Derivative-Based Diagnosis of Regular Expression Ambiguity

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

[1] Karlsruhe University of Applied Sciences
martin.sulzmann@hs-karlsruhe.de
[2] Nanyang Polytechnic
luzhuomi@gmail.com

Abstract. Regular expressions are often ambiguous. We present a novel method based on Brzozowski's derivatives to aid the user in diagnosing ambiguous regular expressions. We introduce a derivative-based finite state transducer to generate parse trees and minimal counter-examples. The transducer can be easily customized to either follow the POSIX or Greedy disambiguation policy and based on a finite set of examples it is possible to examine if there are any differences among POSIX and Greedy.

Keywords: regular expressions, derivatives, ambiguity, POSIX, Greedy

1 Introduction

A regular expression is *ambiguous* if a string can be matched in more than one way. For example, consider the expression $x^* + x$ where input string x can either be matched against x^* or x. Hence, this expression is ambiguous.

Earlier works There exist well-established algorithms to check for regular expression ambiguity. However, most works report ambiguity in terms of an automata which results from an ambiguity-preserving translation of the original expression, e.g. see the work by Book, Even, Greibach and Ott [3]. From a user perspective, it is much more useful to report ambiguity in terms of the original expression. We are only aware of two works which like us perform the ambiguity analysis on the original expression.

Brabrand and Thomsen [5] establish a structural relation to detect ambiguity based on which they can provide minimal counter examples. They consider some disambiguation strategies but do not cover the POSIX interpretation.

Borsotti, Breveglieri, Crespi-Reghizzi and Morzenti [4] show how to derive parse trees based on marked regular expressions [9] as employed in the Berry-Sethi algorithm [2]. They establish criteria to identify ambiguous regular expressions. Like ours, their approach can be customized to support either the POSIX [8] or Greedy [11] disambiguation policy. However, for POSIX/Greedy disambiguation, their approach requires tracking of dynamic data based on the Okui-Suzuki method [10]. Our approach solely relies on derivatives, no dynamic tracking of data is necessary.

Our work Brzozowski's derivatives [6] support the symbolic construction of automata where expressions represent automata states. This leads to elegant algorithms and based on (often simple) symbolic reasoning. In earlier work [13], we have studied POSIX matching based on derivatives. In this work, we show how to adapt and extend the methods developed in [13] to diagnose ambiguous expressions.

Contributions and outline In summary, our contributions are:

- We employ derivatives to compute all parse trees for a large class of (non-problematic) regular expressions (Section 3).
- We can build a finite state transducer to compute these parse trees (Section 4).
- We can easily customize the transducer to either compute the POSIX or greedy parse tree (Section 5).
- We can identify simple criteria to detect ambiguous expressions and to derive a finite set of minimal counter-examples. Thus, we can statically verify if there are any differences among POSIX and Greedy (Section 6).
- We have implemented the approach in Haskell. The implementation is available via `http://www.home.hs-karlsruhe.de/~suma0002/dad.html`.

In the upcoming section, we introduce our notion of regular expression, parse trees and ambiguity.

Proof sketches for results stated can be found in the optional appendix.

2 Regular Expressions, Parse Trees and Ambiguity

The development largely follows [7] and [5]. We assume that symbols are taken from a fixed, finite alphabet Σ. We generally write x, y, z for symbols.

Definition 1 (Words and Regular Expressions). *Words are either empty or concatenation of words and defined as follows:* $w ::= \epsilon \mid x \in \Sigma \mid w \cdot w$.

We denote regular expressions by r, s, t. *Their definition is as follows:* $r ::= x \in \Sigma \mid r^* \mid r \cdot r \mid r + r \mid \epsilon \mid \phi$ *The mapping to words is standard.* $\mathcal{L}(x) = \{x\}$. $\mathcal{L}(r^*) = \{w_1 \cdot ... \cdot w_n \mid n \geq 0 \wedge w_i \in \mathcal{L}(r) \wedge i \in \{1, .., n\}\}$. $\mathcal{L}(r \cdot s) = \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(r) \wedge w_2 \in \mathcal{L}(s)\}$. $\mathcal{L}(r + s) = \mathcal{L}(r) \cup \mathcal{L}(s)$. $\mathcal{L}(\epsilon) = \{\epsilon\}$. $\mathcal{L}(\phi) = \{\}$.

We say an expression r *is* nullable *iff* $\epsilon \in \mathcal{L}(r)$.

As it is common, we assume that $+$ and $\cdot$ are right-associative. That is, $x + y + x \cdot y \cdot z$ stands for $x + (y + (x \cdot (y \cdot z)))$.

A parse tree explains which subexpressions match which subwords. We follow [7] and view expressions as types and parse trees as values.

Definition 2 (Parse Trees). *Parse tree values are built using data constructors such as lists, pairs, left/right injection into a disjoint sum etc. In case of repetitive matches such as in case of Kleene star, we make use of lists. We use Haskell style notation and write* $[v_1, ..., v_n]$ *as a short-hand for* $v_1 : ... : v_n : []$.

$$v \ ::= \ () \mid x \mid (v, v) \mid L\ v \mid R\ v \mid vs \qquad vs \ ::= \ [] \mid v : vs$$

The valid relations among parse trees and regular expressions are defined via a natural deduction style proof system.

$$\vdash [] : r^* \qquad \frac{\vdash v : r \quad \vdash vs : r^*}{\vdash (v : vs) : r^*} \qquad \frac{\vdash v_1 : r_1 \quad \vdash v_2 : r_2}{\vdash (v_1, v_2) : r_1 \cdot r_2}$$

$$\frac{\vdash v_1 : r_1}{\vdash L\ v_1 : r_1 + r_2} \qquad \frac{\vdash v_2 : r_2}{\vdash R\ v_2 : r_1 + r_2} \qquad \vdash () : \epsilon \qquad \frac{x \in \Sigma}{\vdash x : x}$$

