$

$$\text{(A1)}\ \frac{\mathit{len}\ |v_2| > \mathit{len}\ |v_1|}{\mathit{Right}\ v_2 >_{r_1+r_2} \mathit{Left}\ v_1} \qquad \text{(A2)}\ \frac{\mathit{len}\ |v_1| \geq \mathit{len}\ |v_2|}{\mathit{Left}\ v_1 >_{r_1+r_2} \mathit{Right}\ v_2}$$

$$\text{(A3)}\ \frac{v_2 >_{r_2} v_2'}{\mathit{Right}\ v_2 >_{r_1+r_2} \mathit{Right}\ v_2'} \qquad \text{(A4)}\ \frac{v_1 >_{r_1} v_1'}{\mathit{Left}\ v_1 >_{r_1+r_2} \mathit{Left}\ v_1'}$$

$$\text{(K1)}\ \frac{|v : vs| = \epsilon}{[] >_{r^*} v : vs} \qquad \text{(K2)}\ \frac{|v : vs| \neq \epsilon}{v : vs >_{r^*} []}$$

$$\text{(K3)}\ \frac{v_1 >_r v_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2} \qquad \text{(K4)}\ \frac{v_1 = v_2 \quad vs_1 >_{r^*} vs_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2}$$

where helper function len computes the number of letters in a word.

Let r be a regular expression and v_1 and v_2 parse trees such that $\vdash v_1 : r$ and $\vdash v_2 : r$. We define $v_1 \geq_r v_2$ iff either v_1 and v_2 are equal or $v_1 >_r v_2$ where $|v_1| = |v_2|$. We say that v_1 is the POSIX *parse tree w.r.t. r iff $\vdash v_1 : r$ and $v_1 \geq_r v_2$ for any parse tree v_2 where $\vdash v_2 : r$ and $|v_1| = |v_2|$.*

The above ordering relation is an adaptation of the *Greedy* parse tree order defined in [8]. The (Greedy) rule $\mathit{Left}\ v >_{r_1+r_2} \mathit{Right}\ v'$ is replaced by rules (A1) and (A2). All other rules remain unchanged compared to [8].

Rules (A1) and (A2) guarantee that preference is given to *longest* left-most parse trees as stipulated by the POSIX submatching policy [10]:

> "Subpatterns should match the longest possible substrings, where subpatterns that start earlier (to the left) in the regular expression take priority over ones starting later. Hence, higher-level subpatterns take priority over their lower-level component subpatterns. Matching an empty string is considered longer than no match at all."

For example, consider again our running example. For expression $(a + (b + ab))^*$ and word ab we find parse trees $[\mathit{Right}\ \mathit{Right}\ (a, b)]$ and $[\mathit{Left}\ a, \mathit{Right}\ \mathit{Left}\ b]$. Due to rule (A1), we have that $\mathit{Right}\ \mathit{Right}\ (a, b)$ is greater than $\mathit{Left}\ a$ because $\mathit{Right}\ \mathit{Right}\ (a, b)$ contains a longer match than $\mathit{Left}\ a$. Hence,

$$[\mathit{Right}\ \mathit{Right}\ (a, b)] \geq_{(a+(b+ab))^*} [\mathit{Left}\ a, \mathit{Right}\ \mathit{Left}\ b]$$

In contrast, under the Greedy order we would find that $[\mathit{Left}\ a, \mathit{Right}\ \mathit{Left}\ b]$ is greater than $[\mathit{Right}\ \mathit{Right}\ (a, b)]$.

POSIX is Non-Problematic In case of the Greedy parse tree order, it is well-observed [8] that special care must be given to problematic expressions/parse trees. Roughly, an expression induces problematic parse trees if we find empty matches under a Kleene star. The potential danger of problematic expressions is that we might end up with an infinite chain of larger parse trees. This causes possible complications for a Greedy parsing algorithm, as the algorithm attempts to compute the "largest" parse tree. Fortunately, none of this is an issue for POSIX.

For example, consider the problematic expression ϵ^*. For the empty input we find the following infinite chain of parse trees

$$v_0 = [],\ v_1 = [()],\ v_2 = [(),()] \ ...$$

Parse tree v_0 is the largest according to our ordering relation. See rule (K1).

Let's consider another more devious, problematic expression $(\epsilon + a)^*$ where for input a we find

$$v_0 = [\mathit{Right}\ a],\ v_1 = [\mathit{Left}\ (), \mathit{Right}\ a],\ v_2 = [\mathit{Left}\ (), \mathit{Left}\ (), \mathit{Right}\ a] \ ...$$

Due to rule (A1), v_0 is the largest parse tree according to our POSIX ordering relation. In contrast, under the Greedy order each v_{i+1} is larger than v_i. Hence, the Greedy order does not enjoy maximal elements unless we take special care of problematic expressions. For details see [8].

To summarize, expressions which are problematic under the Greedy order are "not problematic" under the POSIX order. For *any* expression, the POSIX order defined in Definition 1 is well-defined in the sense that the order is total and enjoys maximal elements.

Proposition 1 (Maximum and Totality of POSIX Order). *For any expression r, the ordering relation $\geq_r$ is total and has a maximal element.*

Annotated regular expressions: $r ::= (x : r) \mid l \mid r^* \mid rr \mid r + r \mid \epsilon \mid \phi$
Submatch binding environment: $\Gamma ::= \{\} \mid \{x \mapsto w\} \mid \Gamma \cup \Gamma$

$$\boxed{v \vdash r \leadsto \Gamma}$$

$$\frac{v \vdash r \leadsto \Gamma}{v \vdash (x : r) \leadsto \{x \mapsto |v|\} \cup \Gamma} \quad () \vdash \epsilon \leadsto \{\} \quad l \vdash l \leadsto \{\}$$

$$[] \vdash r^* \leadsto \{\} \quad \frac{v \vdash r \leadsto \Gamma}{[v] \vdash r^* \leadsto \Gamma} \quad \frac{v \vdash r \leadsto \Gamma_1 \ \ vs \vdash r^* \leadsto \Gamma_2}{v : vs \vdash r^* \leadsto \Gamma_1 \cup \Gamma_2}$$

$$\frac{\begin{array}{c} v_1 \vdash r_1 \leadsto \Gamma_1 \\ v_2 \vdash r_2 \leadsto \Gamma_2 \end{array}}{(v_1, v_2) \vdash r_1 r_2 \leadsto \Gamma_1 \cup \Gamma_2} \quad \frac{v_1 \vdash r_1 \leadsto \Gamma_1}{\mathit{Left}\ v_1 \vdash r_1 + r_2 \leadsto \Gamma_1} \quad \frac{v_2 \vdash r_2 \leadsto \Gamma_2}{\mathit{Right}\ v_2 \vdash r_1 + r_2 \leadsto \Gamma_2}$$

Fig. 2. From Parsing to Submatching

Parsing versus Submatching For space reasons, practical implementations only care about certain subparts and generally only record the last match in case of a Kleene star iteration. For example, consider expression $((x : a^*) + (b+c)^*)^*$ where via an annotation we have marked the subparts we are interested in. Matching the above against word *abaacc* yields the submatch binding $x \mapsto aa$. For comparison, here is the parse tree resulting from the match against the input word *abaacc*

$$[\mathit{Left}\ [a], \mathit{Right}\ \mathit{Left}\ [b], \mathit{Left}\ [a,a], \mathit{Right}\ \mathit{Right}\ [c,c]]$$

Instead of providing a stand-alone definition of POSIX submatching, we show how to derive submatchings from parse trees. In Figure 2, we extend the syntax of regular expressions with submatch annotations $(x : r)$ where variables x are always distinct. For parsing purposes, submatch annotations will be ignored. Given a parse tree v of a regular expression r, we obtain the submatch environment Γ via judgments $v \vdash r \rightsquigarrow \Gamma$. We simply traverse the structure of v and r and collect the submatch binding Γ.

