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. 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 relate parsing to the more specific submatching problem (Section 2.1).
- We present a method for computation of POSIX parse trees based on Brzozowski's regular expression derivatives [1]. We formally verify its correctness and establish a linear run-time complexity (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. Experiments confirm that our method performs well in practice (Section 4).

Section 5 discusses related work and concludes. The (optional) Appendix contains supplementary material such as formal proofs.

2 Regular Expressions and Parse Trees

Words:

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

Regular expressions:

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

Parse trees:

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

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

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

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

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

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

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

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

Flattening:

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

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 literals 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 *Right Left* b as a short-hand for *Right* (*Left* b). The derivation trees are shown below:

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

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

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. Our POSIX parse tree order is derived from a POSIX submatching order described in [24]. The connection between parsing and submatching will be highlighted in the up-coming Section 2.1.

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 rulres are as follow*

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

$$\textit{(P1)}\ \frac{\textit{len}\ |v_2| > \textit{len}\ |v_1|}{\textit{Right}\ v_2 >_{r_1+r_2} \textit{Left}\ v_1} \qquad \textit{(P2)}\ \frac{\textit{len}\ |v_1| \geq \textit{len}\ |v_2|}{\textit{Left}\ v_1 >_{r_1+r_2} \textit{Right}\ v_2}$$

$$\frac{v_2 >_{r_2} v_2'}{\textit{Right}\ v_2 >_{r_1+r_2} \textit{Right}\ v_2'} \qquad \frac{v_1 >_{r_1} v_1'}{\textit{Left}\ v_1 >_{r_1+r_2} \textit{Left}\ v_1'}$$

$$\textit{(S1)}\ \frac{|v : vs| = \epsilon}{[] >_{r^*} v : vs} \qquad \textit{(S2)}\ \frac{|v : vs| \neq \epsilon}{v : vs >_{r^*} []}$$

$$\frac{v_1 >_r v_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2} \qquad \frac{v_1 = v_2 \quad vs_1 >_{r^*} vs_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2}$$

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 gives preference to longest left-most parse trees. This is easy to see for cases $r_1 r_2$ and r^*. More interesting is $r_1 + r_2$. If the underlying word is strictly longer, we give preference to the right alternative. See (P1). For our running example, we find that

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

Subcase (P2) guarantees that we strictly give preference to the left alternative as long as the underlying matched word is longer or equal. This is important and guarantees that there is not infinite chain of "larger" parse trees. For example, consider $(\epsilon + a)^*$ where for input a we find the following infinite chain of parse trees

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

Clearly, v_0 is the largest parse tree according to our ordering relation. This would be no longer the case if we would drop the side in (P1). Then, each v_{i+1} would be larger than v_i. Hence, we impose the side condition $\textit{len}\ |v_1| \geq \textit{len}\ |v_2|$ in (P2).

Let's consider the Kleene star case. Subcase (S1) gives preference to [] if elements in a list of parse trees match the empty string. Otherwise, we favor a non-empty parse tree, see subcase (S2), or perform an element-wise comparison, see the remaining two cases.

Like in case of alternation, subcases (S1) and S2 rule out infinite chains of "larger" parse trees when comparing parse trees which match the empty string. For example, consider ϵ^* and the empty input for which 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. This would be no longer the case if we would replace (S1) and (S2) by $v : vs >_{r^*} []$. Then, each v_{i+1} would be larger than v_i. Hence, we find subcases (S1) and (S2).

To summarize, Definition 1 yields a well-defined POSIX order among parse trees which enjoys maximal elements. That is, infinite chains of "larger" parse trees are ruled out. The order is also total for parse trees v_1, v_2 of a regular expression r where v_1 and v_2 share the same underlying word. This can be verified by structural induction over r and by observing the various cases for v_1 and v_2.

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

2.1 Parsing versus Submatching

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

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

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

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

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

Fig. 2. From Parsing to Submatching

In practice, we rarely require the full parse tree. Often, we only care about certain subparts and only record the last match in case of a Kleene star iteration for space reasons. 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 $x \mapsto aa$. In contrast, here is the parse tree resulting from the match against the input word *abaacc*

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

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):

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

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 $\{x \mapsto \epsilon\}$.

