Fixing Regular Expression Matching Failure

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 a popular formalism to specify patterns of input. There exist effective algorithms to check if a regular expression matches some input. Little is known how to fix expressions in case of matching failure. We introduce a novel method to fix regular expressions based on Antimirov's partial derivatives. Our method guarantees that fixes are small and the approach can be implemented with little effort.

1 Introduction

Regular expressions are widely used in a multitude of applications, ranging from web search engines, description of semi-structured to their classical use in lexical analysis. Their theory is well-understood and there exist established algorithms to check if a regular expression matches a word. For example, the standard approach is to convert the expression into an FSA and then run the FSA against the input string. A less well-studied problem is how to deal with matching failure.

Our interest is to fix regular expressions in case an expression r does not match some input word w. Concretely, in case $w \notin \mathcal{L}(r)$ we wish to find to some expression s such that $w \in \mathcal{L}(s)$ and $\mathcal{L}(r) \subseteq \mathcal{L}(s)$. The fix s accepts the word and subsumes the original expression r. The problem could be trivially solved by providing the fix $r + w$ where $+$ represents the alternative operator. This approach is of course far too naive. If possible we wish to apply small fixes to r.

In terms of the FSA implied by the regular expression, we wish to add as few states and transitions as possible such that the fixed FSA accepts the word. One issue is that the changes made to the FSA must be transfered back to the expression from which the FSA is derived from. This appears to be non-trivial assuming standard automata-based matching methods.

In this work, we introduce an entirely symbolic approach to fix regular expressions. For regular expression matching we use Antimirov's partial derivatives [1]. Partial derivatives represent states of an NFA. Importantly, partial derivatives are subterms of the original expression. In case of matching failure where we encounter a failure state, failure can be traced back to faulty subterms in the original expression. Faulty subterms are either symbol or ϵ expressions which clash with an input symbol. Roughly, the fix is to provide the input symbol as an alternative. To the best of our knowledge, we are the first to study the problem of fixing matching failure. We offer a solution by providing small fixes.

Specifically, our contributions are:

- We revisit Antimirov's partial derivatives (Section 3). In our formulation, we attach labels to certain subterms which allows us to establish precise connections between subterms in partial derivatives and the original expression (Section 3.3).
- Based on this information, we introduce a symbolic method to fix expressions in case of regular expression matching failure (Section 4).

Related work is discussed in Section 5. We conclude in Section 6. The upcoming section highlights the key ideas of our approach. The (optional) Appendix contains proofs which did not fit within the 15 page space restriction of the main paper.

2 Key Ideas

The basic idea behind Antimirov's matching method is to repeatedly build partial derivatives for symbols in the input word. Once all symbols have been consumed we need to check if any nullable partial derivatives are reached. If yes the expression matches the input word.

For example, consider expression $(x \cdot y + x)^*$ and input word $x \cdot y$. Operator $\cdot$ denotes concatenation and $+$ denotes choice. Whenever possible we drop parentheses by assuming that $\cdot$ binds tighter than $+$.

Similar to Brzozowski's derivatives [4], partial derivatives are computed by traversal over the regular expression by taking away some leading symbol. The difference is that derivatives correspond to DFA states whereas partial derivatives correspond to NFA states. Hence, the function $\partial_x(r)$ to build partial derivatives for some expression r w.r.t. symbol x yields a set.

For our example, we find the following computation steps.

1. Consumption of first input symbol x:

$$\partial_x((x \cdot y + x)^*) = \{y \cdot (x \cdot y + x)^*, (x \cdot y + x)^*\}$$

2. Consumption of second input symbol y:

$$\begin{array}{r}\partial_y(y \cdot (x \cdot y + x)^*) = \{(x \cdot y + x)^*\} \\ \partial_y((x \cdot y + x)^*) = \{\}\end{array}$$

The empty set $\{\}$ indicates a failure state. However, we also reach $(x \cdot y + x)^*$ which is nullable. Hence, we can establish that $(x \cdot y + x)^*$ matches input $x \cdot y$.

Let's consider an example where matching fails. Suppose we try to match $(x \cdot y + x)^*$ against $x \cdot z$. The second step yields

$$\begin{array}{r}\partial_z(y \cdot (x \cdot y + x)^*) = \{\} \\ \partial_z((x \cdot y + x)^*) = \{\}\end{array}$$

which indicates matching failure.

To provides fixes in case of failure, we adopt Antimirov's method as follows. We introduce the additional expression $(z \mid \not{x})$ to denote that we expect z but find x. Thus, we can keep track of matching failure when building partial derivatives. By nature of the partial derivative operation, such 'mismatch' expressions always occur in the left-most position and only involve simple terms (either symbols or ϵ). For our example, we find

$$\begin{aligned} \partial_z(y \cdot (x \cdot y + x)^*) &= \{(z \mid \not{y}) \cdot (x \cdot y + x)^*\} \\ \partial_z((x \cdot y + x)^*) &= \{(z \mid \not{x}) \cdot y \cdot (x \cdot y + x)^*, (z \mid \not{x}) \cdot (x \cdot y + x)^*\} \end{aligned}$$

As we can see, all partial derivatives are faulty. How to fix them?

Now comes the crucial observation. The subterms which are in conflict with the expected symbol are subterms in the original expression. This is due to the fact that subterms in partial derivatives are subterms in the original expression. To make this point more clear we attach labels to subterms.

The labeled (original) expression is $(x_1 \cdot y_2 + x_3)^*$. The computation steps including labels are

$$\begin{aligned} \partial_z(y_2 \cdot (x_1 \cdot y_2 + x_3)^*) &= \{(z \mid \not{y})_2 \cdot (x_1 \cdot y_2 + x_3)^*\} \\ \partial_z((x \cdot y + x)^*) &= \{(z \mid \not{x})_1 \cdot y_2 \cdot (x_1 \cdot y_2 + x_3)^*, (z \mid \not{x})_3 \cdot (x_1 \cdot y_2 + x_3)^*\} \end{aligned}$$

Fixing a faulty partial derivative is surprisingly simple. A mismatch such as $(z \mid \not{x})_3$ is resolved by providing the alternative z at label position 3 in the original expression. Recall that the fixed expression shall subsume the original expression. Hence, from $(z \mid \not{x})_3$ we derive the fix $(x \cdot y + \underline{x + z})^*$ where we have underlined the replaced part. Similarly, from $(z \mid \not{x})_1$ and $(z \mid \not{y})_2$ we derive the fixes $(\underline{(x + z)} \cdot y + x)^*$ and $(x \cdot \underline{(y + z)} + x)^*$. Fixing a faulty partial derivative does not immediately imply that the expression is fixed.