For our above example, we obtain the binding $\{x \mapsto a, x \mapsto aa\}$. Repeated bindings resulting from Kleene star are removed by only keeping the last submatch. Technically, we achieve this by exhaustive application of the following rule on submatch bindings (from left to right):

$$\Gamma_1 \cup \{x \mapsto w_1\} \cup \Gamma_2 \cup \{x \mapsto w_2\} \cup \Gamma_3 = \Gamma_1 \cup \Gamma_2 \cup \{x \mapsto w_2\} \cup \Gamma_3$$

Hence, we find the final submatch binding $\{x \mapsto aa\}$. As another example, consider expression $(x : a^*)^*$ and the empty input string. The POSIX parse tree for $(x : a^*)^*$ is $[]$ which implies the POSIX submatching $\{x \mapsto \epsilon\}$.

We believe that the submatchings resulting from POSIX parse trees correspond to the POSIX submatchings described in [25]. The formal details need yet to be worked out.

Construction of a full parse tree is of course wasteful, if we are only interested in certain submatchings. However, both constructions are equally challenging in case we wish to obtain the proper POSIX candidate. That is, even if we only keep the last match in case of a Kleene star iteration, we must compare the set of accumulated submatches to select the appropriate POSIX, i.e. longest left-most, match.

A naive method to obtain the POSIX parse tree is to perform an exhaustive search. Such a method is obviously correct but potentially has an exponential run time due to backtracking. Next, we develop a systematic method to compute the POSIX parse trees.

3 Parse Tree Construction via Derivatives

Our idea is to apply Brzozowski's regular expression derivatives [1] for parsing. The derivative operation $r \backslash l$ performs a symbolic transformation of regular expression r and extracts (takes away) the leading letter l. In formal language terms, we find

$$lw \in L(r) \text{ iff } w \in L(r \backslash l)$$

Thus, it is straightforward to obtain a regular expression matcher. To check if regular expression r matches word $l_1 ... l_n$, we simply build a sequence of derivatives and test if the final regular expression is nullable, i.e. accepts the empty string:

Regular expression derivatives:

$$
\begin{array}{ll}
\phi\backslash l & = \phi \\
\epsilon\backslash l & = \phi \\
l_1\backslash l_2 & = \begin{cases} \epsilon \text{ if } l_1 == l_2 \\ \phi \text{ otherwise} \end{cases} \\
(r_1 + r_2)\backslash l & = r_1\backslash l + r_2\backslash l \\
(r_1 r_2)\backslash l & = \begin{cases} (r_1\backslash l)r_2 + r_2\backslash l & \text{if } \epsilon \in L(r_1) \\ (r_1\backslash l)r_2 & \text{otherwise} \end{cases} \\
r^*\backslash l & = (r\backslash l)r^*
\end{array}
$$

Empty parse tree construction and parse tree transformation:

$$
\begin{array}{l}
\mathit{mkEps}_{r^*} = [] \\
\mathit{mkEps}_{r_1 r_2} = (\mathit{mkEps}_{r_1}, \mathit{mkEps}_{r_2}) \\
\mathit{mkEps}_{r_1 + r_2} \\
\quad |\epsilon \in L(r_1) = \mathit{Left}\ \mathit{mkEps}_{r_1} \\
\quad |\epsilon \in L(r_2) = \mathit{Right}\ \mathit{mkEps}_{r_2} \\
\mathit{mkEps}_{\epsilon} = ()
\end{array}
$$

$$
\begin{array}{l}
\mathit{inj}_{r^*\backslash l}(v, vs) = (\mathit{inj}_{r\backslash l}\ v) : vs \\
\mathit{inj}_{(r_1 r_2)\backslash l} = \\
\quad \lambda v.\mathsf{case}\ v\ \mathsf{of} \\
\qquad (v_1, v_2) \rightarrow (\mathit{inj}_{r_1\backslash l}\ v_1, v_2) \\
\qquad \mathit{Left}\ (v_1, v_2) \rightarrow (\mathit{inj}_{r_1\backslash l}\ v_1, v_2) \\
\qquad \mathit{Right}\ v_2 \rightarrow (\mathit{mkEps}_{r_1}, \mathit{inj}_{r_2\backslash l}\ v_2) \\
\mathit{inj}_{(r_1 + r_2)\backslash l} = \\
\quad \lambda v.\mathsf{case}\ v\ \mathsf{of} \\
\qquad \mathit{Left}\ v_1 \rightarrow \mathit{Left}\ (\mathit{inj}_{r_1\backslash l}\ v_1) \\
\qquad \mathit{Right}\ v_2 \rightarrow \mathit{Right}\ (\mathit{inj}_{r_2\backslash l}\ v_2) \\
\mathit{inj}_{l\backslash l}() = l
\end{array}
$$

Parsing with derivatives:

$$
\begin{array}{l}
\mathit{parse}\ r\ \epsilon \\
\quad |\epsilon \in L(r) = \mathit{mkEps}_r \\
\mathit{parse}\ r\ lw = \mathit{inj}_{r\backslash l}(\mathit{parse}\ r\backslash l\ w)
\end{array}
$$

Fig. 3. Parsing Tree Construction with Derivatives

$$\text{Matching by extraction:} \qquad r_0 \xrightarrow{l_1} r_1 \xrightarrow{l_2} ... \xrightarrow{l_n} r_n$$

In the above, we write $r \xrightarrow{l} r'$ for applying the derivative operation on r where r' equals $r\backslash l$. In essence, derivatives represent DFA states and $\xrightarrow{l}$ represents the DFA transition relation.

Our insight is that based on the first *matching* pass we can build the POSIX parse tree via a second *injection* pass:

$$\text{Parse trees by injection} \qquad v_0 \xleftarrow{l_1} v_1 \xleftarrow{l_2} ... \xleftarrow{l_n} v_n$$

The basic idea is as follows. After the final matching step, we compute the parse tree v_n for a nullable expression r_n. Then, we apply a sequence of parse tree transformations. In each transformation step, we build the parse tree v_i for expression r_i given the tree v_{i+1} for r_{i+1} where $r_i \xrightarrow{l} r_{i+1}$ This step is denoted by $v_i \xleftarrow{l} v_{i+1}$. In essence, the derivative operation removes leading letters from an expression whereas the transformation via injection step simply reverses this effect at the level of parse trees. In the above, we *inject* the *removed* letter l into the parse tree v_{i+1} of the derived expression r_{i+1} which results in a parse tree v_i of expression r_i. Thus, we incrementally build the parse tree v_0 for the initial expression r_0. Importantly, our method yields POSIX parse tree because (a) we build the POSIX parse tree v_n for the nullable expression r_n and (b) each parse tree transformation step $v_i \overset{l}{\leftarrow} v_{i+1}$ maintains the POSIX property.

Next, we introduce the details of the above sketched POSIX parsing method followed by a worked out example. Finally, we present an improvement parsing algorithm which performs the 'backward' construction of POSIX parse trees during the 'forward' matching pass.

POSIX Parse Tree Construction via Injection Figure 3 summarizes our method for construction of POSIX parse trees based on the above idea. We first repeat the standard derivative operation $r\backslash l$. Next, we find function $mkEps_r$ to compute an empty parse tree assuming that r is nullable. The function is defined by structural induction over r where as a notational convention the cases are written as subscripts. Similarly, we find function $inj_{r\backslash l}$ which takes as an input a parse tree of the derivative $r\backslash l$ and yields a parse of r by (re)injecting the removed letter l. Thus, we can define the transformation step $v_i \overset{l}{\leftarrow} v_{i+1}$ by $v_i = inj_{r_i\backslash l}\ v_{i+1}$. Function *parse* computes a parse tree by first applying the derivative operation until we obtain a parse tree for the empty tree via *mkEps*. Starting with this parse tree, we then repeatedly apply *inj*.