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 tree in linear time (in the size of the input string).

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:

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

In the above, we write $r \xrightarrow{l} r'$ for applying the derivative operation on r where r' equals $r\backslash l$.

Regular expression derivatives:

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

Parse tree transformation:

```
$mkEps_{r^*} = []$
$mkEps_{r_1 r_2} = (mkEps_{r_1}, mkEps_{r_2})$
$mkEps_{r_1+r_2}$
    $|\epsilon \in L(r_1) = Left\ mkEps_{r_1}$
    $|\epsilon \in L(r_2) = Right\ mkEps_{r_2}$
$mkEps_\epsilon = ()$
```

```
$inj_{r^*\backslash l}(v, vs) = (inj_{r\backslash l}\ v) : vs$
$inj_{(r_1 r_2)\backslash l} =$
  $\lambda v.$case $v$ of
      $(v_1, v_2) \rightarrow (inj_{r_1\backslash l}\ v_1, v_2)$
      $Left\ (v_1, v_2) \rightarrow (inj_{r_1\backslash l}\ v_1, v_2)$
      $Right\ v_2 \rightarrow (mkEps_{r_1}, inj_{r_2\backslash l}\ v_2)$
$inj_{(r_1+r_2)\backslash l} =$
  $\lambda v.$case $v$ of
      $Left\ v_1 \rightarrow Left\ (inj_{r_1\backslash l}\ v_1)$
      $Right\ v_2 \rightarrow Right\ (inj_{r_2\backslash l}\ v_2)$
$inj_{l\backslash l}() = l$
```

Parsing with derivatives:

```
$parse\ r\ \epsilon$
    $|\epsilon \in L(r) = mkEps_r$
$parse\ r\ lw = inj_{r\backslash l}(parse\ r\backslash l\ w)$
```

Fig. 3. Parsing Tree Construction with Derivatives

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 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 $r_i \xrightarrow{l} r_{i+1}$. In the above, this step is denoted $v_i \xleftarrow{l} v_{i+1}$. Thus, we incrementally build the parse tree v_0 for the initial expression r_0.

This method yields POSIX parse trees because the derivative matching pass extracts the longest left-most sequence of letters $l_1...l_n$ from r_0. The injection pass simply reverses this effect and (as we will see) by construction maintains POSIX parse trees. For efficiency reasons, it is entirely possible to perform the 'backward' construction of POSIX parse trees during the 'forward' matching pass. For presentation purposes, we describe both passes separately.

Figure 3 summarizes our method for construction of POSIX parse trees based on the above idea. To explain our method in more detail, we use a simple running example. For expression $(a + ab)(b + \epsilon)$ and word ab it is easy to see that the POSIX parse tree is $(\mathit{Right}\ (a, b), \mathit{Right}\ ())$.

Let's apply the 'forward' matching pass on our example:

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

In detail, the last step $\xrightarrow{b}$ is as follows:

$$\begin{array}{l} \quad ((\epsilon + \epsilon b)(b + \epsilon))\backslash b \\ = ((\epsilon + \epsilon b)\backslash b)(b + \epsilon) + (b + \epsilon)\backslash b \\ = (\epsilon\backslash b + (\epsilon b)\backslash b)(b + \epsilon) + (b\backslash b + \epsilon\backslash b) \\ = (\phi + ((\epsilon\backslash b)b + b\backslash b))(b + \epsilon) + (\epsilon + \phi) \\ = (\phi + (\phi b + \epsilon))(b + \epsilon) + (\epsilon + \phi) \end{array}$$

Next, we check that the final expression $(\phi+(\phi b+\epsilon))(b+\epsilon)+(\epsilon+\phi)$ is nullable which is the case here. Computing a POSIX tree for a nullable expression is performed by recursing over the regular expression structure and strictly favoring left branches. See function mkEps_r in Figure 3. For example,

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

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'.

Transformation of parse trees is carried out by function $\mathit{inj}_{r\backslash l}$ in Figure 3. This function takes as an input a parse tree of the derivative $r\backslash l$ and yields a parse of r by (re)injecting the extracted letter l. Thus, we can define the transformation step $v_i \overset{l}{\leftarrow} v_{i+1}$ by $v_i = \mathit{inj}_{r_i\backslash l}\ v_{i+1}$. Importantly, the definition of inj follows closely the structure of the derivative operation $\cdot\backslash\cdot$. This guarantees that injection maintains POSIX parse trees.