For example, consider the (intermediate) fix $((x + z) \cdot y + x)^*$. Restarting the matching process for this expression yields

$$\partial_{x \cdot z}(((x + z) \cdot y + x)^*) = \{y \cdot ((x + z) \cdot y + x)^*\}$$

where we have repeatedly applied the (standard) partial derivative operation first for x and then for z. The faulty partial derivative $(z \mid \not{x})_1 \cdot y_2 \cdot (x_1 \cdot y_2 + x_3)^*$ from above has been fixed. However, there is still matching failure because $y \cdot ((x + z) \cdot y + x)^*$ is not nullable. Resolution of non-nullable partial derivatives is similar to the resolution of mismatches. We need to compute the subterms (in the original expression) such that $y \cdot ((x + z) \cdot y + x)^*$ becomes nullable. A possible fix here is $((x + z) \cdot \underline{(y + \epsilon)} + x)^*$.

We conclude that for expression $(x \cdot y + x)^*$ and input $x \cdot z$ our method yields the following fixes:

1. $(x \cdot y + x + z)^*$,
2. $(x \cdot (y + z) + x)^*$ and
3. $((x + z) \cdot (y + \epsilon) + x)^*$.

In general, fixes are small because the resolution of mismatches always involves an input symbol and a simple subterm. For our example, we can argue that (1) and (2) are minimal as they are obtained via a single resolution step. A more detailed discussion regarding minimal/best fixes follows later.

To summarize. We enrich partial derivatives and the operation to compute them to keep track of mismatches. A mismatch represents a conflict between an input symbol and a simple subterm which arises in the original expression. Repeated resolution of mismatches and non-nullable partial derivatives eventually yields an expression which accepts the input. In the remainder, we introduce the technical machinery to formalize our approach.

3 Partial Derivatives with Labels

We assume a fixed alphabet Σ consisting of a finite set of symbols. We commonly refer to symbols as $x, y, \ldots$. The set Σ^* denotes the set of finite words over Σ. We write ϵ to denote the empty word and $u \cdot v$ to denote the concatenation of two words $u, v \in \Sigma^*$.

We attach labels to subterms where labels are taken from a denumerable set. In examples, we will use the set $\mathbb{N}$ of natural numbers. Via labels we can connect partial derivative subterms to their originating expression. We will only care about subterms consisting of a symbol x or ϵ expression. Hence, only those subterms are labeled.

Definition 1 (Labeled Regular Expressions). *Labeled regular expressions over Σ are defined by the following grammar:*

$$r, s, t ::= \epsilon_l \mid x_l \in \Sigma \mid r + r \mid r \cdot r \mid r^*$$

where l refers to a label.

We write R to denote sets of expressions.

$$R ::= \{\} \mid \{r\} \cup R$$

The meaning of labeled regular expressions is defined by function $\mathcal{L}()$ which maps expressions to sets of words.

$$\mathcal{L}(\epsilon_l) = \{\epsilon\}$$
$$\mathcal{L}(x_l) = \{x\}$$
$$\mathcal{L}(r + s) = \mathcal{L}(r) \cup \mathcal{L}(s)$$

$$\mathcal{L}(r \cdot s) = \{u \cdot v \mid u \in \mathcal{L}(r) \wedge v \in \mathcal{L}(s)\}$$
$$\mathcal{L}(r^*) = \{\epsilon\} \cup \{u_1 \cdot \ldots \cdot u_n \mid u_i \in \mathcal{L}(r) \wedge i \in \{1,..,n\} \wedge n \in \mathbb{N}\}$$

The definition of $\mathcal{L}()$ extends to R as follows:

$$\mathcal{L}(\{\}) = \{\}$$
$$\mathcal{L}(\{r\} \cup R) = \mathcal{L}(r) \cup \mathcal{L}(R)$$

To omit parentheses, we assume that $\cdot$ has a higher precedence than $+$. For example, $r + s \cdot t$ is the short form of $r + (s \cdot t)$.

An expression is properly labeled if each subterm ϵ and x is attached with a distinct label. For example, $x+\epsilon_1 \cdot y_1$ is not properly labeled because x is missing a label and the label 1 appears twice. As we will see, there may be duplicate label occurrences in partial derivatives resulting from some properly labeled regular expression.

To define partial derivatives, we first introduce a predicate $\nu(\cdot)$ to detect nullable expressions.

Definition 2 (Nullable). *A expression r is* nullable *iff $\epsilon \in \mathcal{L}(r)$. Function $\nu()$ tests if an expression is nullable.*

$$\begin{aligned}\nu(\epsilon_l) &= \mathsf{true}\\ \nu(x_l) &= \mathsf{false}\\ \nu(r+s) &= \nu(r) \vee \nu(s)\\ \nu(r \cdot s) &= \nu(r) \wedge \nu(s)\\ \nu(r^*) &= \mathsf{true}\end{aligned}$$

The definition extends to R as follows:

$$\begin{aligned}\nu(\{\}) &= \mathsf{false}\\ \nu(\{r\} \cup R) &= \nu(r) \vee \nu(R)\end{aligned}$$

Proposition 1. *For any expression r we find that $\epsilon \in \mathcal{L}(r)$ iff $\nu(r)$ holds.*

Next, we introduce partial derivatives. The definition follows [1] with the addition of dealing with labeled subterms.