Let us take a closer look at $mkEps_r$. We recurse over the structure of r. There is no case for letter l and empty language ϕ as we assume that r must be nullable. The cases for Kleene star r^* and empty word ϵ are straightforward and yield [], respectively, (). For concatenation $r_1 + r_2$, we build the pair consisting of the empty parse trees for r_1 and r_2. The most interesting case is choice $r_1 + r_2$ where we are careful to first check if r_1 is nullable. Otherwise, we consider r_2. Thus, we can guarantee that the resulting parse tree is the largest according to our POSIX order in Definition 1.

Lemma 1 (Empty POSIX Parse Tree). *Let r be a regular expression such that $\epsilon \in L(r)$. Then, $\vdash mkEps_r : r$ and $mkEps_r$ is the POSIX parse tree of r for the empty word.*

Next, we take a closer look at the definition *inj*. For example, the most simple (last) case is $inj_{l\backslash l}() = l$ where we transform the empty parse tree () into l. Recall that $l\backslash l$ equals ϵ. The definition for choice is also simple. We check if either a parse for the left or right component exists. Then, we apply *inj* on the respective component.

Let's consider the first case dealing with Kleene star. By definition $r^*\backslash l = (r\backslash l)r^*$. Hence, the input consists of a pair (v, vs). Function $inj_{r\backslash l}$ is applied recursively on v to yield a parse tree for r.

Concatenation r_1r_2 is the most involved case. There are three possible subcases. The first subcase covers the case that r_1 is not nullable. The other two cases deal with the nullable case.

In case r_1 is not nullable, we must find a pair (v_1, v_2). Recall that for this case $(r_1r_2)\backslash l = (r_1\backslash l)r_2$. Hence, the derivative operation has been applied on r_1 which implies that *inj* will also be applied on v_1.

Let's consider the two subcases dealing with nullable expressions r_1. Recall that in such a situation we have that $(r_1r_2)\backslash l = (r_1\backslash l)r_2 + r_2\backslash l$. Hence, we need to check if either a parse tree for the left or right expression exists. In case of a left parse tree, we apply *inj* on the leading component (like for non-nullable r_1). In case of a right parse tree, none of the letters have been extracted from r_1. Hence, we build a pair consisting of an 'empty' parse tree $mkEps_{r_1}$ for r_1 and r_2's parse tree by injecting l back into v_2 via $inj_{r_2\backslash l}$.

It is not difficult to see that $inj_{r\backslash l}$ applied on a parse tree of $r\backslash l$ yields a parse tree of r. The important property for us is that injection also maintains POSIX parse trees.

Lemma 2 (POSIX Preservation under Injection). *Let r be a regular expression, l a letter, v a parse tree such that $\vdash v : r\backslash l$ and v is POSIX parse tree of $r\backslash l$ and $|v|$. Then, $\vdash (inj_{r\backslash l}\ v) : r$ and $(inj_{r\backslash l}\ v)$ is POSIX parse tree of r and $l|v|$ where $|(inj_{r\backslash l}\ v)| = l|v|$.*

We have a rigorous proof for this statement. The proof is rather involved and requires a careful analysis of the various (sub)cases. Briefly, *inj* strictly injects letters at the *left-most* position. Recall that derived expressions are obtained by greedily removing leading letters from the left. Injection also preserves the *longest* left-most property because the derivative operation favors subexpressions that start earlier. Recall the case for choice

$$(r_1 r_2)\backslash l = \begin{cases} (r_1\backslash l)r_2 + r_2\backslash l & \text{if } \epsilon \in L(r_1) \\ (r_1\backslash l)r_2 & \text{otherwise} \end{cases}$$

where we favor subexpression r_1. Thus, injection can guarantee that longest left-most parse trees are preserved.

Based on the above lemmas we reach the following result.

Theorem 1 (POSIX Parsing). *Function parse computes POSIX parse trees.*

POSIX Parsing Example To illustrate our method, we consider expression $(a + ab)(b + \epsilon)$ and word ab for which we find parse trees $(Right\ (a, b), Right\ ())$ and $(Left\ a, Left\ b)$. The former is the POSIX parse tree whereas the latter is the Greedy parse tree.

We first build the derivative w.r.t. a and then w.r.t. b. For convenience, we use notation $\xrightarrow{l}$ to denote derivative steps. For our example, we find:

$$\begin{array}{ll} & (a + ab)(b + \epsilon) \\ \xrightarrow{a} & (\epsilon + \epsilon b)(b + \epsilon) \\ \xrightarrow{b} & (\phi + (\phi b + \epsilon))(b + \epsilon) + (\epsilon + \phi) \end{array}$$

where the last step $\xrightarrow{b}$ in more detail is as follows:

$$\begin{array}{ll} & ((\epsilon + \epsilon b)(b + \epsilon))\backslash b \\ = & ((\epsilon + \epsilon b)\backslash b)(b + \epsilon) + (b + \epsilon)\backslash b \\ = & (\epsilon\backslash b + (\epsilon b)\backslash b)(b + \epsilon) + (b\backslash b + \epsilon\backslash b) \\ = & (\phi + ((\epsilon\backslash b)b + b\backslash b))(b + \epsilon) + (\epsilon + \phi) \\ = & (\phi + (\phi b + \epsilon))(b + \epsilon) + (\epsilon + \phi) \end{array}$$

Next, we check that the final expression $(\phi + (\phi b + \epsilon))(b + \epsilon) + (\epsilon + \phi)$ is nullable which is the case here. Hence, we can compute the empty POSIX parse tree via

$$mkEps_{(\phi+(\phi b+\epsilon))(b+\epsilon)+(\epsilon+\phi)} = \mathit{Left}\ (\mathit{Right}\ (\mathit{Right}\ ()), \mathit{Right}\ ())$$

What remains is to apply the 'backward' injection pass where the POSIX parse tree v' of $r\backslash l$ is transformed into a POSIX parse tree v of r by injecting the letter l appropriately into v'.

We find

$$\begin{array}{l} \quad inj_{((\epsilon+\epsilon b)(b+\epsilon))\backslash b}\ (\mathit{Left}\ (\mathit{Right}\ (\mathit{Right}\ ()), \mathit{Right}\ ())) \\ = (inj_{(\epsilon+\epsilon b)\backslash b} \mathit{Right}\ (\mathit{Right}\ ()), \mathit{Right}\ ()) \\ = (\mathit{Right}\ (inj_{(\epsilon b)\backslash b}\ (\mathit{Right}\ ())), \mathit{Right}\ ()) \\ = (\mathit{Right}\ (mkEps_{\epsilon}, inj_{b\backslash b}()), \mathit{Right}\ ()) \\ = (\mathit{Right}\ ((), b), \mathit{Right}\ ()) \end{array}$$

where $(\mathit{Right}\ ((), b), \mathit{Right}\ ())$ is the POSIX parse tree of $(\epsilon + \epsilon b)(b + \epsilon)$ and word b.

Another injection step yields

$$inj_{((a+ab)(b+\epsilon))\backslash a}\ (\mathit{Right}\ ((), b), \mathit{Right}\ ()) \ = \ (\mathit{Right}\ (a, b), \mathit{Right}\ ())$$

As we know the above is the POSIX parse tree for expression $(a+ab)(b+\epsilon)$ and word ab.

Incremental Bit-Coded Forward Parse Tree Construction Next, we show how to perform parsing more efficiently by incrementally building up parse trees during matching. That is, the 'second' injection step is immediately applied during matching. Thus, we avoid to record the entire path of derived expressions. In addition, we use bit-codes to represent parse trees more compactly.

Our bit-code representation of parse trees follows the description in [17]. See Figure 4. Bit-code sequences are represented as lists where we use Haskell notation. The symbol [] denotes the empty list and $b : bs$ denotes a list with head b and tail bs. We write ++ to concatenate two lists. Function $encode_r$ computes a bit-code representation of parse tree v where $\vdash\ v : r$. Function $decode_r$ turns a bit-code representation back into a parse tree.