Definition 3 (Flattening). *We can flatten a parse tree to a word as follows:*

$$\begin{array}{llll} |()| = \epsilon & |x| = x & |L\ v| = |v| & |v : vs| = |v| \cdot |vs| \\ |[]| = \epsilon & |(v_1, v_2)| = |v_1| \cdot |v_2| & |R\ v| = |v| & \end{array}$$

Proposition 1 (Frisch/Cardelli [7]). *Let r be a regular expression. If $w \in \mathcal{L}(r)$ for some word w, then there exists a parse tree v such that $\vdash v : r$ and $|v| = w$. If $\vdash v : r$ for some parse tree v, then $|v| \in \mathcal{L}(r)$.*

Example 1. We find that $x \cdot y \in \mathcal{L}((x \cdot y + x + y)^*)$ where $[L\ (x, y)]$ is a possible parse tree. Recall that $+$ is right-associative and therefore we interpret $(x \cdot y + x + y)^*$ as $(x \cdot y + (x + y))^*$.

An expression is ambiguous if there exists a word which can be matched in more than one way. That is, there must be two distinct parse trees which share the same underlying word.

Definition 4 (Ambiguous Regular Expressions). *We say a regular expression r is* ambiguous *iff there exist two distinct parse trees v_1 and v_2 such that $\vdash v_1 : r$ and $\vdash v_2 : r$ where $|v_1| = |v_2|$.*

Example 2. $[L\ (x, y)]$ and $[R\ (L\ x), R\ (R\ y)]$ are two distinct parse trees for expression $(x \cdot y + x + y)^*$ and word $x \cdot y$.

Our ambiguity diagnosis methods will operate on arbitrary expressions. However, formal results are restricted to a certain class of 'non-problematic' expressions.

Definition 5 (Problematic Expressions). *We say an expression r is* problematic *iff it contains some sub-expression of the form s^* where $\epsilon \in \mathcal{L}(s)$.*

For problematic expressions, the set of parse trees is infinite, otherwise finite.

Example 3. Consider the problematic expression ϵ^* where for the empty input word we find the following (infinite) sequence of parse trees [], [()], [(), ()], ...

Proposition 2 (Frisch/Cardelli [7]). *For non-problematic expressions, the set of distinct parse trees which share the same underlying word is always finite.*

Next, we consider computational methods based on Brzozowski's derivatives to compute parse trees.

3 Computing Parse Trees via Derivatives

Derivatives denote left quotients and they can be computed via a simple syntactic transformation.

Definition 6 (Regular Expression Derivatives). *The* derivative *of expression r w.r.t. symbol x, written $d_x(r)$, is computed by induction on r:*

$$d_x(\phi) = \phi \quad d_x(\epsilon) = \phi \quad d_x(r_1 + r_2) = d_x(r_1) + d_x(r_2) \quad d_x(r^*) = d_x(r) \cdot r^*$$
$$d_x(y) = \begin{cases} \epsilon \text{ if } x = y \\ \phi \text{ otherwise} \end{cases} \quad d_x(r_1 \cdot r_2) = \begin{cases} d_x(r_1) \cdot r_2 + d_x(r_2) & \text{if } \epsilon \in \mathcal{L}(r_1) \\ d_x(r_1) \cdot r_2 & \text{otherwise} \end{cases}$$

The extension to words is as follows: $d_\epsilon(r) = r$. $d_{x \cdot w}(r) = d_w(d_x(r))$.

A descendant *of r is either r itself or the derivative of a descendant. We write $r \preceq s$ to denote that s is a descendant of r. We write $d(r)$ to denote the set of descendants of r.*

Proposition 3 (Brzozowski [6]). *For any expression r and symbol x we find that $\mathcal{L}(d_x(r)) = \{w \mid x \cdot w \in \mathcal{L}(r)\}$.*

Thus, we obtain a simple word matching algorithm by repeatedly building the derivative and then checking if the final derivative is nullable. That is, $w \in \mathcal{L}(r)$ iff $\epsilon \in \mathcal{L}(d_w(r))$. Nullability can easily be decided by induction on r. We omit the straightforward details.

Example 4. Consider expression $(x+y)^*$ and input $x \cdot y$. We find $d_x((x+y)^*) = (\epsilon + \phi) \cdot (x+y)^*$ and $d_y(d_x((x+y)^*)) = (\phi + \phi) \cdot (x+y)^* + (\phi + \epsilon) \cdot (x+y)^*$. The final expression is nullable. Hence, we can conclude that $x \cdot y \in \mathcal{L}((x+y)^*)$.

Based on the derivative method, it is surprisingly easy to compute parse trees for some input word w. The key insights are as follows:

1. Build all parse trees for the final (nullable) expression.
2. Transform a parse tree for $d_x(r)$ into a parse tree for r by injecting symbol x into $d_x(r)$'s parse tree. Injecting can be viewed as reversing the effect of the derivative operation.