Definition 3 (Partial Derivatives). *Function $\partial()$ computes the set of partial derivatives for an expression w.r.t. some fixed symbol x. Its definition is as follows:*

$$\begin{aligned}\partial_x(\epsilon_l) &= \{\}\\ \partial_x(x_l) &= \{\epsilon_l\}\\ \partial_x(y_l) &= \{\}\\ \partial_x(r+s) &= \partial_x(r) \cup \partial_x(s)\\ \partial_x(r \cdot s) &= \begin{cases}\partial_x(r) \odot s \cup \partial_x(s) & \textit{if } \nu(r)\\ \partial_x(r) \odot s & \textit{otherwise}\end{cases}\\ \partial_x(r^*) &= \partial_x(r) \odot r^*\end{aligned}$$

where the smart concatenation constructor $\odot$ is defined as follows.

$$r_1 \odot r_2 = \begin{cases} r_2 & r_1 = \epsilon\\ r_1 & r_2 = \epsilon\\ r_1 \cdot r_2 & \textit{otherwise}\end{cases}$$

$$R \odot r_2 = \{r_1 \odot r_2 \mid r_1 \in R\}$$

The definition of $\partial_x(r)$ extends to words as follows. $\partial_\epsilon(r) = \{r\}$. $\partial_{x\cdot w}(r) = \bigcup_{s\in\partial_x(r)} \partial_w(s)$.

We say that the elements in $\partial_x(r)$ are the partial derivatives *of r w.r.t. symbol x. A* descendant *of some expression r is either r, or a partial derivative of r or a partial derivative of a descendant of r. Often, we refer to descendants as partial derivatives.*

For example, we find that

$$\begin{array}{l} \quad \partial_x((x \cdot y + x)^*) \\ = \partial_x(x \cdot y + x) \odot (x \cdot y + x)^* \\ = \{y, \epsilon\} \odot (x \cdot y + x)^* \\ = \{y \cdot (x \cdot y + x)^*, (x \cdot y + x)^*\} \end{array}$$

3.1 Regular Expression Matching via Partial Derivatives

We summarize the matching method based on partial derivatives.

Proposition 2 (Antimirov [1]). *For any expression r and symbol x we find that $\mathcal{L}(\partial_x(r)) = \{w \mid x \cdot w \in \mathcal{L}(r)\}$.*

Definition 4 (Matching).

$$\begin{array}{r} \mathsf{match}_\epsilon(R) = \nu(R) \\ \mathsf{match}_{x\cdot w}(R) = \bigvee_{r\in R} \mathsf{match}_w(\partial_x(r)) \end{array}$$

Proposition 3. *For any expression r and word w we find that $w \in \mathcal{L}(r)$ iff $\mathsf{match}_w(\{r\})$ holds.*

As we have seen earlier, $\mathsf{match}_{x\cdot y}(\{(x \cdot y + x)^*\})$ holds. In general, it is more effective to pre-compute the partial derivatives arising from an expression instead of computing them on the fly during matching. This is possible because Antimirov proved that the set of descendants is finite [1]. As we are mainly interested in dealing with matching failure (where the expression changes) such a treatment is not necessary here.

Before we can present the details of our approach how to fix matching failure, we need to establish some important properties how subterms in partial derivatives are connected to subterms in the originating expression. First, we repeat some results already present in [1].

3.2 Partial Derivative Subterms

From Antimirov [1] we know that partial derivatives (and their descendants) of some expression r are of a particular shape.

Definition 5 (Partial Derivative Subterms). *We write $r[s]$ to denote a subterm s within expression r. We define a restricted form of expressions w.r.t. some expression r which we abbreviate by p_r or q_r.*

$$p_r, q_r ::= \epsilon_l \mid r \mid s_1 \cdot \ldots \cdot s_n$$

where for each s_i we have that $r[s_i]$ where $s_i \neq \epsilon$.

Notation p_r makes the initial expression an implicit parameter of all its descendants. This comes in handy when accessing the initial expression given some descendant.

Proposition 4 (Antimirov [1]). *For any expression r and word w we find that each expression t in $\partial_w(r)$ is of the form p_r.*

That is, partial derivatives (and their descendants) of some expression r are either r itself, the empty word, or a concatenation of the form $s_1 \cdot ... \cdot s_n$ where s_i are subterms of r. The last case includes the special case s where s is a subterm of r. The result is rather straightforward by observing the definition of $\partial()$. Recall the application of the smart constructor for concatenation. Therefore, we find that $s_i \neq \epsilon$.

The above result assumes the use of the associativity law $r \cdot (s \cdot t) = (r \cdot s) \cdot t$ which allows us to drop parentheses in case of concatenated partial derivatives of the form $s_1 \cdot ... \cdot s_n$. For example, by definition we find that $\partial_x(((x \cdot y) \cdot z) \cdot z) = \{(y \cdot z) \cdot z\}$. By application of associativity, we consider $(y \cdot z) \cdot z$ as isomorphic to $y \cdot z \cdot z$.

Next, we establish some further important properties about partial derivatives subterms.

3.3 Tracing and Extension of Simple Partial Derivative Subterms

In our formulation, we attach labels to symbols and ϵ to make stronger statements about partial derivatives and their subterms in connection to the original expression.

As stated earlier, we assume that expressions are properly labeled. This property does not apply to partial derivatives. For example, consider

$$\partial_x((x_1 \cdot x_2)^*) = \{x_2 \cdot (x_1 \cdot x_2)^*\}$$

where label 2 appears twice.

However, we can guarantee that if there are several occurrences of the same label, then the expressions connected to that label are all identical. We first introduce some definitions before stating this property more formally.

Definition 6 (Simple Subterms). *We write γ, δ to denote (unlabeled) simple subterms representing either a symbol x or the empty word ϵ. With labels we simply write γ_l and δ_l.*

Proposition 5 (Identical Simple Subterms). *Let p_r be a descendant of some expression r such that $p_r[\delta_l]$ and $p_r[\gamma_l]$ for some simple subterms γ, δ and some label l. Then, we have that γ and δ are identical.*

Proof. (Sketch) This property is due to the smart constructor. The only time the expression connected to a label changes is in case $\partial_x(x_l) = \{\epsilon_l\}$. By definition of the partial derivative operation this will only ever take place on the left subexpression in a concatenated expression. Thanks to the smart constructor, the 'empty' left subexpression will be removed immediately. □

The above property justifies to strengthen Definition 5. In case of simple subterms connected to the same label, these subterms must be identical. Hence, the following definition is well-defined.