The main challenge is the incremental construction of parse trees during matching. The idea is to incrementally annotate regular expressions with partial parse tree information during the derivative step.

Annotated regular expressions ri are defined in Figure 5. Each annotation bs represents some partial parse tree information in terms of bit-code sequences. There is no annotation for ϕ as there is no parse tree for the empty language.

Function *internalize* transforms a standard regular expressions r into an annotated regular expressions ri by inserting empty annotations []. In addition, choice $+$ is transformed into $\oplus$. The purpose of this transformation is that all parse tree information can be derived from the operands of $\oplus$ without having to inspect the surrounding structure. For example, we attach 0 ("left position") to the internalized expression resulting from left alternative r_1 of $r_1 + r_2$ where helper function *fuse* attaches a bit-code sequence to the top-most position of an annotated regular expression.

As an example, consider application of function *internalize* to the expression $(a + ab)(b + \epsilon)$ which yields the annotated expression

$$[]@(([0]@a) \oplus ([1]@([]@a)([]@b)))(([0]@b) \oplus ([1]@\epsilon)) \qquad (1)$$

Bit-codes $b ::= 0 \mid 1 \quad bs ::= [] \mid b : bs$

$$
\begin{array}{l}
\mathit{encode}_\epsilon\ () = [] \\
\mathit{encode}_l\ l = [] \\
\mathit{encode}_{r_1+r_2}\ (\mathit{Left}\ v) = 0 : \mathit{encode}_{r_1}\ v \\
\mathit{encode}_{r_1+r_2}\ (\mathit{Right}\ v) = 1 : \mathit{encode}_{r_2}\ v \\
\mathit{encode}_{r_1 r_2}\ (v_1, v_2) = \mathit{encode}_{r_1}\ v_1 \mathbin{+\!\!+} \mathit{encode}_{r_2}\ v_2 \\
\mathit{encode}_{r^*}\ [] = [1] \\
\mathit{encode}_{r^*}\ (v : vs) = (0 : \mathit{encode}_r\ v) \mathbin{+\!\!+} \mathit{encode}_{r^*}\ vs
\end{array}
$$

```
$decode_r\, bs$ = let $(v, p) = decode'_r\, bs$
             in case $p$ of
                  $[] \rightarrow v$
$decode'_\epsilon\ bs = ((), bs)$
$decode'_l\ bs = (l, bs)$
$decode'_{r_1+r_2}\ (0 : bs)$ = let $(v, p) = decode'_{r_1}\ bs$
                         in $(Left\ v, p)$
$decode'_{r_1+r_2}\ (1 : bs)$ = let $(v, p) = decode'_{r_2}\ bs$
                         in $(Right\ v, p)$
$decode'_{r_1 r_2}\ bs$ = let $(v_1, p_1) = decode'_{r_1}\ bs$
                   $(v_2, p_2) = decode'_{r_2}\ p_1$
               in $((v_1, v_2), p_2)$
$decode'_{r^*}\ (0 : bs)$ = let $(v, p_1) = decode'_r\ bs$
                       $(vs, p_2) = decode'_{r^*}\ p_1$
                   in $((v : vs), p_2)$
$decode'_{r^*}\ (1 : bs) = ([], bs)$
```

Fig. 4. Bit-Code Representation of Parse Trees

The derivative operation now operates on annotated regular expressions. See Figure 5. To avoid confusion, we denote the refined derivative operation by $ri\backslash_b l$. As can be seen, the definition of $ri\backslash_b l$ follows closely the definition of the standard derivative operation $r\backslash l$. The difference is that $ri\backslash_b l$ propagates and inserts parse tree information in terms of annotations. In essence, $ri\backslash_b l$ is an amalgamation of $r\backslash l$ and $inj_{r\backslash l}$.

For example, consider $ri_1\ ri_2$ where $\epsilon \in L(ri_1)$. Like in the standard case, the letter l could either be extracted from ri_1 or ri_2. We keep track of both alternatives by combining them via $\oplus$. The interesting case is if l is extracted from ri_2 which implies that the parse tree of ri_1 must be empty. We record this information via $\mathit{fuse}\ \mathit{mkEpsBC}_{ri_1}\ (ri_2\backslash_b l)$. Helper function $\mathit{mkEpsBC}_{ri_1}$ computes an empty parse tree of ri_1 in terms of the bit-code representation. This information is then attached to the top-most annotation in $ri_2\backslash_b l$ via the helper function fuse.

Similarly, the annotations resulting from Kleene star must record the number of iterations we have performed. For example, $\mathit{fuse}\ [0]\ ri\backslash_b l$ records that the Kleene star has been unrolled once. The existing annotation bs is moved to the resulting concatenation whereas we attach $[]$ to ri^* to indicate the start of a new Kleene star iteration.

Bit-code annotated regular expressions:

$$ri \ ::= \ \phi \mid (bs@\epsilon) \mid (bs@l) \mid (bs@ri \oplus ri) \mid (bs@ri\ ri) \mid (bs@ri^*)$$

$$\begin{array}{l}
\textit{internalize}\ \phi = \phi \\
\textit{internalize}\ \epsilon = ([]@\epsilon) \\
\textit{internalize}\ l = ([]@l) \\
\textit{internalize}\ (r_1 + r_2) = ([]@(\textit{fuse}\ [0]\ (\textit{internalize}\ r_1)) \oplus (\textit{fuse}\ [1]\ (\textit{internalize}\ r_2))) \\
\textit{internalize}\ (r_1\ r_2) = ([]@(\textit{internalize}\ r_1)\ (\textit{internalize}\ r_2)) \\
\textit{internalize}\ r^* = ([]@(\textit{internalize}\ r)^*)
\end{array}$$

$$\begin{array}{l}
\textit{fuse}\ bs\ \phi = \phi \\
\textit{fuse}\ bs\ (p@\epsilon) = (bs{+}{+}p@\epsilon) \\
\textit{fuse}\ bs\ (p@l) = (bs{+}{+}p@l) \\
\textit{fuse}\ bs\ (p@ri_1 \oplus ri_2) = (bs{+}{+}p@ri_1 \oplus ri_2) \\
\textit{fuse}\ bs\ (p@ri_1\ ri_2) = (bs{+}{+}p@ri_1\ ri_2) \\
\textit{fuse}\ bs\ (p@ri^*) = (bs{+}{+}p@ri^*)
\end{array}$$

Incremental POSIX parsing:

$$\begin{array}{ll}
\phi \backslash_b l & = \phi \\
(bs@\epsilon)\backslash_b l & = \phi \\
(bs@l_1)\backslash_b l_2 & = \begin{cases} (bs@\epsilon) & \text{if } l_1 == l_2 \\ \phi & \text{otherwise} \end{cases} \\
(bs@ri_1 \oplus ri_2)\backslash_b l & = (bs@ri_1\backslash_b l \oplus ri_2\backslash_b l) \\
(bs@ri_1\ ri_2)\backslash_b l & = \begin{cases} (bs@(ri_1\backslash_b l)\ ri_2) \oplus\ (\textit{fuse}\ \textit{mkEpsBC}_{ri_1}\ (ri_2\backslash_b l)) & \text{if } \epsilon \in L(ri_1) \\ (bs@(ri_1\backslash_b l))\ ri_2 & \text{otherwise} \end{cases} \\
(bs@ri^*)\backslash_b l & = (bs@(\textit{fuse}\ [0]\ ri\backslash_b l)\ ([]@ri^*))
\end{array}$$

$$\begin{array}{l}
\textit{mkEpsBC}_{(bs@\epsilon)} = bs \\
\textit{mkEpsBC}_{(bs@ri_1 \oplus ri_2)} \\
\quad \mid \epsilon \in L(ri_1) = bs{+}{+}\textit{mkEpsBC}_{ri_1} \\
\quad \mid \epsilon \in L(ri_2) = bs{+}{+}\textit{mkEpsBC}_{ri_2} \\
\textit{mkEpsBC}_{(bs@ri_1\ ri_2)} = bs{+}{+}\textit{mkEpsBC}_{ri_1}{+}{+}\textit{mkEpsBC}_{ri_2} \\
\textit{mkEpsBC}_{(bs@ri^*)} = bs{+}{+}[1]
\end{array}$$

$$\begin{array}{l}
\textit{parseBC}'ri\ \epsilon \\
\quad \mid \epsilon \in L(r) = \textit{mkEpsBC}_{ri} \\
\textit{parseBC}'ri\ lw = \textit{parseBC}'\ ri\backslash_b l\ w \\
\textit{parseBC}rw = \textit{decode}_r(\textit{parseBC}'(\textit{internalize}\ r)w)
\end{array}$$