Definition 7 (Empty Parse Trees). *Let r be an expression. Then, $allEps_r$ yields a set of parse trees. The definition of $allEps_r$ is by induction on r.*

$$allEps_\epsilon = \{ () \} \quad allEps_\phi = \{ \} \quad allEps_x = \{ \}$$
$$allEps_{r^*} = \{ [] \} \quad allEps_{r_1 \cdot r_2} = \{(v_1, v_2) \mid v_1 \in allEps_{r_1} \wedge v_2 \in allEps_{r_2} \}$$
$$allEps_{r_1 + r_2} = \{ L\ v_1 \mid v_1 \in allEps_{r_1} \} \cup \{ R\ v_2 \mid v_2 \in allEps_{r_2} \}$$

If the expression is not nullable it is easy to see that we obtain an empty set. For nullable expressions, $allEps_r$ yields empty parse trees.

Proposition 4 (Empty Parse Trees). *Let r be a nullable expression. Then, for any $v \in allEps_r$ we have that $\vdash v : r$ and $|v| = \epsilon$.*

Example 5. For the final (nullable) expression from Example 4 we find that $allEps_{(\phi+\phi)\cdot(x+y)^*+(\phi+\epsilon)\cdot(x+y)^*} = R\ (R\ (), [])$.

For nullable, non-problematic expressions r, we can state that $allEps_r$ yields all parse trees v for r where $|v| = \epsilon$.

Proposition 5 (All Empty Non-Problematic Parse Trees). *Let r be a non-problematic expression such that $\epsilon \in \mathcal{L}(r)$. Let v be a parse tree such that $\vdash v : r$ where $|v| = \epsilon$. Then, we find that $v \in allEps_r$.*

The non-problematic assumption is necessary. Recall Example 3.

What remains is to describe how to derive parse trees for the original expression. We achieve this by injecting symbol x into $d_x(r)$'s parse tree.

Definition 8 (Injecting Symbols into Parse Trees). *Let r be an expression and x be a symbol. Then, $injs_{d_x(r)}$ is a function which maps a $d_x(r)$'s parse tree to a set of parse trees of r. footnoteAdditional arguments are x and r but we choose the notation $injs_{d_x(r)}$ to highlight the definition is defined by pattern match over the various cases of the derivative operation. The definition is by induction on r.*

$$
\begin{array}{l}
injs_{d_x(\epsilon)}\ \_ = \{\ \} \quad injs_{d_x(\phi)}\ \_ = \{\ \} \quad injs_{d_x(x)}\ () = \{\ x\ \} \quad injs_{d_x(y)}\ \_ = \{\ \} \\
injs_{d_x(r^*)}\ (v, vs) = \{\ v' : vs \mid v' \in injs_{d_x(r)}\ v\ \} \\
injs_{d_x((r_1 \cdot r_2))} = \\
\quad \lambda v.\, \mathsf{case}\ v\ \mathsf{of} \\
\qquad (v_1, v_2) \to \{\ (v,\ v_2) \mid v \in injs_{d_x(r_1)}\ v_1\ \} \\
\qquad L\ (v_1, v_2) \to \{\ (v,\ v_2) \mid v \in injs_{d_x(r_1)}\ v_1\ \} \\
\qquad R\ v_2 \to \{\ (v, v') \mid v \in allEps_{r_1} \wedge v' \in injs_{d_x(r_2)}\ v_2\ \} \\
injs_{d_x((r_1 + r_2))} = \\
\quad \lambda v.\, \mathsf{case}\ v\ \mathsf{of} \\
\qquad L\ v_1 \to \{\ L\ v \mid v \in injs_{d_x(r_1)}\ v_1\ \} \\
\qquad R\ v_2 \to \{\ R\ v \mid v \in injs_{d_x(r_2)}\ v_2\ \}
\end{array}
$$

In the above, we use Haskell style syntax such as lambda-bound functions etc. The first couple of cases are straightforward. For brevity, we use the 'don't care' pattern _ and make use of a non-linear pattern in the third equation. In case of Kleene star, the parse tree is represented by a sequence. We call the injection function of the underlying expression on the first element. In case of concatenation $r_1 \cdot r_2$, we observe the shape of the parse tree of $d_x(r_1 \cdot r_2)$. For example, if we encounter $R\ v_2$, the left component r_1 must be nullable. Hence, we apply $allEps_{r_1}$.

Via a straightforward inductive proof on r, we can verify that the injection function yields valid parse trees.

Proposition 6 (Soundness of Injection). *Let r be an expression, x be a symbol and v be a parse tree such that $\vdash v : d_x(r)$. Then, for any $v' \in injs_{d_x(r)}$ we find that $\vdash v' : r$.*

Example 6. Consider our running example where $\vdash\ R\ (R\ (), []) : d_y(d_x((x + y)^*))$. Then, $inj s_{d_y(d_x((x+y)^*))}\ \ (R\ (R\ (), [])) = \{(L\ (), [y])\}$ where $\vdash\ (L\ (), [y]) : d_x((x + y)^*)$ and $d_x((x + y)^*) = (\epsilon + \phi) \cdot (x + y)^*$.

As in case of $allEps_r$, we can only guarantee completeness for non-problematic expressions.

Proposition 7 (Completeness of Non-Problematic Injection). *Let r be a non-problematic expression and v a parse tree such that $\vdash\ v : r$ where $|v| = x \cdot w$ for some letter x and word w. Then, there exists a parse tree v' such that (1) $\vdash\ v' : d_x(r)$ and (2) $v \in inj s_{d_x(r)}\ v'$.*

Definition 9 (Parse Tree Construction). *Let r be an expression. Then, the derivative-based procedure to compute all parse trees is as follows.*

$$allParse\ r\ \epsilon\ =\ allEps_r$$
$$allParse\ r\ x \cdot w\ =\ \{\ v\ \mid\ v\ \in\ inj s_{d_x(r)}\ v' \wedge v' \in (allParse\ d_x(r)\ w)\ \}$$

Proposition 8 (Valid Parse Trees). *Let r be an expression. Then, for each $v \in allParse\ r\ |v|$ we find that $\vdash\ v : r$.*

For non-problematic expressions, we obtain a complete parse tree construction method.