Definition 7 (Label Occurrences). *Let p_r be some descendant of some expression r. We write $p_r[\![\delta_l]\!]$ to refer to all occurrences of label l in r for some δ.*

For p_r of shape r find that $r[\![\delta_l]\!]$ can be replaced by $r[\delta_l]$ due to the assumption that the initial expression is properly labeled.

The next property establishes that simple subterms connected to the same label can be traced back to the expression they are derived from.

We write $\delta = \epsilon \vee y$ to denote that δ either equals ϵ or some symbol y.

Proposition 6 (Tracing of Simple Subterms). *Let r be some expression and p_r and q_r some descendants such that $p_r[\![\delta_l]\!] \in \partial_x(q_r)$ for some symbol x where $\delta = \epsilon \vee y$ for some symbol y and $p_r \neq \epsilon$. Then, $q_r[\![\delta_l]\!]$.*

Proof. (Sketch) Partial derivatives are composed of subexpressions of the originating expression. Hence, label l must exist in r. We distinguish among the following cases. (1) The partial derivative operation has not been applied on the expression connected to label l. So obviously if $p_r[\![\delta_l]\!] \in \partial_x(q_r)$, then $q_r[\![\delta_l]\!]$. Also recall Proposition 5. (2) The partial derivative operation has been applied on the expression connected to label l. Cases $\partial_x(\epsilon_l) = \{\}$ and $\partial_x(y_l) = \{\}$ do not apply as label l occurs in p_r. So the only relevant case is $\partial_x(x_l) = \{\epsilon_l\}$. We know that due to the smart constructor ϵ_l has been removed. As we assume that label l occurs in p_r and $p_r \neq \epsilon$ it can only be the case that $y = x$. Hence, we again have established the fact that if $p_r[\![\delta_l]\!] \in \partial_x(q_r)$, then $q_r[\![\delta_l]\!]$. □

For example, recall the earlier example

$$\partial_x((x_1 \cdot x_2)^*) = \{x_2 \cdot (x_1 \cdot x_2)^*\}$$

where subterms x_1 and x_2 in the derived partial derivative can be traced back to the originating expression.

Based on the tracing of simple subterms property, we can even add alternatives while maintaining that extended expressions are still in partial derivative relation.

Definition 8 (Alternative Extension). *Let p_r be a descendant of r where $p_r[\![\delta_l]\!]$ for some simple subterm δ and label l. Let t be some expression. Then, we write $p_r[\![\delta_l]\!] \xrightarrow{+t} p_r[\![\delta_l + t]\!]$ to denote that all occurrences of δ_l in p_r are replaced by $\delta_l + t$ which then results in the expression $p_r[\![\delta_l + t]\!]$. We refer to $p_r[\![\delta_l + t]\!]$ as the* extension *of $p_r[\![\delta_l]\!]$.*

For example, consider building the extension for the partial derivative $x_2 \cdot (x_1 \cdot x_2)^*$ and its originating expression $(x_1 \cdot x_2)^*$.

$$x_2 \cdot (x_1 \cdot x_2)^* [\![x_2]\!] \xrightarrow{+x_3 \cdot y_4} (x_2 + x_3 \cdot y_4) \cdot (x_1 \cdot (x_2 + x_3 \cdot y_4))^*$$

$$(x_1 \cdot x_2)^* [\![x_2]\!] \xrightarrow{+x_3 \cdot y_4} (x_1 \cdot (x_2 + x_3 \cdot y_4))^*$$

In the first case, there are two occurrences of x_2. Hence, the replacement step is applied twice.

An important property is that the thus extended partial derivative is in the set of the extended expression. For our example, the extended partial derivative $(x_2 + x_3 \cdot y_4) \cdot (x_1 \cdot (x_2 + x_3 \cdot y_4))^*$ is again in the set of partial derivatives (w.r.t. symbol x) of the extended expression $(x_1 \cdot (x_2 + x_3 \cdot y_4))^*$ as we find that

$$(x_2 + x_3 \cdot y_4) \cdot (x_1 \cdot (x_2 + x_3 \cdot y_4))^* \in \partial_x((x_1 \cdot (x_2 + x_3 \cdot y_4))^*)$$

Here is the general formalization of this property.

Proposition 7 (Extension of Simple Subterms). *Let r be some expression and p_r and q_r some descendants such that $p_r[\![\delta_l]\!] \in \partial_x(q_r)$ for some symbol x where $\delta = \epsilon \vee y$ for some symbol y and $p_r \neq \epsilon$. Let t be some expression. Let $p_r[\![\delta_l]\!] \xrightarrow{+t} p_r[\![\delta_l + t]\!]$ and $q_r[\![\delta_l]\!] \xrightarrow{+t} q_r[\![\delta_l + t]\!]$. Then, we find that $p_r[\![\delta_l + t]\!] \in \partial_x(q_r[\![\delta_l + t]\!])$. Furthermore, $\mathcal{L}(q_r[\![\delta_l]\!]) \subseteq \mathcal{L}(q_r[\![\delta_l + t]\!])$.*

The proof argument is similar to Proposition 6. As the extension is built by providing alternatives for simple subterms it follows immediately that the extension subsumes the original expression.

By slightly abusing notation, the above proposition guarantees that

$$\partial_x(q_r)[\![\delta_l + t]\!] \subseteq \partial_x(q_r[\![\delta_l + t]\!]) \tag{1}$$

The subset relation is proper for certain examples as shown by the following extension for our running example.

$$x_2 \cdot (x_1 \cdot x_2)^* [\![x_1]\!] \xrightarrow{+x_3 \cdot y_4} x_2 \cdot ((x_1 + x_3 \cdot y_4) \cdot x_2)^*$$

$$(x_1 \cdot x_2)^* [\![x_1]\!] \xrightarrow{+x_3 \cdot y_4} ((x_1 + x_3 \cdot y_4) \cdot x_2)^*$$

We find that

$$\partial_x(((x_1 + x_3 \cdot y_4) \cdot x_2)^*) = \{x_2 \cdot ((x_1 + x_3 \cdot y_4) \cdot x_2)^*, y_4 \cdot x_2 \cdot ((x_1 + x_3 \cdot y_4) \cdot x_2)^*\}$$

The element $y_4 \cdot x_2 \cdot ((x_1 + x_3 \cdot y_4) \cdot x_2)^*$ shows that the above subset relation (1) is proper.