Fig. 5. Incremental Bit-Coded Forward POSIX Parse Tree Construction

For example, $\backslash_b a$ applied on the above annotated expression (1) yields

$$[]@(([0]@\epsilon) \oplus ([1]@([]@\epsilon)([]@b)))(([0]@b) \oplus ([1]@\epsilon))$$

Let us take a closer look at $\textit{mkEpsBC}_{ri}$ which follows the definition of *mkEps* in Figure 3. Like in case of *mkEps*, we first check the left and then the right alternative in $ri_1 \oplus ri_2$. One difference is that operands themselves record the information which alternative (left or right) they originated. Recall the definition of *internalize*. Hence, it suffices to collect the annotations in either ri_1 or ri_2.

Thus, incremental parsing via function *parseBC* is performed by (1) internalizing the regular expression, (2) repeated application of the refined derivative operation, (3) extraction of the accumulated annotations of the final 'empty' regular expression, and (4) turning the bit-code representation into parse tree from.

Simplifications A well-known issue is that the size and number of derivatives may explode. For example, consider the following derivative steps.

$$a^* \xrightarrow{a} \epsilon a^* \xrightarrow{a} \phi a^* + \epsilon a^* \xrightarrow{a} (\phi a^* + \epsilon a^*) + (\phi a^* + \epsilon a^*) \xrightarrow{a} ...$$

As can easily be seen, subsequent derivatives are all equivalent to ϵa^*.

$isPhi\ (bs@ri^*) = False$
$isPhi\ (bs@ri_1\ ri_2) = isPhi\ ri_1 \lor isPhi\ ri_2$
$isPhi\ (bs@ri_1 \oplus ri_2) = isPhi\ ri_1 \land isPhi\ ri_2$
$isPhi\ (bs@l) = False$
$isPhi\ (bs@\epsilon) = False$
$isPhi\ \phi = True$

We assume that $\oplus$ takes a list of operands, written $(bs@\oplus [ri_1, ..., ri_n])$.

```
simp $(bs@(bs'@\epsilon)\ ri)$
    $|isPhi\ r = \phi$
    $|\text{otherwise} = fuse\ (bs{+}{+}bs')ri$
simp $(bs@ri_1\ ri_2)$
    $|isPhi\ ri_1 \lor isPhi\ ri_2 = \phi$
    $|\text{otherwise} = bs@(simp\ ri_1)\ (simp\ ri_2)$
$simp\ (bs@\oplus\ []) = \phi$
$simp\ (bs@\oplus\ ((bs'@\oplus rsi_1) : rsi_2)) = bs@\oplus\ ((map\ (fuse\ bs')\ rsi_1){+}{+}rsi_2)$
$simp\ (bs@\oplus [ri]) = fuse\ bs(simp\ ri)$
$simp\ (bs@\oplus (ri : rsi)) = bs@\oplus\ (nub\ (filter\ (not.isPhi\ )\ ((simp\ ri) : map\ simp\ rsi)))$
```

Fig. 6. Simplifications

To ensure that that the size and number of derivatives remains finite, we simplify regular expressions. Each simplification step must maintain the parse tree information represented by the involved regular expression. Figure 6 performs simplification of annotated expressions in terms of function *simp*. We assume that *simp* is applied repeatedly until a fixpoint is reached.

For convenience, we assume that $\oplus$ takes a list of operands, instead of just two, and therefore write $(bs@\oplus [ri_1, ..., ri_n])$. This notational convention makes it easier to put alternatives into right-associative normal form and apply simplification steps to remove duplicates and expressions equivalent to the empty language. Helper *isPhi* indicates if an expression equals the empty language. We can safely remove such cases via *filter*. In case of duplicates in a list of alternatives, we only keep the first occurrence via *nub*.

Linear-Time Complexity Claim It is easy to see that each call of one of the functions/operations $\backslash_b$, *simp*, *fuse*, *mkEpsBC* and *isPhi* leads to subcalls whose number is bound by the size of the regular expression involved. We claim that thanks to aggressively applying *simp* this size remains finite. Hence, we can argue that the above mentioned functions/operations have constant time complexity which implies that we can incrementally compute bit-coded parse trees in linear time in the size of the input. We yet need to work out detailed estimates regarding the space complexity of our algorithm.

Correctness Claim We further claim that the incremental parsing method in Figure 5 in combination with the simplification steps in Figure 6 yields POSIX parse trees. We have tested this claim extensively by using the method in Figure 3 as a reference but yet have to work out all proof details.

For example, we claim that $r\backslash_b l$ is related to $inj_{r\backslash l}$ as follows. Let r be a regular expression, l be a letter and v' a parse tree such that $\vdash v' : r\backslash l$. Then, we claim that $inj_{r\backslash l}\ v' = decode_r\ (retrieve_{(internalize\ r)\backslash_b l}\ v')$ where

$$\begin{array}{l}
retrieve_{(bs\ @\ \epsilon)}\ \ () \ =\ bs \\
retrieve_{(bs\ @\ l)}\ \ l\ =\ bs \\
retrieve_{(bs\ @\ ri_1\ \oplus\ ri_2)}\ \ (Left\ \ v)\ =\ bs +\!\!+\ retrieve_{ri_1}\ \ v \\
retrieve_{(bs\ @\ ri_1\ \oplus\ ri_2)}\ \ (Right\ \ v)\ =\ bs +\!\!+\ retrieve_{ri_2}\ \ v \\
retrieve_{(bs\ @\ ri_1\quad ri_2)}\ \ (v_1, v_2)\ =\ bs +\!\!+\ retrieve_{ri_1}\ \ v_1 +\!\!+\ retrieve_{ri_2}\ \ v_2 \\
retrieve_{(bs\ @\ ri^*)}\ \ []\ =\ bs +\!\!+ [1] \\
retrieve_{(bs\ @\ ri^*)}\ \ (v : vs)\ =\ bs +\!\!+ [0] +\!\!+\ retrieve_{ri}\ \ v +\!\!+\ retrieve_{([]\ @\ ri^*)}\ \ vs
\end{array}$$

Function *retrieve* assembles a complete parse tree in bit-code representation based on the annotations in $(internalize\ \ r)\backslash_b l$ with respect to a given parse tree v'.

A similar claim applies to *simp*. We plan to work out the formal proof details in future work.

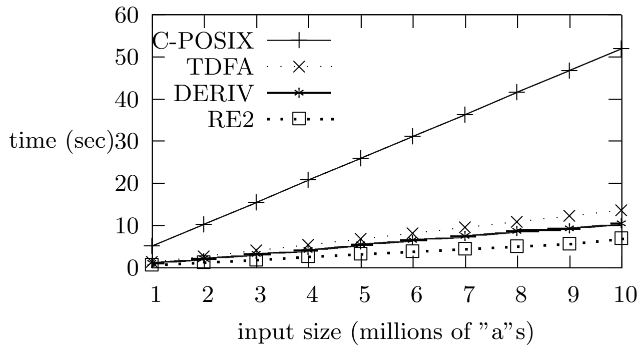
4 Experiments

We have implemented the incremental bit-coded POSIX parsing approach in Haskell. An explicit DFA is built where each transition has its associated parse tree transformer attached. Thus, we avoid repeated computations of the same calls to $\backslash_b$ and *simp*. Bit codes are built lazily using a purely functional data structure [18, 6].