Proposition 9 (All Non-Problematic Parse Trees). *Let r be a non-problematic expression and v a parse tree such that $\vdash\ v : r$. Then, we find that $v \in allParse\ r\ |v|$.*

In case of a fixed expression r, calls to $allParse\ r$ repeatedly build the same set of derivatives. We can be more efficient by constructing a finite state transducer (FST) for a fixed expression r where states are descendants of r. The outputs are parse tree transformation functions. This is what we will discuss next.

4 Derivative-Based Finite State Transducer

The natural candidate for FST states are derivatives. That is, $\delta(r, x) = d_x(r)$. In general, descendants (derivatives) are not finite. Thankfully, Brzozowski showed that the set of dissimilar descendants is finite.

Definition 10 (Similarity). *We say two expressions r and s are* similar*, written $r \approx s$, if one can be transformed into the other by application of the following rules.*

$$\textit{(Idemp)}\ r + r \approx r \qquad \textit{(Comm)}\ r_1 + r_2 \approx r_2 + r_1$$
$$\textit{(Assoc)}\ (r_1 + r_2) + r_3 \approx r_1 + (r_2 + r_3) \qquad \textit{(Ctxt)}\ \frac{s \approx t}{R[s] \approx R[t]}$$

The (Ctxt) rules assumes expressions with a hole. We write $R[s]$ to denote the expression where the hole $[]$ is replaced by s.

$$\textit{(Hole Expressions)}\ R[] ::= [] \mid R[] \cdot s \mid s \cdot R[] \mid R[] + s \mid s + R[]$$

There is no hole inside Kleene star because the derivative operation will only ever be applied on unfoldings of the Kleene star but never within a Kleene star expression.

We write $d(r)/\approx$ to denote the set of equivalence classes of $d(r)$ w.r.t. the equivalence relation $\approx$.

Proposition 10 (Brzozowski [6]). *$d(r)/\approx$ is finite for any expression r.*

Based on the above, we build an automata where the set of states consists of a canonical representative for all descendants of some expression r. A similar approach is discussed in [14].

Definition 11 (Canonical Representative). *For each expression r we compute an expression $\mathcal{C}(r)$ by systematic application of the similarity rules: (1) Put alternatives in right-associative normal form via rule (Assoc). (2) Remove duplicates via rules (Idemp) where via rule (Comm) we push the right-most duplicates to the left. (3) Repeat until there are no further changes.*

Proposition 11 (Canonical Normal From). *Let r be an expression. Then, $\mathcal{C}(r)$ represents a canonical normal form of r.*

Furthermore, alternatives keep their relative position. For example, $\mathcal{C}(r + s + s_1 + ... + s_n + s + t) = r + s + s_1 + ... + s_n + t$. This is important for the upcoming construction of POSIX and Greedy parse trees.

Proposition 12 (Finite Dissimilar Canonical Descendants). *Let r be an expression. Then, the set $\mathcal{D}(r) = \{\mathcal{C}(s) \mid r \preceq s\}$ is finite.*

Like in case of $injs$, we need to maintain information how to transform parse trees among similar expressions. Hence, we attach parse tree transformation functions to the similarity rules.

Definition 12 (Similarity with Parse Tree Transformation). *We write $r \overset{f}{\gg} s$ to denote that expressions r and s are similar and a parse tree of s can be transformed into a parse tree of r via function f. In case the function returns a set of parse trees we write $r \overset{fs}{\gg} s$. We write $r \gg s$ if the parse tree transformation*

is not of interest.

$$\textit{(Idemp)}\ \frac{fs(u) = \{L\ u, R\ u\}}{r + r \overset{fs}{\gg} r} \qquad \textit{(Comm)}\ \frac{\begin{array}{c} f(L\ u) = R\ u \\ f(R\ u) = L\ u \end{array}}{r_1 + r_2 \overset{f}{\gg} r_2 + r_1}$$

$$\textit{(Assoc)}\ \frac{\begin{array}{ll} f(L\ u_1) & = L\ (L\ u_1) \\ f(R\ (L\ u_2)) & = L\ (R\ u_2) \\ f(R\ (R\ u_3)) & = R\ u_3 \end{array}}{(r_1 + r_2) + r_3 \overset{f}{\gg} r_1 + (r_2 + r_3)} \qquad \textit{(Lift)}\ \frac{\begin{array}{c} r \overset{f}{\gg} s \\ fs(u) = \{f(u)\} \end{array}}{r \overset{fs}{\gg} s}$$

$$\textit{(C1)}\ \frac{\begin{array}{c} s \overset{fs}{\gg} t \\ gs(u_r, u_t) = \{(u_r, u_s) \mid u_s \in fs(u_t)\} \end{array}}{r \cdot s \overset{gs}{\gg} r \cdot t} \qquad \textit{(C2)}\ \frac{\begin{array}{c} s \overset{fs}{\gg} t \\ gs(u_t, u_r) = \{(u_s, u_r) \mid u_s \in fs(u_t)\} \end{array}}{s \cdot r \overset{gs}{\gg} t \cdot r}$$

$$\textit{(C3)}\ \frac{\begin{array}{c} s \overset{fs}{\gg} t \\ gs(L\ u_r) = \{L\ u_r\} \\ gs(R\ u_t) = \{R\ u_s \mid u_s \in fs(u_t)\} \end{array}}{r + s \overset{gs}{\gg} r + t} \qquad \textit{(C4)}\ \frac{\begin{array}{c} s \overset{fs}{\gg} t \\ gs(L\ u_t) = \{L\ u_s \mid u_s \in fs(u_t)\} \\ gs(R\ u_r) = \{R\ u_r\} \end{array}}{s + r \overset{gs}{\gg} t + r}$$

The above rules are derived from the ones in Definition 10 by providing the appropriate parse tree transformations. Due to the similarity rule (Idemp) we may obtain a set of parse trees. Rules (C1-4) cover all the cases described by rule (Ctxt). The attached (transformation) functions yield valid parse trees (soundness) and every parse tree of a similar expression can be obtained (completeness).