The tracing and extension property are key technical ingredients in our algorithm to fix regular expressions in case of matching failure. This is what we discuss next.

4 Fixing Matching Failure

We incrementally introduce the various components to fix matching failure based on partial derivatives. The actual algorithm is presented at the end.

First, we introduce two new forms of expressions to keep track of mismatches.

$$\begin{array}{l} f ::= (x \mid \not{y})_l \mid (x \mid \not{\epsilon})_l \\ r ::= \epsilon_l \mid x_l \in \Sigma \mid r + r \mid r \cdot r \mid r^* \mid f \end{array}$$

The form $(x \mid \not{y})_l$ indicates that at location l we encounter y where we expect x. Similarly, the form $(x \mid \not{\epsilon})_l$ indicates that the expected x does not match the present ϵ.

These forms only arise in partial derivatives. The adjusted definition is as follows.

Definition 9 (Faulty Partial Derivatives).

$$\begin{array}{rcl} \partial_x(\epsilon_l) &=& \{(x \mid \not{\epsilon})_l\} \\ \partial_x(x_l) &=& \{\epsilon_l\} \\ \partial_x(y_l) &=& \{(x \mid \not{y})_l\} \\ \partial_x(r+s) &=& \partial_x(r) \cup \partial_x(s) \\ \partial_x(r \cdot s) &=& \begin{cases} \partial_x(r) \odot s \cup \partial_x(s) & \mathit{if}\ \nu(r) \\ \partial_x(r) \odot s & \mathit{otherwise} \end{cases} \\ \partial_x(r^*) &=& \partial_x(r) \odot r^* \end{array}$$

We adjust Definition 5 to include faulty subterms which by construction must occur in leading position.

Definition 10 (Partial Derivative Subterms with Faults).

$$p_r, q_r ::= \epsilon_l \mid r \mid s_1 \cdot \ldots \cdot s_n \mid f \cdot s_1 \cdot \ldots \cdot s_n$$

where for each s_i we have that $r[s_i]$ where $s_i \neq \epsilon$. We say that p_r is faulty *if p_r is of the form $f \cdot s_1 \cdot \ldots \cdot s_n$. This includes the special case f. Otherwise, we say that p_r is* non-faulty.

In case we encounter a faulty partial derivative, we use the attached mismatch information to resolve the conflict. We crucially rely on the property that partial derivative subterms are subterms in expressions they are derived from. Hence, in case of a $(x \mid \not{y})_l$ mismatch we simply need to provide the alternative x at all occurrences of y_l. Recall that we introduce alternatives because our goal is not to alter the set of words accepted by the expression.

A $(x \mid \not{\epsilon})_l$ mismatch indicates that we encounter ϵ where we expect x. The simple subterm in the originating expression may be either ϵ or a symbol. For example, consider expression y_1 and input $y \cdot x$ for which we find

$$\begin{array}{rcl} \{\epsilon_1\} &=& \partial_y(y_1) \\ \{(x \mid \not{\epsilon})_1\} &=& \partial_x(\epsilon_1) \end{array}$$

The fix here is to replace y_1 by $y_1 + y_2 \cdot x_3$.

For expression ϵ_1 and input x, we need to provide for the alternative to match against the empty word. In general, we encounter a simple term δ_l. To resolve the mismatch we simply provide the alternative $\delta_{l_2} \cdot x_{l_3}$.

Here is the formal definition to resolve mismatches.

Definition 11 (Mismatch Resolution). *Let p_r be a non-faulty partial derivative. We write $p_r[\![y_l]\!] \xrightarrow{(x|\cancel{y})_l} p_r[\![y_l + x_{l_2}]\!]$ to denote that all occurrences of y_l in p_r are replaced by $y_l + x_{l_2}$ where l_2 is some fresh label.*

Similarly, we write $p_r[\![\delta_l]\!] \xrightarrow{(x|\cancel{\epsilon})_l} p_r[\![\delta_l + \delta_{l_2} \cdot x_{l_3}]\!]$ to denote that all occurrences of δ_l in p_r are replaced by $\delta_l + \delta_{l_2} \cdot x_{l_3}$ where l_2, l_3 are fresh labels.

The introduction of fresh labels is important in case the resolution step is applied on the initial expression. Thus, the resulting expression remains properly labeled.

The following results guarantee that the mismatch resolutions indeed resolve a mismatch. We first consider the case of a mismatched symbol.

We write $x^1 \cdot ... \cdot x^n$ to denote a word consisting of n symbols. Notation x^i refers to the ith position. We use superscript to avoid confusion with labels attached to simple terms.

Proposition 8 (Symbol Mismatch Resolution). *Let r, p_{n+1_r}, ..., p_{1_r}, x^1, ..., x^n such that $p_{1_r} = r$ and $p_{i+1_r} \in \partial_{x^i}(p_{i_r})$ for $i = 1, ..., n$ where p_{1_r}, ..., p_{n_r} are non-faulty and p_{n+1_r} is faulty of the form $(x^n \mid \cancel{y})_l \cdot s_1 \cdot ... \cdot s_m$ for some label l, symbol y and some subterms s_i of r. Let $r \xrightarrow{(x^n|\cancel{y})_l} r'$ and $s_j \xrightarrow{(x^n|\cancel{y})_l} s'_j$ for $j = 1, ..., m$. Then, we find that $s'_1 \cdot ... \cdot s'_m \in \partial_{x^1 \cdot ... \cdot x^n}(r')$. Furthermore, $\mathcal{L}(r) \subseteq \mathcal{L}(r')$.*

For ϵ mismatch, a similar result applies.