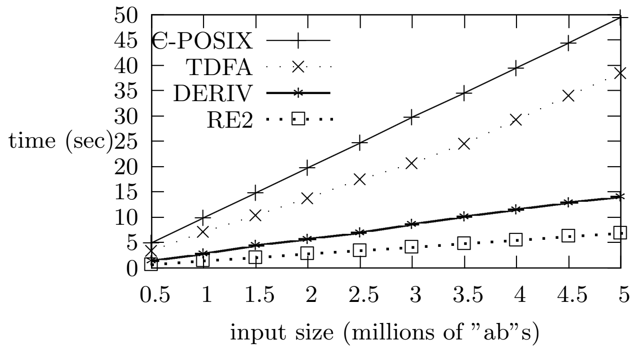
Experiments show that our implementation is competitive for inputs up to the size of about 10 Mb compared to highly-tuned C-based tools such as [2]. For larger inputs, our implementation is significantly slower (between 10-50 times) due to what seems to be high memory consumption. A possible solution is to use our method to compute the proper POSIX 'path' and then use this information to guide a space-efficient parsing algorithm such as [9] to build the POSIX parse tree. This is something we are currently working on.

For the specialized submatching case we have built another Haskell implementation referred to as **DERIV**. In DERIV, we only record the last match in case of Kleene star which is easily achieved in our implementation by 'overwriting' earlier with later matches.

We have benchmarked DERIV against three contenders which also claim to implement POSIX submatching: **TDFA**, a Haskell-based implementation [23] of



(a) Matching $(a + b + ab)^*$ with sequences of as



(b) Matching $(a + b + ab)^*$ with sequences of abs

Fig. 7. Ambiguous Pattern Benchmark

an adapted Laurikari-style tagged NFA. The original implementation [14] does not always produce the proper POSIX submatch and requires the adaptations described in [13]. **RE2**, the google C++ re2 library [2] where for benchmarking the option `RE2::POSIX` is turned on. **C-POSIX**, the Haskell wrapper of the default C POSIX regular expression implementation [22].

To our surprise, RE2 and C-POSIX report incorrect results, i.e. non-POSIX matches, for some examples. For RE2 there exists a prototype version [3] which appears to compute the correct POSIX match. We have checked the behavior for a few selected cases. Regardless, we include RE2 and C-POSIX in our experiments.

We have carried out an extensive set of benchmarks consisting of contrived as well as real-world examples which we collected from various sources, e.g. see [12, 17, 5]. The benchmarks were executed under Mac OS X 10.7.2 with 2.4GHz Core 2 Duo and 8GB RAM where results were collected based on the median over several test runs. The complete set of results as well as the implementation can be retrieved via [15]. A brief summary of our experimental results follows.

Overall our DERIV performs well and for most cases we beat TDFA and C-POSIX. RE2 is generally faster but then we are comparing a Haskell-based implementation against a highly-tuned C-based implementation.

Our approach suffers for cases where the size of a DFA is exponentially larger compared to the equivalent NFA. Most of the time is spent on building the DFA. The actual time spent on building the match is negligible. A surprisingly simple and efficient method to improve the performance of our approach is to apply some form of abstraction. Instead of trying to find matches for all subpattern locations, we may only be interested in certain locations. That is, we use the POSIX DFA only for subparts we are interested in. For subparts we don't care about, rely on an NFA.

For us the most important conclusion is that DERIV particularly performs well for cases where computation of the POSIX result is non-trivial. See Figure 7 which shows the benchmarks results for our example from the introduction. We see this as an indication that our approach is promising to compute POSIX results correctly *and* efficiently.

5 Related Work and Conclusion

The work in [7] studies like us the efficient construction of regular expression parse trees. However, the algorithm in [7] neither respects the Greedy nor the POSIX disambiguation strategy.

Most prior works on parsing and submatching focus on Greedy instead of POSIX. The greedy result is closely tied to the structure of the regular expression where priority is given to left-most expressions. Efficient methods for obtaining the greedy result transform the regular expression into an NFA. A 'greedy' NFA traversal then yields the proper result. For example, consider [14] for the case of submatching and [9, 8] for the general parsing case.

Adopting greedy algorithms to the POSIX setting requires some subtle adjustments to compute the POSIX, i.e. longest left-most, result. For example, see [4, 13, 19]. Our experiments confirm that our method particularly performs well for cases where there is a difference between the POSIX and Greedy result. By construction our method yields the POSIX result whereas the works in [4, 13, 19] require some additional bookkeeping (which causes overhead) to select the proper POSIX result.

The novelty of our approach lies in the use of derivatives. Regular expression derivatives [1] are an old idea and recently attracted again some interest in the context of lexing/parsing [20, 16]. We recently became aware of [26] which like us applies the idea of derivatives but only considers submatching.

To the best of our knowledge, we are the first to give an efficient algorithm for constructing POSIX parse trees including a formal correctness result. Our experiments show good results for the specialized submatching case. We are currently working on improving the performance for the full parsing case.

Acknowledgments

We thank referees for LATA'14 and FLOPS'14 for their helpful comments on earlier versions of this paper. We thank Christian Urban and Mark Sangster for their comments.

References

1. Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
2. Russ Cox. re2 – an efficient, principled regular expression library. `http://code.google.com/p/re2/`.
3. Russ Cox. NFA POSIX, 2007. `http://swtch.com/~rsc/regexp/nfa-posix.y.txt`.
4. Russ Cox. Regular expression matching: the virtual machine approach - digression: Posix submatching, 2009. `http://swtch.com/~rsc/regexp/regexp2.html`.
5. Russ Cox. Regular expression matching in the wild, 2010. `http://swtch.com/~rsc/regexp/regexp3.html`.
6. `http://hackage.haskell.org/package/dequeue-0.1.5/docs/Data-Dequeue.html`.
7. Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Inf.*, 37(2):121–144, 2000.
8. Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Proc. of ICALP'04*, pages 618– 629. Spinger-Verlag, 2004.
9. Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen, and Ulrik Terp Rasmussen. Two-pass greedy regular expression parsing. In *Proc. of CIAA'13*, volume 7982 of *LNCS*, pages 60–71. Springer, 2013.
10. Institute of Electrical and Electronics Engineers (IEEE): Standard for information technology – Portable Operating System Interface (POSIX) – Part 2 (Shell and utilities), Section 2.8 (Regular expression notation), New York, IEEE Standard 1003.2 (1992).
11. Chris Kuklewicz. Regex POSIX. `http://www.haskell.org/haskellwiki/Regex_Posix`.
12. Chris Kuklewicz. The regex-posix-unittest package. `http://hackage.haskell.org/package/regex-posix-unittest`.
13. Chris Kuklewicz. Forward regular expression matching with bounded space, 2007. `http://haskell.org/haskellwiki/RegexpDesign`.
14. Ville Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *SPIRE*, pages 181–187, 2000.
15. Kenny Z. M. Lu and Martin Sulzmann. POSIX Submatching with Regular Expression Derivatives. `http://code.google.com/p/xhaskell-regex-deriv`.
16. Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: a functional pearl. In *Proc. of ICFP'11*, pages 189–195. ACM, 2011.
17. Lasse Nielsen and Fritz Henglein. Bit-coded regular expression parsing. In *Proc. of LATA'11*, volume 6638 of *LNCS*, pages 402–413. Springer-Verlag, 2011.
18. Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.
19. Satoshi Okui and Taro Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In *Proc. of CIAA'10*, pages 231–240. Springer-Verlag, 2011.
20. Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, 2009.
21. PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.
22. regex-posix: The posix regex backend for regex-base. `http://hackage.haskell.org/package/regex-posix`.
23. regex-tdfa: A new all haskell tagged dfa regex engine, inspired by libtre. `http://hackage.haskell.org/package/regex-tdfa`.
24. Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives. In *Proc. of PPDP'12*, pages 79–90. ACM, 2012.
25. Stijn Vansummeren. Type inference for unique pattern matching. *ACM TOPLAS*, 28(3):389–428, May 2006.
26. Jérôme Vouillon. ocaml-re - Pure OCaml regular expressions, with support for Perl and POSIX-style strings. `https://github.com/avsm/ocaml-re`.