We take a closer look at the definition inj. For example, the most simple (last) case is $\mathit{inj}_{l\backslash l}() = l$ 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 $\mathit{inj}_{r\backslash l}$ is applied recursively on v to yield a parse tree for r.

Concatenation $r_1 r_2$ is the most involved case. There are three possible subcases. The first subcases 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 $(r_1 r_2)\backslash l = (r_1\backslash l)r_2$. 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 $(r_1 r_2)\backslash l = (r_1\backslash l)r_2 + r_2\backslash l$. 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 $inj_{r_2\backslash l}$.

By construction, $inj_{r\backslash l}$ applied on a parse tree of $r\backslash l$ yields a parse tree of r. The important property for us is that injection maintains POSIX parse trees. The intuition is that the matching pass removes the leading (left-most) letters. The derivative operation maintains the structure of expressions and therefore guarantees that we extract the longest left-most sequence of letters. The injection function simply reverses this effect.

Let's consider application of injection for our running example. Recall that the input parse tree $(Left\ (Right\ (Right\ ()), Right\ ()))$ is the POSIX parse computed via $mkEps_{(\phi+(\phi b+\epsilon))(b+\epsilon)+(\epsilon+\phi)}$.

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

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

Another application step yields

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

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

We formally state that our method yields POSIX parse trees.

Lemma 2 (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.*

The proof is by simple induction over the structure of r.

Lemma 3 (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 $|(inj_{r\backslash l}\ v)| = l|v|$.*

The proof is given in the Appendix.

Based on the above lemmas we reach the following result.

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

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^*.

To identify similar derivatives, the work in [1] identifies three rewrite rules to simplify derivatives:

$$(1)\ r + r \Rightarrow r \quad (2)\ r_2 + r_1 \Rightarrow r_1 + r_2 \text{ where } r_1 < r_2$$

$$(3)\ (r_1 + r_2) + r_3 \Rightarrow r_1 + (r_2 + r_3)$$

where $r_1 < r_2$ establishes a structural ordering among expression. As shown in [1], the set of simplified, w.r.t. rewrite rules (1-3), derivatives as well as their size is finite.

In our setting, applying a simplification $r_1 \Rightarrow r_2$ requires to transform r_2's parse tree into a parse tree of r_1. Of course, we wish to maintain POSIX parse trees. For (1) and (3) it is straightforward to define such transformations. For (2) this will not be possible because POSIX is clearly not stable under the commutativity law. For example, consider expressions $a^* + a$ and $a + a^*$ for which we find different POSIX parse trees for word a.

Plainly abandoning (2) will not work for cases such as $(r_1+r_2)+r_1$ where we wish to simplify the expression to r_1+r_2. The solution is as follows. Expressions are first put into right-associative normal form, e.g. $r_1 + r_2 + r_1$ which stands for $r_1 + (r_2 + r_1)$. Then, we simply apply a more general variant of (1) which will directly simplify $r_1 + r_2 + r_1$ to $r_1 + r_2$.

In Figure 4 we define a function *simp* which takes a regular expression r and yields an expression r' and function f where f transforms r' parse tree into a parse tree of r. The simplifications effectively code up the rewrite rules (1-3) and are carefully chosen such that we maintain POSIX parse trees. The notation

$$(r_1 \ + \ ... \ + \ r_{i-1} \ + \ r_i \ + \ r_{i+1} \ + \ ... \ + \ r_n) \ \text{where} \ (r_1 \ == \ r_{i+1}) \ \rightarrow$$

means that we check for pattern $(r_1 + ... + r_{i-1} + r_i + r_{i+1} + ... + r_n)$ which additionally satisfies the guard condition $(r_1 == r_{i+1})$. We write Right^k as a short-hand for k-nested applications of *Right*. For example, case $(**)$ simplifies $r_1 + (r_2 + r_1)$ to $r_1 + r_2$.