Proposition 9 (ϵ Mismatch Resolution). *Let r, p_{n+1_r}, ..., p_{1_r}, x^1, ..., x^n such that $p_{1_r} = r$ and $p_{i+1_r} \in \partial_{x^i}(p_{i_r})$ for $i = 1, ..., n$ where p_{1_r}, ..., p_{n_r} are non-faulty and p_{n+1_r} is faulty of the form $(x^n \mid \cancel{\epsilon})_l \cdot s_1 \cdot ... \cdot s_m$ for some label l and some subterms s_i of r. Let $r \xrightarrow{(x^n|\cancel{\epsilon})_l} r'$ and $s_j \xrightarrow{(x^n|\cancel{\epsilon})_l} s'_j$ for $j = 1, ..., m$. Then, either (1) $\epsilon_{l'} \in \partial_{x^1 \cdot ... \cdot x^n}(r')$ for some label l' or (2) $s'_1 \cdot ... \cdot s'_m \in \partial_{x^1 \cdot ... \cdot x^n}(r')$. Furthermore, $\mathcal{L}(r) \subseteq \mathcal{L}(r')$.*

Besides a mismatch due to an unmatched symbol, failure may arise because of a non-nullable final expression. For example, consider expression $x_1 \cdot y_2$ and input x where we find

$$\{y_2\} = \partial_x(x_1 \cdot y_2)$$

Hence, we perform a replacement similar to the above.

Unlike in case of mismatch resolution where for each faulty partial derivative there is only one mismatch (in the leading subterm), we may need to resolve several subterms. For example, consider

$$\{y_2 \cdot z_3\} = \partial_x(x_1 \cdot y_2 \cdot z_3)$$

where we need to make labels 2 and 3 nullable.

Identifying to be made nullable labels can be done by observing the structure of the (non-nullable) expression following Definition 2.

Definition 12 (Non-Nullable Resolution). *Function* $\mathsf{mkNullable}()$ *computes the set of labels that are non-nullable.*

$$\begin{aligned}\mathsf{mkNullable}(\epsilon_l) &= \{\}\\ \mathsf{mkNullable}(x_l) &= \{l\}\\ \mathsf{mkNullable}(r+s) &= \mathsf{mkNullable}(r) \cup \mathsf{mkNullable}(s)\\ \mathsf{mkNullable}(r\cdot s) &= \mathsf{mkNullable}(r) \cup \mathsf{mkNullable}(s)\\ \mathsf{mkNullable}(r^*) &= \{\}\end{aligned}$$

The expression is made nullable by applying the following rule:

$$\frac{l_1, l_2 \text{ are fresh labels}}{p_r[\![\delta_l]\!] \xrightarrow{l} p_r[\![\delta_{l_1} + \epsilon_{l_2}]\!]}$$

$$p_r \xrightarrow{\{\}} p_r \qquad \frac{p_r \xrightarrow{l} q_r}{p_r \xrightarrow{\{l\}} q_r} \qquad \frac{p_r \xrightarrow{L_1} p'_r \quad p'_r \xrightarrow{L_1} p''_r}{p_r \xrightarrow{L_1 \cup L_2} p''_r}$$

where L_1 *and* L_2 *denote sets of labels.*

We perform an over-approximation. For example, in case of alternatives it is sufficient to collect the labels from say the left component only. We will discuss such design choices shortly.

Proposition 10 (Non-Nullable Resolution). *Let* r, p_{n+1_r}, ..., p_{1_r}, x^1, ..., x^n *such that* $p_{1_r} = r$ *and* $p_{i+1_r} \in \partial_{x^i}(p_{i_r})$ *for* $i = 1, ..., n$ *where* p_{i_r} *are not faulty. Let* $L = \mathsf{mkNullable}(p_{n+1_r})$ *and* $r \xrightarrow{L} r'$ *for some* r'. *Then, we find that there exists* $q_r \in \partial_{x^1\cdot\ldots\cdot x^n}(r')$ *such that* q_r *is nullable. Furthermore,* $\mathcal{L}(r) \subseteq \mathcal{L}(r')$.

We have now everything in place to define the algorithm to fix regular expressions in case of matching failure. We use Haskell-style syntax.

Definition 13 (Fixing Mismatches).

```
matchFix2 $R$ $\epsilon$
 | $\exists\, r \in R.\, \nu(r) = \{\,\}$
 | otherwise $= \{\, r' \mid p_r \in R \wedge r \xrightarrow{\mathsf{mkNullable}(p_r)} r' \,\}$
matchFix2 $R$ $x \cdot w$ =
   let $R' = \bigcup_{p_r \in R} \partial_x(p_r)$
       $F = \{\, p_r \in R' \mid p_r \textit{ faulty} \,\}$
       $N = \{\, p_r \in R' \mid p_r \textit{ not faulty} \,\}$
   in if $N \neq \{\,\}$
      then matchFix2 $N$ $w$
      else $\{\, r' \mid p_r = f \cdot s_1 \cdot \ldots \cdot s_n \in F \wedge r \xrightarrow{f} r' \,\}$
```

```
matchFix $r\ w$ =
  let $R'$ = matchFix2 $\{ r \}$ $w$
  in if $R' = \{ \}$
     then $\{ r \}$
     else matchFix $R'$ $w$
matchFix $R$ $w$ = $\bigcup_{r \in R}$ matchFix $r$ $w$
```

Helper matchFix2 takes a set R of descendants of some expression r and some input word w. The initial call is matchFix2 $\{r\}$ w. If the input word is empty, we check if there is any nullable descendant. If yes, the match is successful. Nothing needs to be done and we simply return the empty set $\{\}$. If none of the descendants is nullable, we resolve this form of mismatch as described in Definition 12. The resolution step is applied on the initial expression r which we can access via notation p_r. That is, expression r is an implicit parameter of all its descendants. Proposition 10 guarantees that the fix is effective.