A Proof of Lemma 2

The proof that injection preserves POSIX parse trees requires a projection function:

```
$proj_{(l,l)} = \lambda\, \_ \,.\, ()$
$proj_{(r^*,l)} = \lambda\, (v : vs).\, (proj_{(r,l)}\ v,\ vs)$
$proj_{(r_1+r_2,l)} =$
  $\lambda\, v.$ case $v$ of
      $Left\ v_1 \rightarrow Left\ (proj_{(r_1,l)}\ v_1)$
      $Right\ v_2 \rightarrow Right\ (proj_{(r_2,l)}\ v_2)$
$proj_{(r_1\ r_2,l)} =$
  $\lambda\, (v_1, v_2).$
     if $|v_1| \not\models \epsilon$
     then if $\epsilon \in L(r_1)$
          then $Left\ (proj_{(r_1,l)}\ v_1,\ v_2)$
          else $(proj_{(r_1,l)}\ v_1,\ v_2)$
     else $Right\ (proj_{(r_2,l)}\ v_2)$
```

Injection and projection are inverses. Like injection, projection preserves POSIX parse trees and $proj_{(r,l)}$ shall only be applied on non-empty parse trees with the proper leading letter l.

Lemma 3 (Projection and Injection). *Let r be a regular expression, l a letter and v a parse tree.*

1. *If $\vdash\ v : r$ and $|v| = lw$ for some word w, then $\vdash\ proj_{(r,l)}\ v : r\backslash l$.*
2. *If $\vdash\ v : r\backslash l$ then $(proj_{(r,l)} \circ inj_{r\backslash l})\ v = v$.*
3. *If $\vdash\ v : r$ and $|v| = lw$ for some word w, then $(inj_{r\backslash l} \circ proj_{(r,l)})\ v = v$.*

MS:BUG[Come accross this issue when going back to our constructive reg-ex work] Consider $\vdash\ [Right\ (), Left\ a] : (a+\epsilon)^*$. However, $proj_{((a+\epsilon)^*,a)}\ [Right\ (), Left\ a]$ fails! The point is that $proj$ only works correctly if applied on POSIX parse trees.

MS:Possible fixes We only ever apply $proj$ on Posix parse trees.

For convenience, we write "$\vdash\ v : r$ is POSIX" where we mean that $\vdash\ v : r$ holds and v is the POSIX parse tree of r for word $|v|$.

Lemma 2 follows from the following statement.

Lemma 4 (POSIX Preservation under Injection and Projection). *Let r be a regular expression, l a letter and v a parse tree.*

1. *If $\vdash\ v : r\backslash l$ is POSIX, then $\vdash\ (inj_{r\backslash l}\ v) : r$ and $(inj_{r\backslash l}\ v)$ is POSIX.*
2. *If $\vdash\ v : r$ is POSIX. and $|v| = lw$ for some word w, then $\vdash\ (proj_{(r,l)}\ v) : r\backslash l$ is POSIX.*

Proof. There is a mutually dependency between statements (1) and (2). Both are proven by induction over r. We first verify statement (1) by case analysis.

– Case $r_1 + r_2$: We consider the possible shape of v.
  • First, we consider subcase $v = Right\ v_2$ where $\vdash\ Right\ v_2 : r_1\backslash l + r_2\backslash l$.

1. By assumption *Right* v_2 is the POSIX parse tree of $r_1\backslash l + r_2\backslash l$.
2. Hence, we can conclude that $\vdash v_2 : r_2\backslash l$ where v_2 is the POSIX parse tree of $r_2\backslash l$.
3. We are in the position to apply the induction hypothesis on $r_2\backslash l$ and find that $\vdash (inj_{r_2\backslash l}\ v_2) : r_2$ where $inj_{r_2\backslash l}\ v_2$ is the POSIX parse tree.
4. We immediately find that $\vdash Right\ (inj_{r_2\backslash l}\ v_2) : r_1 + r_2$.
5. What remains is to verify that *Right* $(inj_{r_2\backslash l}\ v_2)$ is the POSIX parse tree. Suppose the opposite. We distinguish among two cases (either there is POSIX 'right' or 'left' alternative).
   (a) i. Suppose there exists a POSIX parse tree *Right* v_2' such that $\vdash Right\ v_2' : r_1 + r_2$ and $v_2' \neq inj_{r_2\backslash l}\ v_2$ (*).
   ii. From (2) we obtain the POSIX parse tree $\vdash Right\ (proj_{(r_1+r_2,l)}\ v_2') : r_1\backslash l + r_2\backslash l$.
   iii. By assumption, *Right* v_2 is also POSIX.
   iv. Hence, $proj_{(r_1+r_2,l)}\ v_2' = v_2$.
   v. By application of (4) and the above we find that $v_2' = inj_{r_2\backslash l}\ v_2$ which yields a contradiction to (*).
   (b) i. Suppose there exists a POSIX parse tree *Left* v_1' such that $\vdash Left\ v_1' : r_1+r_2$ for some v_2' where it must hold that $|v_2'| = lw$ for some word w.
   ii. From (2) we obtain the POSIX parse tree $\vdash Left\ (proj_{(r_1+r_2,l)}\ v_2') : r_1\backslash l + r_2\backslash l$.
   iii. This contradicts our initial assumption that *Right* v_2 is the POSIX parse tree of $r_1\backslash l + r_2\backslash l$.

   In both cases, we have reached a contradiction. Hence, *Right* $(inj_{r_2\backslash l}\ v_2)$ is the POSIX parse tree.

- Subcase $v = Left\ v_3$ can be proven similarly. Hence, we can establish the induction step in case of alternatives.

– Case $r_1 r_2$: There are three possible subcases dictated by derivative operation. Either $v = (v_1, v_2)$, $v = Left\ (v_1, v_2)$ or $v = Right\ v_2$.

- First, we consider subcase $v = (v_1, v_2)$ where $\vdash (v_1, v_2) : (r_1\backslash l)r_2$. This implies that $\epsilon \notin L(r_1)$.
  1. By assumption (v_1, v_2) is POSIX. Hence, we can follow that $\vdash v_1 : r_1\backslash l$ is POSIX as well.
  2. We are in the position to apply the induction hypothesis and obtain that $\vdash (inj_{(r_1\backslash l)}\ v_1) : r_1$ is POSIX.
  3. It immediately follows that $\vdash (inj_{(r_1\backslash l)}\ v_1, v_2) : r_1 r_2$. What remains is to verify that this is the POSIX parse tree. We proceed again assuming the opposite.
     (a) Suppose there exists a POSIX parse tree (v_1', v_2').
     (b) This implies that either (i) $v_1' >_{r_1} inj_{(r_1\backslash l)}\ v_1$ or (ii) $v_1' = inj_{(r_1\backslash l)}\ v_1$ and $v_2' >_{r_2} v_2$.
     (c) Case (i) contradicts the fact that $inj_{(r_1\backslash l)}\ v_1$.
     (d) Hence, (ii) can only apply.
     (e) But then via (2) and (3) we can conclude that (v_1, v_2') is POSIX which contradicts our initial assumption that (v_1, v_2) is POSIX.
     (f) Hence, $(inj_{(r_1\backslash l)}\ v_1, v_2)$ is POSIX and so is $inj_{(r_1 r_2)\backslash l}(v_1, v_2)$.
- We consider the second subcase that $\vdash Left\ (v_1, v_2) : (r_1\backslash l)r_2 + r_2\backslash l$ is POSIX. For this case $\epsilon \in L(r_1)$. We conclude that $\vdash (v_1, v_2) :$

$(r_1\backslash l)r_2$ and using the same arguments as above we can verify that $inj_{(r_1r_2)\backslash l}(\mathit{Left}\ (v_1, v_2))$ is POSIX.