Lemma 4 (POSIX Preservation under Simplifications). *Let r,r' be regular expressions, v' a parse tree, f a transformation function among parse trees such that simp $r = (r', f)$, $\vdash v' : r'$ and v' is the POSIX parse tree of r' for word $|v'|$. Then, $\vdash f v' : r$ and $f v'$ is the POSIX parse tree of r for word $|v'|$.*

Lemma 5 (Finite Number of Derivatives [1]). *The set and size of derivatives which are dissimilar with respect to simplifications in Figure 4 is finite.*

Next, we consider the complexity of our parsing approach. It is easy to see that each call of one of these functions leads to subcalls whose number is bound by the size of the regular expression involved. We assume that the parse tree values are kept in place and *not* copied. For example, recall the injection case for Kleene star

```
$simp\ r_1 r_2 =$ let $(r_1', f_1) = simp\ r_1$
                  $(r_2', f_2) = simp\ r_2$
              in $(r_1' r_2', \lambda.(v_1', v_2').(f_1\ v_1', f_2\ v_2'))$
$simp\ r =$
  case $r$ of
  $((r_1 + r_2) + r_2) \rightarrow$
    $(r_1 + (r_2 + r_3), \lambda v.$case $v$ of
                                $Left\ v_1 \rightarrow Left\ (Left\ v_1)$
                                $Right\ (Left\ v_2) \rightarrow Left\ (Right\ v_2)$
                                $Right\ (Right\ v_3) \rightarrow Right\ v_3)$
  $(r_1 + ... + r_{i-1} + r_i + r_{i+1} + ... + r_n)$ where $(r_1 == r_{i+1}) \rightarrow$
    $((r_1 + ... + r_{i-1} + r_{i+1} + ... + r_n), \lambda v.$case $v$ of
                                          $Right^{i-1}\ v' \rightarrow Right^{i}\ v'$
                                          $v' \rightarrow v')$
  $(r_1 + ... + r_{i-1} + r_i)$ where $(r_1 == r_{i+1} \&\& i == 1) \rightarrow$
    $(r_1, \lambda v.Left\ v)$
  $(r_1 + ... + r_{i-1} + r_i)$ where $(r_1 == r_{i+1} \&\& i > 2) \rightarrow$             $(**)$
    $(r_1 + ... + r_{i-1}, \lambda v.$case $v$ of
                        $Right^{i-2}\ v' \rightarrow Right^{i-2}\ (Left\ v')$
                        $v' \rightarrow v')$
  $(r_1 + r_2) \rightarrow$
    let $(r_1', f_1) = simp\ r_1$
        $(r_2,', f_2) = simp\ r_2$
    in $(r_1' + r_2', \lambda.v.$case $v$ of
                        $Left\ v_1' \rightarrow Left\ (f_1 v_1')$
                        $Right\ v_2' \rightarrow Right\ (f_2 v_2'))$
$simp\ r = (r, \lambda v.v)$
```

Fig. 4. Simplifications

$$inj_{r^*\backslash l}\,(v, vs) \;=\; (inj_{r\backslash l}\;\; v) \;:\; vs$$

where value vs is kept in place and the resulting parse tree $(inj_{r\backslash l}\ v) : vs$ maintains a pointer to the original value. Thus, we can argue that functions $mkEps$, inj and $simp$ are constant time operations.

Lemma 6 (Parse Tree Transformation in Constant Time). *Functions mkEps, inj and simp are constant time operations assuming that (a) we treat the size of regular expressions as a constant and (b) parse trees are not copied but rather kept in place.*

We obtain a linear-time POSIX parsing method by aggressively performing simplifications.

```
$parseSimp\ r\ \epsilon$
  $\mid \epsilon \in L(r) \;=\; mkEps_r$
$parseSimp\ r\ lw \;=\;$ let $(r', f) \;=\; simp\ r$
                    in $f \circ (inj_{r'\backslash l}\,(parse\ r'\backslash l\ w))$
```

Theorem 2 (POSIX Linear Run-Time). *Function parseSimp computes POSIX parse trees in linear time in the size of the input.*

In practice, further simplifications such as $\epsilon r \Rightarrow r$, $\phi r \Rightarrow \phi$ etc may yield even 'smaller' derivatives. For example, see [20] for an extensive list of simplifications. In our setting, we of course need to be careful that simplifications and their associated parse tree transformers maintain the POSIX property and still guarantee the constant time property of Lemma 6.