For a non-empty word, we build the set of (immediate) descendants for each expression in R. If we find some non-faulty descendants (see N) we continue as in case of the standard matching algorithm (see Definition 4). Otherwise, we resolve the mismatch as described in Definition 11 where the corresponding Proposition 8 guarantees that the fix is effective.

Function matchFix calls matchFix2 to check if there are any fixes necessary. If not we simply return the expression. Otherwise, we run matchFix (and thus matchFix2) until all mismatches have been resolved.

Correctness of the algorithm follows immediately from our earlier results (Propositions 8, 9 and 10).

Proposition 11 (Fixing Expressions). *Let r be an expression and w be a word. Then,* matchFix r w *yields a finite set R for any $r' \in R$ we have that $w \in \mathcal{L}(r')$ and $\mathcal{L}(r) \subseteq \mathcal{L}(r')$.*

It is also easy to see that the (fixing) process must eventually terminate as the size of the input word is of a fixed length.

5 Related Work

As far as we know there is no related work which attempts to fix faulty regular expressions. In the following, we summarize works which have some connections to our work.

Debugging There exist several tools, some are commercial [5, 12] and some are public domain [11, 3], which provide highlighting facilities to visualize the matching process for regular expressions. In case of matching failure, the erroneous position in the input string is typically highlighted. Based on the matching history (up to the point of failure) the user obtains information how to possible fix the failure. Such a form of debugging should be easily supported by our approach. We give a brief discussion in the conclusion.

Synthesis The works in [2, 8] apply machine learning and genetic programming techniques to derive an expression which (not) matches a fixed set of positive (negative) inputs. For example, the work in [8] starts of with an initial expressions r_0 and attempts to find a refinement r of r_0 which still satisfies all inputs where $\mathcal{L}(r) \subseteq \mathcal{L}(r_0)$. In our setting, we assume that new inputs arrive which possible requires to extend the expression r. Hence, learning of regular expressions and our approach of fixing regular expression have different goals.

Parsing Parsing commonly refers to matching in case of context-free grammars. Recovering from parsing failure is a classic topic. The two common approaches are either to fix the input, e.g. consider[9, 7, 6], or simply to skip as much as possible erroneous parts of the input, e.g. consider [10]. Transferring ideas of our work to the setting of parsing is an interesting topic for future work which of course requires first to establish the notation of partial derivatives for context-free grammars.

6 Conclusion

We have introduced a novel method to fix regular expressions in case of matching failure. This is something which to the best of our knowledge has not been attempted before. There are several avenues for future work which we discuss in the following.

Based on the mismatch information, we could provide detailed debugging information for the user. For example, by highlighting the erroneous parts and support a user-guided resolution mechanism. This is useful in the context of teaching regular expressions but also has practical applications, e.g. assist test engineers to fix erroneous specifications in the context of formal testing. We plan to build such a regular expression debugging tool and explore its applications.

We argue that the fixes we provide are small as we only adjust the label position connected to a simple subterm. Can we guarantee that fixes are *minimal*? A fix is minimal if we apply the least number of resolution steps to obtain an expression which accepts the word.

We claim that for many cases we find minimal fixes. As discussed, we would need to adjust Definition 12 to guarantee that the non-nullable resolution steps are minimal. This is straightforward. For brevity, we omit the details. However, there are still some corner cases left for which our method does not yield minimal fixes.

For example, consider expression $x \cdot y$ and input y. Our method yields the fix $(x + y) \cdot (y + \epsilon)$. As we will see, this fix is not minimal. First, we observe the computation steps to obtain this fix. In a first step, we encounter the mismatch $(y \mid \not{x}) \cdot y$ which yields the intermediate fix $(x + y) \cdot y$ where for brevity we ignore labels. In the next round, matching of $(x + y) \cdot y$ against word y yields y. As this (final) expression is not nullable, we apply another fix which the results into $(x + y) \cdot (y + \epsilon)$.

It is easy to see that this fix is not minimal as we could apply the non-nullable resolution step immediately on subterm x. This then results into the minimal fix $(x + \epsilon) \cdot y$. Incorporating this heuristic into our method is straightforward. Briefly, for each partial derivative p_r of shape $s \cdot t$ where s is not nullable we build $r \xrightarrow{\mathsf{mkNullable}(s)} r'$. The process then starts again with r'.

Another question is how to select the *best* fix in case there are several minimal fixes. Recall the example from Section 2. For expression $(x \cdot y + x)^*$ and input $x \cdot z$ our method yields the fixes (1) $(x \cdot y + x + z)^*$, (2) $(x \cdot (y + z) + x)^*$ and (3) $((x + z) \cdot (y + \epsilon) + x)^*$ where (1) and (2) are minimal. How to give a precise characterization that one improves over the other? One idea is to look into matching policies such as greedy and POSIX and adjust them to our setting. This is some topic for future work.

References

1. V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
2. Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Playing regex golf with genetic programming. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 1063–1070, New York, NY, USA, 2014. ACM.
3. Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. Regviz: Visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 504–507, New York, NY, USA, 2014. ACM.
4. J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
5. Debuggex: Online visual regex tester. javascript, python, and pcre. `http://www.debuggex.com/`.
6. Charles N. Fischer, D. R. Milton, and S. B. Quiring. Efficient LL(1) error correction and recovery using only insertions. *Acta Inf.*, 13:141–154, 1980.
7. J. C. C. Hermeler. Error correction and recovery in a ll(1) parser. http://gerbil.org/ftp/pub/tom/papers/ErrorRec.ps.gz, 1998.
8. Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, and Shivakumar Vaithyanathan. Regular expression learning for information extraction. In *Proc. of CEMNLP'08*, pages 21–30, 2008.
9. Ajit B. Pai and Richard B. Kieburtz. Global context recovery: A new strategy for syntactic error recovery by table-drive parsers. *ACM Trans. Program. Lang. Syst.*, 2(1):18–41, January 1980.
10. Roman R. Redziejowski. Mouse: From parsing expressions to a practical parser. In *Proc. o the CS&P 2009 Workshop*, pages 514–525, 2009.
11. Regex101: Pcre-based regular expression debugger with real time explanation, error detection and highlighting. `https://regex101.com/`.
12. Regexbuddy: Learn, create, understand, test, use and save regular expression. `http://www.regexbuddy.com/`.