Proposition 13 (Soundness of Transformation). *Let r and s be two expressions and fs a function such that $r \overset{fs}{\gg} s$. Then, we find that (1) $r \approx s$ and (2) for any parse tree v where $\vdash v : s$ we have that $\vdash v' : r$ for any $v' \in fs(v)$.*

Proposition 14 (Completeness of Transformation). *Let r and s be two expressions and v be a parse tree such that $\vdash v : r$ and $r \approx s$. Then, $r \overset{fs}{\gg} s$ where $v \in fs(v')$ for some v' such that $\vdash v' : s$.*

The FST to compute parse trees for some expression r consists of states $\mathcal{D}(r)$. Each state transition from s to $\mathcal{C}(d_x(s))$ yields as output a parse tree transformer function which is a composition of $injs_{d_x(s)}$ and fs where $d_x(s) \overset{fs}{\gg} \mathcal{C}(d_x(s))$.

Definition 13 (FST Construction). *Let r be an expression. We define $\mathcal{FST}(r) = (Q, \Sigma, \delta, q_0, F)$ where $Q = \mathcal{D}(r)$, $q_0 = r$, $F = \{s \in Q \mid \epsilon \in \mathcal{L}(s)\}$ and for each $s \in Q$ and $x \in \Sigma$ we set $\delta(s, x) = (\mathcal{C}(d_x(s)), gs)$ where $d_x(s) \overset{fs}{\gg} \mathcal{C}(d_x(s))$ and $gs(u) = \{u_2 \mid u_1 \in fs(u) \wedge u_2 \in injs_{d_x(s)}\ u_1\}$.*

The transition relation δ is inductively extended to words as follows. We define $\delta(s, \epsilon) = (s, \lambda u.\{u\})$ and $\delta(s, x \cdot w) = (r, fs)$ where $\delta(s, x) = (t, gs)$ and $\delta(t, w) = (r, hs)$ where $fs(u) = \{u_2 \mid u_1 \in hs(u) \wedge u_2 \in gs(u_1)\}$.

Proposition 15 (All Non-Problematic Parse Trees via FST). *Let r be a non-problematic expression and v a parse tree such that $\vdash v : r$. Let $\mathcal{FST}(r) = (Q, \Sigma, \delta, q_0, F)$. Then, we find that $v \in fs(allEps_{r'})$ where $\delta(r, |v|) = (r', fs)$.*

5 Computing POSIX and Greedy Parse Trees

Based on our earlier work [13] we can immediately conclude that the 'first' (left-most) match obtained by executing $\mathcal{FST}(r)$ is the POSIX match. [3] The use of derivatives guarantees that the longest left-most (POSIX) parse tree is computed.

Proposition 16 (POSIX). *Let r be an expression and w be a word such that $w \in \mathcal{L}(r)$. Let $\mathcal{FST}(r) = (Q, \Sigma, \delta, q_0, F)$. Let $\delta(r, w) = (r', fs)$ for some expression r' and transformer fs. Then, $fs(allEps_{r'}) = \{v_1, ..., v_n\}$ for some parse trees v_i where v_1 is the POSIX match.*

With little effort it is possible to customize our FST construction to compute Greedy parse trees. The insight is to normalize derivatives such that they effectively correspond to partial derivatives. Via this normalization step, we obtain as the 'first' result the Greedy (left-most) parse tree. This follows from our earlier work [12] where we showed that partial derivatives naturally yield greedy matches.

We first define partial derivatives which are a non-deterministic generalization of derivatives. Instead of a single expression, the partial derivative operation yields a set of expressions.

Definition 14 (Partial Derivatives). [4] *Let r be an expression and x be a symbol. Then, the* partial derivative *of r w.r.t. x is computed as follows:*

$$pd_x(\phi) = \{\} \quad pd_x(\epsilon) = \{\} \quad pd_x(y) = \begin{cases} \{\epsilon\} & \text{if } x = y \\ \{\} & \text{otherwise} \end{cases}$$

$$pd_x(r_1 + r_2) = pd_x(r_1) \cup pd_x(r_2) \qquad pd_x(r^*) = \{r' \cdot r^* \mid r' \in pd_x(r)\}$$

$$pd_x(r_1 \cdot r_2) = \begin{cases} \{r_1' \cdot r_2 \mid r_1' \in pd_x(r_1)\} \cup pd_x(r_2) & \text{if } \epsilon \in \mathcal{L}(r_1) \\ \{r_1' \cdot r_2 \mid r_1' \in pd_x(r_1)\} & \text{otherwise} \end{cases}$$

Let $M = \{r_1, ..., r_n\}$ be a set of expressions. Then, we define $+M = r_1 + ... + r_n$ and $+\{\} = \phi$.

To derive partial derivatives via derivatives, we impose the following additional similarity rules.