For example, the following simplification breaks the constant time assumption.

$$\begin{array}{lll} simp\ r^* & = & \mathsf{let}\ (r_1, f)\ =\ simp\ r \\ & & \mathsf{in}\ ({r_1}^*,\ \lambda\, vs.\, map\ \ f\ \ vs) \end{array}$$

We simplify the expression below a Kleene star and therefore are required to traverse the entire sequence $[v_1, ..., v_n]$ of parse tree results of r.

Fortunately, performing simplification below Kleene star is strictly not necessary because we never generate such an expression. Therefore, function *simp* in Figure 4 recurses over the structure of the expression with the exception of the Kleene star which we leave untouched. Obviously, we could assume that this simplification step is only applied on the initial regular expression.

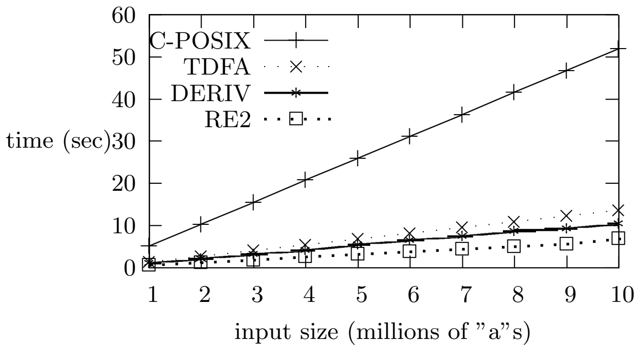
4 Experiments

We have implemented the derivative-based 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 *mkEps*, *inj* and *simp*. Instead of applying a sequence of 'backwards' transformation steps on the final (empty) parse tree, we incrementally build up the POSIX parse tree during the matching pass. Following [17], we use a space efficient bit-code representation of parse trees (see Appendix for details). Bit codes are built lazily using a purely functional data structure [18, 6].

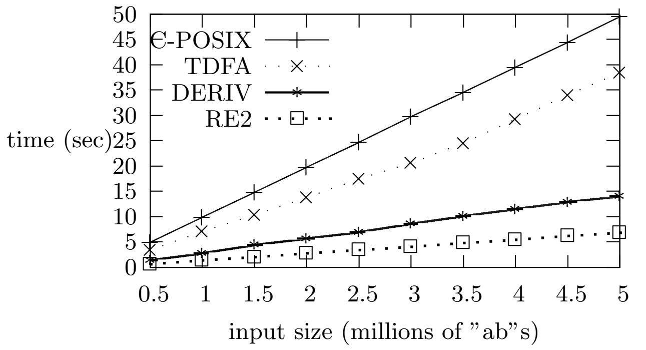
Experiments show that our implementation is competitive for inputs up to the size of about 10 Mb compared to highly-tuned C-based tools such as [2]. For larger inputs, our implementation is significantly slower (between 10-50 times) due to what seems high memory consumption. A possible solution is to 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 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



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



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

Fig. 5. Ambiguous Pattern Benchmark

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 5 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, which can be done in linear time, 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 POSIX and greedy. 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 [25] 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 algorithm for constructing POSIX parse trees in linear time 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 for their helpful comments on an earlier version 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. Stijn Vansummeren. Type inference for unique pattern matching. *ACM TOPLAS*, 28(3):389–428, May 2006.
25. Jérôme Vouillon. ocaml-re - Pure OCaml regular expressions, with support for Perl and POSIX-style strings. `https://github.com/avsm/ocaml-re`.

Optional material such as formal proofs

A Proof of Lemma 3

The formal 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 7 (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 $(inj_{r\backslash l} \circ proj_{(r,l)})\ v = v$.*

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 3 follows from the following statement.