A Proofs

A.1 Proof of Proposition 8

Statement: Let r, p_{n+1_r}, ..., p_{1_r}, x^1, ..., x^n such that $p_{1_r} = r$ and $p_{i+1_r} \in \partial_{x^i}(p_{i_r})$ for $i = 1, ..., n$ where p_{1_r}, ..., p_{n_r} are non-faulty and p_{n+1_r} is faulty of the form $(x^n \mid \not{y})_l \cdot s_1 \cdot ... \cdot s_m$ for some label l, symbol y and some subterms s_i of r. Let $r \xrightarrow{(x^n|\not{y})_l} r'$ and $s_j \xrightarrow{(x^n|\not{y})_l} s'_j$ for $j = 1, ..., m$. Then, we find that $s'_1 \cdot ... \cdot s'_m \in \partial_{x^1 \cdot ... \cdot x^n}(r')$. Furthermore, $\mathcal{L}(r) \subseteq \mathcal{L}(r')$.

Proof. The statement $\mathcal{L}(r) \subseteq \mathcal{L}(r')$ follows immediately as we obtain r' from r by providing alternatives for simple subterms.

The mismatch arises in the last step where we find that $(x^n \mid \not{y})_l \cdot s_1 \cdot ... \cdot s_m \in \partial_{x^n}(p_{n_r}[\![y_l]\!])$. By the tracing property (Proposition 6), we have that $r[y_l]$.

Hence, from $r \xrightarrow{(x^n|\not{y})_l} r'$ we can deduce that $r' = r[y_l + {x^n}_{l_2}]$ for some label l_2.

Via application of the extension property ((Proposition 7) we can argue that $p_{n_r}[\![y_l]\!] + {x^n}_{l_2} \in \partial_{x^1 \cdot ... \cdot x^{n-1}}(r')$.

From above and by observing the definition of the partial derivative operation we can follow that $s'_1 \cdot ... \cdot s'_m \in \partial_{x^n}(p_{n_r}[\![y_l]\!] + {x^n}_{l_2})$.

Hence, $s'_1 \cdot ... \cdot s'_m \in \partial_{x^1 \cdot ... \cdot x^n}(r')$ and we are done. $\square$

A.2 Proof of Proposition 9

Statement: Let r, p_{n+1_r}, ..., p_{1_r}, x^1, ..., x^n such that $p_{1_r} = r$ and $p_{i+1_r} \in \partial_{x^i}(p_{i_r})$ for $i = 1, ..., n$ where p_{1_r}, ..., p_{n_r} are non-faulty and p_{n+1_r} is faulty of the form $(x^n \mid \not{\epsilon})_l \cdot s_1 \cdot ... \cdot s_m$ for some label l and some subterms s_i of r. Let $r \xrightarrow{(x^n|\not{\epsilon})_l} r'$ and $s_j \xrightarrow{(x^n|\not{\epsilon})_l} s'_j$ for $j = 1, ..., m$. Then, either (1) $\epsilon_{l'} \in \partial_{x^1 \cdot ... \cdot x^n}(r')$ for some label l' or (2) $s'_1 \cdot ... \cdot s'_m \in \partial_{x^1 \cdot ... \cdot x^n}(r')$. Furthermore, $\mathcal{L}(r) \subseteq \mathcal{L}(r')$.

Proof. Statement $\mathcal{L}(r) \subseteq \mathcal{L}(r')$ follows immediately.

For the remaining part, we distinguish among the following two cases.

Case $r[\epsilon_l]$: That is, ϵ_l is as a subterm in r. The ϵ mismatch failure only arises in the last step. Hence, for the earlier steps we find that $p_{k+1_r}[\![\epsilon_l]\!] \in \partial_{x^k}(p_{k_r}[\![\epsilon_l]\!])$ for $k = 1, ..., n-1$. By application of the extension property (Proposition 7) we can argue that $p_{k+1_r}[\![\epsilon_l + \epsilon_{l_2} \cdot {x^n}_{l_3}]\!] \in \partial_{x^k}(p_{k_r}[\![\epsilon_l + \epsilon_{l_2} \cdot {x^n}_{l_3}]\!])$ for some fresh labels l_2, l_3 for $k = 1, ..., n-1$.

Let's consider the last (failure) step $(x^n \mid \not{\epsilon})_l \cdot s_1 \cdot ... \cdot s_m \in \partial_{x^n}(p_{n_r}[\![\epsilon_l]\!])$. By observing the definition of the partial derivative operation we find that $s'_1 \cdot ... \cdot s'_m \in \partial_{x^n}(p_{n_r}[\![\epsilon_l + \epsilon_{l_2} \cdot {x^n}_{l_3}]\!])$.

Hence, for $m \geq 1$ we can establish (2), otherwise, (1) holds.

Case $\neg r[\epsilon_l]$: That is, ϵ_l is not a subterm in r. Hence, the term ϵ_l arises in some intermediate step. By definition of the partial derivative operation and due to the application of the smart constructor, it must be the case that $\epsilon_l \in \partial_{x^1 \cdot ... \cdot x^{n-1}}(r)$. That is, $p_{n_r} = \epsilon_l$. More specifically, we can argue that $\epsilon_l \in \partial_{x^{n-1}}(p_{n-1_r}[\![{x^{n-1}}_l]\!])$.

By the tracing property (Proposition 6), we have that $r[{x^{n-1}}_l]$. Hence, from $r \xrightarrow{(x^n|\not\epsilon)_l} r'$ we can deduce that $r' = r[{x^{n-1}}_l + {x^{n-1}}_{l_2} \cdot {x^n}_{l_3}]$ for some labels l_2, l_3.

Via application of the extension property ((Proposition 7) we can argue that ${x^n}_{l_3} \in \partial_{x^1 \cdot \ldots \cdot x^{n-1}}(r')$. Hence, $\epsilon_{l_3} \in \partial_{x^1 \cdot \ldots \cdot x^n}(r')$ and we are done. $\square$