Definition 15 (Partial Derivative Similarity Rules).

$$\textit{(Dist)}\ \frac{f(L\ (u_r, u_t)) = (L\ u_r, u_t) \quad f(R\ (u_s, t_t)) = (R\ u_s, u_t)}{(r + s) \cdot t \overset{f}{\gg} r \cdot t + s \cdot t} \qquad \textit{(ElimPhi)}\ \phi \cdot r \overset{\perp}{\gg} \phi$$

[3] Technically, we treat the set of parse trees like a list. Recall that *allEps* and the simplification rule (Idemp) favor the left-most match. Alternatives keep their relative position in an expression.

[4] We omit the smart constructor found in [1] as this is not relevant here.

Rule (Dist) mimics the set-based operations performed by $pd_{\cdot}(\cdot)$ in case of concatenation and Kleene star. Rule (ElimPhi) covers cases where the set is empty. We use $\perp$ to denote the undefined parse tree transformer function. As there is no parse tree for ϕ this function will never be called.

Proposition 17 (Partial Derivatives as Normalized Derivatives). *Let r be an expression and x be a symbol. Then, we have that $+pd_x(r)$ is syntactically equal to some expression s such that $d_x(r) \gg s$. We ignore the transformer function which is not relevant here.*

Based on the above and our earlier results in [12] we can immediately conclude the following.

Proposition 18 (Greedy). *Let r be an expression and w be a word such that $w \in \mathcal{L}(r)$. Let $\mathcal{FST}(r) = (Q, \Sigma, \delta, q_0, F)$ where we additionally apply the similarity rules in Definition 15 such that canonical representatives satisfy the property stated in Proposition 17. Let $\delta(r, w) = (r', fs)$ for some expression r' and transformer fs. Then, $fs(allEps_{r'}) = \{v_1, ..., v_n\}$ for some parse trees v_i where v_1 is the Greedy parse tree.*

Remark 1 (Linear-Time Complexity). Our approach has linear-time complexity [5] in the size of the input word w assuming we treat the size of the regular expression r as a constant and consider computation of the 'first' parse tree only. The size of dissimilar derivatives is at most exponential in the size of r. The size of a parse tree is bound by r. Time complexity of parse tree transformation functions is linear in the size of the input.

There is quite a bit of scope to improve the performance by for example employing more efficient representations of our parse tree transformation functions. For efficiency reasons, we also may want to specialize $\mathcal{FST}(r)$ to compute the POSIX and Greedy parse trees only. Currently, we rely on Haskell's lazy evaluation strategy to do only the necessary work when extracting the first parse tree from the final result obtained by running $\mathcal{FST}(r)$. These are topics to consider in future work.

6 Ambiguity Diagnosis

We can identify three situations where ambiguity of r arises during the construction of $\mathcal{FST}(r)$. The first situation concerns nullable expressions. If we encounter multiple empty parse trees for a nullable descendant (accepting state) then we end up with multiple parse trees for the initial state. Then, the initial expression is ambiguous.

The second situation concerns the case of injecting a symbol into the parse tree of a descendant. Recall that the $inj s$ function from Definition 8 possibly yields a set of parse trees. This will only happen if we apply the derivative operation on some subterm $t_1 \cdot t_2$ where t_1 is a nullable expression with multiple empty parse trees.

[5] We do not measure the complexity of constructing $\mathcal{FST}(r)$ which can be exponential in the size of r.

The final (third) situation ambiguous situation arises in case we build canonical representatives. Recall Definition 12. We end up with multiple parse trees whenever we apply rule (Idemp).

These are the only situations which may give rise to multiple parse trees. That is, if none of these situations arises the expression must be unambiguous. We summarize these observations in the following result.

Definition 16 (Realizable State). *We say that $s \in \mathcal{D}(r)$ is* realizable*, if there exists a path in $\mathcal{FST}(r)$ such that (1) we reach s and (2) along this path all states (expressions) including s do* not *describe the empty language.*

Proposition 19 (Ambiguity Criteria). *Let r be a non-problematic expression. Then, r is ambiguous iff there exists a realizable $s \in \mathcal{D}(r)$ and some symbol x where one of the following conditions applies:*

A1 $|allEps_s| > 1$, *or*
A2 $s = R[t_1 \cdot t_2]$ *where* $|allEps_{t_1}| > 1$, *or*
A3 $\mathcal{L}(\mathcal{C}(d_x(s))) \neq \{\}$ *and* $d_x(s) \overset{fs}{\gg} \mathcal{C}(d_x(s))$ *with rule (Idemp) applied.*

The above criteria are easy to verify. In terms of the FST generated, criteria **A1** is always connected to a final state whereas criteria **A2** and **A3** are always connected to transitions. Our implementation automatically generates the FST annotated with ambiguity information.

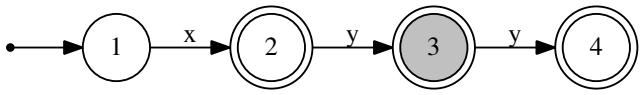

Fig. 1. $\mathcal{FST}((x + x \cdot y) \cdot (y + \epsilon))$

In Figure 1, we consider the FST for $(x + x \cdot y) \cdot (y + \epsilon)$. Final state is highlighted grey to indicate that ambiguity due to **A1** arises. Indeed, for input $x \cdot y$ we can observe that there are two distinct parse trees. Namely, $(L\ x, L\ x)$ and $(R\ (x, y), R\ ())$. Hence, the expression is ambiguous.

Consider another example taken from [4]. See Figure 2. We find ambiguous transitions due to **A2** and **A3**. Such transitions are represented as dotted arrows with labels to indicate **A2** and **A3**. Ambiguity due to **A1** does not arise for this example.