Lemma 8 (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_1r_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 $\vdash (inj_{(r_1\backslash l)}\ v_1, v_2) : r_1r_2$. 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 (a) $v'_1 >_{r_1} inj_{(r_1\backslash l)}\ v_1$ or (b) $v'_1 = inj_{(r_1\backslash l)}\ v_1$ and $v'_2 >_{r_2} v_2$.
     (c) Case (a) contradicts the fact that $inj_{(r_1\backslash l)}\ v_1$.
     (d) Hence, (b) 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 $inj_{(r_1r_2)\backslash l}(v_1, v_2)$.
- 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 $inj_{(r_1r_2)\backslash l}($*Left* $(v_1, v_2))$ is POSIX.

- For the third subcase, we find that $\vdash$ *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 $inj_{(r_1 r_2)\backslash l}(Right\ v_2) = (mkEps_{r_1}, inj_{r_2\backslash l}\ v_2)$ 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 *Left* v_3' of $(r_1\backslash l)r_2 + r_2\backslash l$ which contradicts the assumption that *Right* v_2 is POSIX.

  In all three subcases we could establish the induction step which concludes the proof of case $r_1 r_2$.
- 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:

$$\text{(S1)}\ \frac{|v : vs| = \epsilon}{[] >_{r^*} v : vs} \qquad \text{(S2)}\ \frac{|v : vs| \neq \epsilon}{v : vs >_{r^*} []}$$

$$\text{(S3)}\ \frac{v_1 >_r v_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2} \qquad \text{(S4)}\ \frac{v_1 = v_2 \quad vs_1 >_{r^*} vs_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2}$$

  (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 (S1) and (S2) 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 (S3) 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 (S4) which implies that $v'' = inj_{r\backslash l}\ v$ and $vs'' >_{r^*} vs$ (*). Out 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 7.
  (g) By assumption $\vdash (v, vs) : (r\backslash l, r^*)$ is POSIX. Then, from (**) and via rule (S4) 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 $v =$ *Right* v_2 or $v =$ *Left* v_2.
  We first consider that $\vdash$ *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 $\vdash \mathit{Right}\ (\mathit{proj}_{(r_2,l)}\ v_2) : r_2\backslash l + r_1\backslash l$ is POSIX. Suppose the opposite.
   (a) It is straightforward to reach a contradiction in case there is a POSIX 'right' alternative $\mathit{Right}\ v_2'$.
   (b) Hence, there must exist $\vdash \mathit{Left}\ v_1' : r_2\backslash l + r_1\backslash l$ such that $\mathit{Left}\ v_1'$ is POSIX.
   (c) By application of (1), we find that $\vdash \mathit{Left}\ (\mathit{inj}_{r_1\backslash l}\ v_1) : r_1 + r_2$ which contradicts our initial assumption that $\mathit{Right}\ v_2$ is POSIX.

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

Subcase $\mathit{Left}\ v_2$ can be proven similarly.

– Case $r_1 r_2$:
  1. By assumption $v = (v_1, v_2)$ and $\vdash (v_1, v_2) : r_1 r_2$ 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 $\vdash \mathit{Left}\ (\mathit{proj}_{(r_1,l)}\ v_1, v_2) : (r_1 r_2)\backslash l$.
     (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 $\vdash \mathit{Right}\ (\mathit{proj}_{(r_2,l)}\ v_2) : (r_1\backslash l) r_2 + r_2\backslash l$ is POSIX. Suppose the opposite.
        i. It is straightforward to reach a contradiction in case there is a POSIX 'right' alternative $\mathit{Right}\ v_2'$.
        ii. Hence, there must exist $\vdash \mathit{Left}\ (v_1', v_2') : (r_1\backslash l) r_2 + r_2\backslash l$ and $\mathit{Left}\ (v_1', v_2')$ is POSIX.
        iii. From (1) we then conclude that $\vdash (\mathit{inj}_{r_1\backslash l}\ v_1', v_2') : r_1 r_2$ 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.

B Incremental Bit-Coded Forward Parse Tree Construction

We describe a refined parse tree construction method where

- we use bit-codes to represent parse trees, and
- we incrementally build up parse trees while matching.

The refinement can be directly derived from Figure 3.