- For the third subcase, we find that $\vdash\ \mathit{Right}\ v_2 : (r_1\backslash l)r_2+r_2\backslash l$ is POSIX.
  1. Hence, $\vdash\ v_2 : r_2\backslash l$ POSIX and application of the induction hypothesis yields $\vdash\ (inj_{r_2\backslash l}\ v_2) : r_2$ is POSIX.
  2. We verify that $inj_{(r_1r_2)\backslash l}(\mathit{Right}\ v_2) = (mkEps_{r_1}, inj_{r_2\backslash l}\ v_2)$ is POSIX.
  3. Suppose the contrary. Then, there must be some POSIX (v_1', v_2') where $|v_1'| \neq \epsilon$.
  4. Application of (2) yields then some POSIX parse tree $\mathit{Left}\ v_3'$ of $(r_1\backslash l)r_2 + r_2\backslash l$ which contradicts the assumption that $\mathit{Right}\ v_2$ is POSIX.

In all three subcases we could establish the induction step which concludes the proof of case r_1r_2.

- Case r^*:
  1. By assumption we have that $\vdash\ (v, vs) : (r\backslash l, r^*)$ is POSIX which implies that $\vdash\ v : r\backslash l$ must be POSIX as well. (The case that $|(v, vs)| = \epsilon$ would require some special consideration. Ignored for brevity).
  2. Application of the induction hypothesis yields $\vdash\ (inj_{r\backslash l}\ v) : r$ is POSIX.
  3. By observing the POSIX ordering rules in Definition 1, we can conclude that $\vdash\ ((inj_{r\backslash l}\ v) : vs) : r^*$ is POSIX which establishes the induction step. The proof details are as follows. We first repeat the relevant ordering rules:

$$(\text{K1})\ \frac{|v : vs| = \epsilon}{[] >_{r^*} v : vs} \qquad (\text{K2})\ \frac{|v : vs| \neq \epsilon}{v : vs >_{r^*} []}$$

$$(\text{K3})\ \frac{v_1 >_r v_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2} \qquad (\text{K4})\ \frac{v_1 = v_2 \quad vs_1 >_{r^*} vs_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2}$$

  (a) Assume the contrary: $\vdash\ ((inj_{r\backslash l}\ v) : vs) : r^*$ is not POSIX.
  (b) Then, there must exists $\vdash\ (v'' : vs'') : r^*$ where $(v'' : vs'') >_{r^*} ((inj_{r\backslash l}\ v) : vs)$.
  (c) Rules (K1) and (K2) can be ignored. They, deal with the special [] case which does not apply due to injection of a letter in parse tree.
  (d) Suppose rule (K3) applies. Then, we find $v'' >_r (inj_{r\backslash l}\ v)$ which contradicts our assumption that $\vdash\ (inj_{r\backslash l}\ v) : r$ is POSIX.
  (e) Hence, the only remaining rule is (K4) which implies that $v'' = inj_{r\backslash l}\ v$ and $vs'' >_{r^*} vs$ (*). Our goal is to contradict (*).
  (f) By applying the projection function we obtain $proj_{(r,l)}\ v'' = v$ which in combination with $\vdash\ (v'' : vs'') : r^*$ yields $\vdash\ (v, vs'') : (r\backslash l, r^*)$ (**). Follows from Lemma 3.
  (g) By assumption $\vdash\ (v, vs) : (r\backslash l, r^*)$ is POSIX. Then, from (**) and via rule (K4) we obtain that $vs >_{r^*} vs''$.
  (h) The above statement contradicts (*). Hence, we are done.

The remaining cases for l and ϵ are trivial.

Next, we consider statement (2) and proceed again by case analysis.

- Case $r_1 + r_2$. There are two possible subcases. Either $v = \mathit{Right}\ v_2$ or $v = \mathit{Left}\ v_2$.
  We first consider that $\vdash\ \mathit{Right}\ v_2 : r_1 + r_2$ is POSIX.

1. We conclude that $\vdash v_2 : r_2$ is POSIX.
2. Application of the induction hypothesis yields $\vdash (\mathit{proj}_{(r_2,l)}\ v_2) : r_2\backslash l$ is POSIX.
3. What remains is to show that $\vdash$ *Right* $(\mathit{proj}_{(r_2,l)}\ v_2) : r_2\backslash l + r_1\backslash l$ is POSIX. Suppose the opposite.
   (a) It is straightforward to reach a contradiction in case there is a POSIX 'right' alternative *Right* v_2'.
   (b) Hence, there must exist $\vdash$ *Left* $v_1' : r_2\backslash l + r_1\backslash l$ such that *Left* v_1' is POSIX.
   (c) By application of (1), we find that $\vdash$ *Left* $(\mathit{inj}_{r_1\backslash l}\ v_1) : r_1 + r_2$ which contradicts our initial assumption that *Right* v_2 is POSIX.

   Hence, *Right* $(\mathit{proj}_{(r_2,l)}\ v_2)$ is POSIX which establishes the induction step for this subcase.

Subcase *Left* v_2 can be proven similarly.

– Case r_1r_2:
1. By assumption $v = (v_1, v_2)$ and $\vdash (v_1, v_2) : r_1r_2$ is POSIX which implies that $\vdash v_1 : r_1$ is POSIX.
2. We consider the possible cases of $|v_1|$.
3. Suppose $|v_1| \neq \epsilon$.
   (a) By application of the induction hypothesis we obtain $\vdash (\mathit{proj}_{(r_1,l)}\ v_1) : r_1\backslash l$ is POSIX.
   (b) The above implies $\vdash (\mathit{proj}_{(r_1,l)}\ v_1, v_2) : (r_1\backslash l)r_2$. We are done if $\epsilon \notin L(r_1)$.
   (c) Otherwise, it is straightforward to verify that $\vdash$ *Left* $(\mathit{proj}_{(r_1,l)}\ v_1, v_2) : (r_1r_2)\backslash l$.
   (d) Thus, we establish the induction step under the given assumption.
4. Otherwise, $|v_1| = \epsilon$ which implies $\epsilon \in L(r_1)$.
   (a) By induction we find $\vdash (\mathit{proj}_{(r_2,l)}\ v_2) : r_2\backslash l$ is POSIX.
   (b) What remains is to show that $\vdash$ *Right* $(\mathit{proj}_{(r_2,l)}\ v_2) : (r_1\backslash l)r_2 + r_2\backslash l$ is POSIX. Suppose the opposite.
      i. It is straightforward to reach a contradiction in case there is a POSIX 'right' alternative *Right* v_2'.
      ii. Hence, there must exist $\vdash$ *Left* $(v_1', v_2') : (r_1\backslash l)r_2 + r_2\backslash l$ and *Left* (v_1', v_2') is POSIX.
      iii. From (1) we then conclude that $\vdash (\mathit{inj}_{r_1\backslash l}\ v_1', v_2') : r_1r_2$ is POSIX.
      iv. This contradicts the assumption that (v_1, v_2) is POSIX and $|v_1| = \epsilon$.
      v. Thus, we establish the induction step under the given assumption and are done.

– Case r^*:
1. By assumption $\vdash (v : vs) : r^*$ is POSIX where $|v| \neq \epsilon$.
2. Application of the induction hypothesis yields that $\vdash (\mathit{proj}_{(r,l)}\ v) : r\backslash l$ is POSIX.
3. Immediately, we find that $\vdash ((\mathit{proj}_{(r,l)}\ v, vs) : (r\backslash l)r^*$ is POSIX which establishes the induction step.

The remaining case for l is trivial.