Let us investigate the ambiguous transition from state 2 to state 3. We carry out the constructions of states starting with the initial expression $r \cdot y$ where $r = (x \cdot x^* + y \cdot x + x \cdot y \cdot x)^*$. For brevity, we make use of additional similarity rules such as $\epsilon \cdot s \approx s$ to keep the size of descendants manageable. In the following, we write $r \xrightarrow{x} s$ if $s = d_x(r)$.

$$
\begin{aligned}
& r \cdot y \\
\xrightarrow{x}\ & ((x^* + y \cdot x) \cdot r) \cdot y \\
\xrightarrow{y}\ & (x \cdot r) \cdot y + (x \cdot r) \cdot y + \epsilon \\
\approx\ & (x \cdot r) \cdot y + \epsilon
\end{aligned}
$$

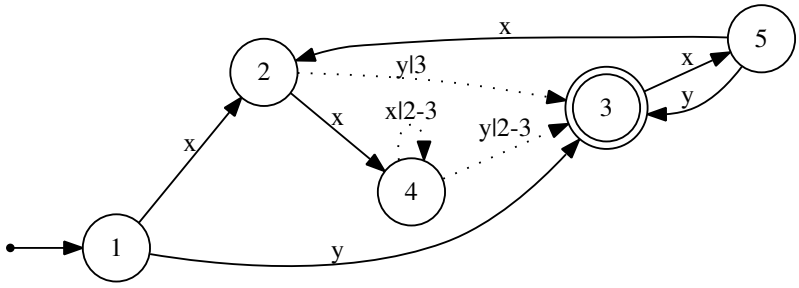

Fig. 2. $\mathcal{FST}((x \cdot x^* + y \cdot x + x \cdot y \cdot x)^* \cdot y)$

In the last step, we apply rule (Idemp). Hence, the ambiguous transition from state 2 to state 3.

State 3 is final, however, $x \cdot y$ is not yet a full counter-example to exhibit ambiguity. In essence, $x \cdot y$ is a prefix of the full counter-example $x \cdot y \cdot x \cdot y$. For this example, we obtain parse trees $([L\ (x, []), L\ (R\ (x, y))], y)$ and $(R\ (R\ (x, (y, x))), y)$. The first one is obtained via Greedy and the second one via POSIX.

To summarize, from the FST it is straightforward to derive minimal prefixes of counter-examples. To obtain actual counter-examples, minimal prefixes need to be extended so that a final state is reached. Based on the FST, we could perform a breadth-first search to calculate all such minimal counter-examples. Alternatively, we can built (minimal) counter-examples during the construction of the FST.

There is clearly much scope for more sophisticated ambiguity diagnosis based on the information provided by the FST. An immediate application is to check (statically) any differences among Greedy and POSIX. We simply check both methods against the set of minimal counter-examples. It is clear that there are only finitely many (minimal) counter-examples as there are a finite number of states and transitions. Obtaining more precise bounds on their size is something to consider in future work.

References

1. Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
2. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, December 1986.
3. R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Trans. Comput.*, 20(2):149–153, February 1971.
4. Angelo Borsotti, Luca Breveglieri, Stefano Crespi-Reghizzi, and Angelo Morzenti. From ambiguous regular expressions to deterministic parsing automata. In *Proc. of CIAA'15*, volume 9223 of *LNCS*, pages 35–48. Springer, 2015.
5. Claus Brabrand and Jakob G. Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. In *Proc. of PPDP'10*, pages 243–254. ACM, 2010.
6. Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

7. Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Proc. of ICALP'04*, pages 618– 629. Spinger-Verlag, 2004.
8. 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).
9. R. McNaughton and H. Yamada. Regular expressions and finite state graphs for automata. *IRE Trans. on Electronic Comput*, EC-9(1):38–47, 1960.
10. 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.
11. PCRE - *Perl Compatible Regular Expressions*. http://www.pcre.org/.
12. Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives. In *Proc. of PPDP'12*, pages 79–90. ACM, 2012.
13. Martin Sulzmann and Kenny Zhuo Ming Lu. POSIX regular expression parsing with derivatives. In *Proc. of FLOPS'14*, volume 8475 of *LNCS*, pages 203–220. Springer, 2014.
14. Bruce W. Watson. A taxonomy of finite automata minimization algorithmes. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.

A Proof Sketches

A.1 Propositions 4 and 5

For both statements, we proceed by induction on r. Consider Proposition 5. For case r^* due to the 'non-problematic' assumption we find that $\vdash [] : r^*$ is the only possible candidate.

A.2 Propositions 6 and 7

Again by induction on r. For Proposition 7, we consider some of the interesting cases.