Regular expressions are now annotated with bit codes:

$$\begin{array}{rll}
b & ::= 0 \mid 0 & \\
bs & ::= [] \mid b : bs & \text{Bit-codes} \\
r & ::= (bs : l) & \\
 & \mid\ (bs : r^*) & \text{Kleene star} \\
 & \mid\ (bs : rr) & \text{Concatenation} \\
 & \mid\ (bs : r + r) & \text{Choice} \\
 & \mid\ (bs : r \oplus r) & \text{Internal Choice} \\
 & \mid\ (bs : \epsilon) & \text{Empty word} \\
 & \mid\ \phi & \text{Empty language}
\end{array}$$

The 'internal' choice represents an expression where one of the alternatives shall be selected without keeping track if it is the left or right alternative. Its exact purpose will be clear shortly.

Given a bit code sequence and a regular expression, we can straightforwardly compute the parse tree.

```
$decode_r\ bs\ =$ let $(v, p)\ =\ decode'_r\ bs$
                  in case $p$ of
                     $[] \rightarrow v$
$decode'_\epsilon\ bs\ =\ ((),\ bs)$
$decode'_l\ bs\ =\ (l,\ bs)$
$decode'_{r_1\ +\ r2}\ (0 : bs)\ =$ let $(v, p)\ =\ decode'_{r_1}\ bs$
                             in $(Left\ v,\ p)$
$decode'_{r_1\ +\ r2}\ (1 : bs)\ =$ let $(v, p)\ =\ decode'_{r_1}\ bs$
                             in $(Right\ v,\ p)$
$decode'_{r_1\ r2}\ bs\ =$ 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 $((v : vs),\ p_2)$
$decode'_{r^*}\ (1 : bs)\ =\ ([],\ bs)$
```

The internal choice case is not relevant here. Function $decode_r$ will be applied on the 'original' expression r which does not carry any bit code information.

Like in the earlier 'injection' approach, we must compute the bit code of a nullable expression.

$$mkEpsBC_{(bs\ :\ r^*)}\ =\ bs\ \text{++}\ [1]$$

$$
\begin{array}{l}
mkEps_{(\,bs\,:\,r_1\;r_2)} \;=\; bs \mathbin{++} mkEps_{r_1} \mathbin{++} mkEps_{r_2})\\
mkEps_{(bs\,:\,r_1+r_2)}\\
\quad \mid \epsilon \in L(r_1) \;=\; bs \mathbin{++} (0 : mkEps_{r_1})\\
\quad \mid \epsilon \in L(r_2) \;=\; bs \mathbin{++} (1 : mkEps_{r_2})\\
mkEps_{(bs\,:\,r_1 \oplus r_2)}\\
\quad \mid \epsilon \in L(r_1) \;=\; bs \mathbin{++} mkEps_{r_1}\\
\quad \mid \epsilon \in L(r_2) \;=\; bs \mathbin{++} mkEps_{r_2}\\
mkEps_{(bs\,:\,\epsilon)} \;=\; bs
\end{array}
$$

The internal choice case may now arise but we do not record any 'direction' information.

The main difference to the 'injection' approach is that bit-coded parse tree information is computed during the application of the derivative operation.

$$
\begin{array}{ll}
\phi\backslash_b l & = \phi\\
(bs : \epsilon)\backslash_b l & = \phi\\
(bs : l_1)\backslash_b l_2 & = \begin{cases} (bs : \epsilon) \text{ if } l_1 == l_2\\ \phi \qquad\; \text{otherwise}\end{cases}\\
(bs : r_1 + r_2)\backslash_b l & = (bs{++}[0] : r_1)\backslash_b l \oplus (bs{++}[1] : r_2)\backslash l\\
(bs : r_1 r_2)\backslash_b l & = \begin{cases} (bs : (r_1\backslash_b l) r_2) \oplus\; (fuse\; mkEpsBC_{r_1}\; (r_2\backslash_b l)) \text{ if } \epsilon \in L(r_1)\\ (bs : (r_1\backslash_b l)) r_2 \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise}\end{cases}\\
r^*\backslash l & = (r\backslash l) r^*
\end{array}
$$

The reason for $\oplus$ becomes now apparent in case of concatenation. For a nullable expression, there are two possible cases which are tried. As we now move *forward*, we must compute the bit-code representation $mkEpsBC_{r_2}$ of the nullable expression r_2. We attach this information to the top-most bit-code annotation in expression r_1 via helper function *fuse*.