- Case r^* where $\vdash v : r^*$:
  1. By assumption r^* is non-problematic. Hence, $v = v_1 : vs$ for some v_1 and vs where $\vdash v_1 : r$ and $|v_1| = x \cdot w$ for some word w.
  2. By induction there exists v' such that $\vdash v' : d_x(r)$ and $v_1 \in injs_{d_x(r)}\ v'$.
  3. By definition of $injs$ we find that $v_1 : vs \in injs_{d_x(r^*)}\ v' : vs$.
  4. Hence, there exists $v' : vs$ which guarantees (1) and (2) and we are done.
- Case $r_1 \cdot r_2$ where $\vdash v : r_1 \cdot r_2$:
  1. By assumption $v = (v_1, v_2)$ where $\vdash v_1 : r_1$, $\vdash v_2 : r_2$ and $|v_1| = x \cdot w$ for some word w.
  2. Suppose $\epsilon \notin \mathcal{L}(r_1)$.
     (a) By induction there exists v' such that $\vdash v' : d_x(r_1)$ and $v_1 \in injs_{d_x(r_1)}\ v'$.
     (b) Under our assumptions $d_x(r_1 \cdot r_2) = d_x(r_1) \cdot r_2$.
     (c) By definition of $injs$ we find that $(v_1, v_2) \in injs_{d_x(r_1 \cdot r_2)}\ (v', v_2)$.
     (d) Hence, there exists (v', v_2) which guarantees (1) and (2) and we are done.
  3. Otherwise $\epsilon \in \mathcal{L}(r_1)$ where we assume that $v_1 = x \cdot w$ for some word w:
     (a) Similar to the above reasoning we find $(v_1, v_2) \in injs_{d_x(r_1) \cdot r_2}\ (v', v_2)$ for some v'.
     (b) By assumption $d_x(r_1 \cdot r_2) = d_x(r_1) \cdot r_2 + d_x(r_2)$.
     (c) Hence, $(v_1, v_2) \in injs_{d_x(r_1 \cdot r_2)}\ L\ (v', v_2)$ via which we can establish (1) and (2) and we are done.
  4. The only remaining case is that $\epsilon \in \mathcal{L}(r_1)$ and $|v_1| = \epsilon$.
     (a) Hence, $|v_2| = x \cdot w$ for some word w.
     (b) By induction on r_2 we find that there exists v' such that $\vdash v' : d_x(r_2)$ and $v_2 \in injs_{d_x(2_1)}\ v'$.
     (c) By Proposition 5 we obtain that $v_1 \in allEps_{r_1}$.
     (d) By definition of $injs$ we find that $(v_1, v_2) \in injs_{d_x(r_1 \cdot r_2)}\ R\ v_2$ which concludes the proof for the subcase of concatenated expressions.

A.3 Propositions 8 and 9

Proposition 8 follows straightforwardly from Propositions 4 and 6.

Consider Proposition 9 where we need to verify that for all non-problematic r and v where $\vdash v : r$ we find that $v \in allParse\ r\ |v|$. We proceed by induction on $|v|$.

Case $|v| = \epsilon$: By Proposition 5 we find that $v \in allEps_r$. By definition of $allParse$ we conclude that $v \in allParse\ r\ |v|$ and we are done.

Case $|v| = x \cdot w$: By Proposition 7, there exists v' such that $\vdash v' : d_x(r)$, $v \in inj s_{d_x(r)}\ v'$ and $|v'| = w$. Via a simple induction we can verify that if r is non-problematic so must be $d_x(r)$. By I.H. (for $d_x(r)$ and v') we find that $v' \in allParse\ d_x(r)\ w$. By definition of $allParse$ we conclude that $v \in allParse\ r\ |v|$ and we are done.

A.4 Propositions 11 and 12

Consider Proposition 11. We can show that the thus systematically applied similarity rules represent a terminating and confluent rewrite system. Hence, we obtain canonical normal forms.

Consider Proposition12. The set of canonical representatives is finite. Follows from Brzozowski's result Proposition 10.

A.5 Propositions 13 and 14

Both results follow by induction. For Proposition 13, by induction on the derivation $r \overset{fs}{\gg} s$. For Proposition 14, by induction $r \approx s$.

A.6 Proposition 15

The set of states and transitions is finite. Follows from Proposition 12. The resulting parse trees must valid as this follows from the respective results for $inj s$ and parse tree transformations resulting from similarity. The same applies for the completeness direction where we require the assumption that the expression is non-problematic.

A.7 Proposition 16

Follows from our earlier results stated in [13]. Note that we strictly favor left-most parse trees. Recall the definitions

$$allEps_{r_1+r_2} = \{L\ v_1 \mid v_1 \in allEps_{r_1}\} \cup \{R\ v_2 \mid v_2 \in allEps_{r_2}\}$$

and

$$\text{(Idemp)}\ \frac{fs(u) = \{L\ u, R\ u\}}{r + r \overset{fs}{\gg} r}$$

A.8 Proposition 17

By induction on r. We only consider the case of $r_1 \cdot r_2$ where $\epsilon \notin \mathcal{L}(r_1)$. We assume that $=$ denotes for syntactic equality.

By induction, $+pd_x(r_1) = s_1$ for some s_1 where $d_x(r_1) \gg s_1$ (1). W.l.o.g., $pd_x(r_1) = s_{1_1} + ... + s_{1_m}$ for some s_{1_j}.

Hence, $+pd_x(r_1 \cdot r_2) = +\{s_{1_1} \cdot r_2, ..., s_{1_m} \cdot r_2\} = s_{1_1} \cdot r_2 + ... + s_{1_m} \cdot r_2$.

By definition, $d_x(r_1 \cdot r_2) = d_x(r_1) \cdot r_2$. By similarity and the above (1), $d_x(r_1 \cdot r_2) \gg s_{1_1} \cdot r_2 + ... + s_{1_m} \cdot r_2$.

Thus, for some s we have that $+pd_x(r_1 \cdot r_2) = s$ and $d_x(r_1 \cdot r_2) \gg s$. Take s equal to $s_{1_1} \cdot r_2 + ... + s_{1_m} \cdot r_2$.

A.9 Proposition 18

Proposition 17 is crucial here. Via the additional similarity rules from Definition 15 we can normalize derivatives such that they effectively correspond to partial derivatives. The result follows then straightforwardly from our earlier results stated in [12].

A.10 Proposition 19

If any of the criteria **A1-3** arise, we can construct a counter-example. See discussion in Section 6. Hence, the expression must be ambiguous.

If none of the criteria arises, the expression must be unambiguous. There is no ambiguity in the parse tree construction. Our completeness results guarantee that all possible parse trees are covered.