$$
\begin{array}{l}
fuse\; bs\; \phi \;=\; \phi\\
fuse\; bs\; (p\; :\; \epsilon) \;=\; (bs \mathbin{++} p\; :\; \epsilon)\\
fuse\; bs\; (p\; :\; l) \;=\; (bs \mathbin{++} p\; :\; l)\\
fuse\; bs\; (p\; :\; r_1\; +\; r_2) \;=\; (bs \mathbin{++} p\; :\; r_1\; +\; r_2)\\
fuse\; bs\; (p\; :\; r_1\; \oplus\; r_2) \;=\; (bs \mathbin{++} p\; :\; r_1\; \oplus\; r_2)\\
fuse\; bs\; (p\; :\; r_1\; r_2) \;=\; (bs \mathbin{++} p\; :\; r_1\; r_2)\\
fuse\; bs\; (p\; :\; r^*) \;=\; (bs \mathbin{++} p\; :\; r^*)
\end{array}
$$

The correctness of the bit-codes accumulated by $\cdot\backslash_b\cdot$ can be easily argued by direct correspondence to inj. Instead of a constructing the parse tree 'backwards', we simply now strictly move 'forward'.

It is also straightforward to incorporate simplifications on expressions which carry bit-codes. For example, consider the rule for turning alternatives into right associativity form.

$$
\begin{array}{l}
simpBC\; (p_1\; :\; (p_2\; :\; r_1\; +\; r_2)\; +\; r_3) \;=\\
\quad [\,]\; :\; (fuse\; (p_1 \mathbin{++} p_2 \mathbin{++} [0,0])\; r_1)\\
\qquad\quad \oplus\; ((fuse\; (p_1 \mathbin{++} p_2 \mathbin{++} [0,1])\; r_2)\; \oplus\; (fuse\; (p_1 \mathbin{++} [1])\; r_3))
\end{array}
$$

For convenience, we combine alternatives via the 'internal' choice operator and record the parse tree information in the bit-code annotations.

Like before, we repeatedly apply the derivative operation and perform simplifications. To extract the bit-code for the original expression we simply retrieve the bit-codes from the final (nullable) expression.

$$
\begin{array}{l}
\mathit{retrieve}_{(p\,:\,\epsilon)} \;=\; p \\
\mathit{retrieve}_{(p\,:\,r_1\,+\,r_2)} \\
\quad \mid \epsilon \in L(r_1) \;=\; p \mathbin{{+}{+}} (0 \;:\; \mathit{retrieve}_{r_1}) \\
\quad \mid \epsilon \in L(r_2) \;=\; p \mathbin{{+}{+}} (0 \;:\; \mathit{retrieve}_{r_2}) \\
\mathit{retrieve}_{(p\,:\,r_1\,\oplus\,r_2)} \\
\quad \mid \epsilon \in L(r_1) \;=\; p \mathbin{{+}{+}} \mathit{retrieve}_{r_1} \\
\quad \mid \epsilon \in L(r_2) \;=\; p \mathbin{{+}{+}} \mathit{retrieve}_{r_2} \\
\mathit{retrieve}_{(p\,:\,r_1\;r_2)} \;=\; p \mathbin{{+}{+}} \mathit{retrieve}_{r_1} \mathbin{{+}{+}} \mathit{retrieve}_{r_2} \\
\mathit{retrieve}_{(p\,:\,r^*)} \;=\; p \mathbin{{+}{+}} [1]
\end{array}
$$

In summary, the incremental forward POSIX parsing algorithm based on bit-codes is as follows:

$$
\begin{array}{l}
\mathit{parseBC}'\; r\; \epsilon \\
\quad \mid \epsilon \;\in\; L(r) \;=\; \mathit{retrieve}_r \\
\mathit{parseBC}'\; r\; lw \;=\; \mathit{parseBC}'\; (\mathit{simpBC}\; (r\backslash_b l))\; w \\
\mathit{parseBC}\; r\; w \;=\; \mathit{decode}_r\; (\mathit{parseBC}'\; r\; w)
\end{array}
$$

We assume that in the call $\mathit{parseBC}'\; r\; w$, the bit-code annotations in expression r are empty, i.e. [].