

Progress Report - 9 Months

Meshal Binnasban

September 10, 2025

Abstract

This is the nine-month progress report for my PhD at King's College London. It summarises the motivation, challenges, and progress made in developing a marked approach-based regular expression matcher with the goal of POSIX value extraction. The report reviews the derivative-based approach and a possible correspondence with the marked approach—a connection we aim to investigate. It also reviews the marked algorithms of Fischer et al. and Asperti et al., which provide matchers only and do not support value extraction. Our work develops several versions of the marked algorithm in Scala aimed at extracting POSIX values. During this process, challenges in handling certain cases led us to refine the algorithm through successive versions, each improving on the last. Future directions include extending the matcher to additional operators, refining disambiguation for repetitions, and formally proving correctness.

Synopsis

This research investigates the marked approach-based method for regular expression matching, with the goal of extending it to provide POSIX value extraction.

Derivative-based methods, though elegant, suffer from severe size explosion and remain sensitive to syntactic form. Sulzmann and Lu [9] extended derivatives with bitcodes to record lexing information, but the size problem persists. The marked approach, described in the works of Fischer et al. [4] and Asperti et al. [2], offers an alternative by propagating marks directly through the expression. These works provide matchers only, without value extraction.

Our work explores how the marked approach can be extended to recover POSIX values. We have implemented several versions of the marked algorithm in Scala, making use of bitcoded annotations to track lexing values. Challenges in handling constructs such as sequences and repetitions required refining the algorithm through successive versions, each improving on the last. Large-scale testing against a derivative-based reference matcher has been used to confirm correctness and uncover edge cases.

The long-term aim of the project is to establish a marked approach-based matcher that consistently yields the POSIX-preferred value for all regular expressions, and to formally prove its correctness. Future work also includes extending the matcher to additional operators such as intersection and negation.

Contents

1	Introduction	4
2	Background	4
2.1	Derivatives	4
2.1.1	Derivative Extension	5
2.1.2	Size Explosion.	8
2.2	Marked Approach	9
2.2.1	Motivation for a Marked Approach	10
2.2.2	Scala Implementation of Fischer’s Marked Approach	10
3	Our Approach	13
3.1	Bit-Annotated, version 1	14
3.2	Bit-Annotated, version 2	18
3.3	String-Carrying Marks	18
4	Future Work	18
	References	20

1 Introduction

The notion of derivatives in regular expressions is well established but has gained renewed attention in the last decade [8, 5]. Their simplicity and compatibility with functional programming have encouraged further study. However, derivatives suffer from growth issues, since each step of taking the derivative can increase the size of subexpressions, as will be reviewed later.

The marked approach-based method propagates marks through the regex without creating new subexpressions and offers a potential replacement for derivatives. At present, only matcher-based algorithms using this approach exist, while our work aims to extend it to provide POSIX value extraction.

This report first reviews Brzozowski derivatives and the bitcoded variant by Sulzmann and Lu [9], followed by background on the marked approach algorithms described by Fischer et al. [4] and Asperti et al. [2]. We then present our work so far, including several versions of the marked algorithm developed during this period.

2 Background

2.1 Derivatives

Brzozowski's derivatives offer an elegant way for string matching. By successively taking the derivative of a regular expression with respect to each input character, one obtains a sequence of derivatives. If the final expression can match the empty string, then the original input is accepted.

To decide whether a string $a_1a_2\dots a_n$ is in the language of a regular expression r , we successively compute its derivatives:

$$r_0 = r, \quad r_1 = \text{der}_{a_1}(r_0), \quad \dots, \quad r_n = \text{der}_{a_n}(r_{n-1}).$$

The string matches r if and only if the final expression r_n accepts the empty string. Here $\text{der}_{a_n}(r)$ stands for the derivative of r with respect to the character a_n , as introduced by Brzozowski [3].

To illustrate how derivatives can be used to match a regular expression against a string, consider the regular expression $(ab+ba)$. The derivative can check the matching of the string **ba** by taking the derivative of the regular

expression with respect to **b**, then **a**.

$$\begin{aligned}
der_b r &= der_b (ab + ba) \\
&= der_b (ab) + der_b (ba) \\
&= (der_b a) \cdot b + (der_b b) \cdot a \\
&= \emptyset \cdot b + \varepsilon \cdot a
\end{aligned}$$

$$\begin{aligned}
der_a (der_b r) &= der_a (\emptyset \cdot b + \varepsilon \cdot a) \\
&= der_a (\emptyset \cdot b) + der_a (\varepsilon \cdot a) \\
&= der_a (\emptyset) \cdot b + (der_a (\varepsilon)) \cdot a + der_a a \\
&= \emptyset \cdot b + (\emptyset \cdot a + \varepsilon)
\end{aligned}$$

Since the final derivative expression contains ε , it matches the empty string. Because no input characters remain, this confirms that the original string **ba** is in the language of the regular expression.

Some subexpressions that arise during the computation of derivatives may be redundant. For example, $\emptyset \cdot b$ expresses matching the empty language, then b . Since \emptyset is the empty language, no string can satisfy this, so $\emptyset \cdot b$ simplifies to \emptyset . Such simplifications are sometimes necessary in the derivative method. Although the construction is elegant and recursively defined, it may duplicate large parts of the expression and is highly sensitive to the syntactic form of the regular expression. Simplification reduces the number of generated expressions, but it does not solve the underlying problem of size explosion.

More simplification rules may be applied after each derivative to provide a finite bound on the number of intermediate expressions [10]. These include associativity $(r + s) + t \equiv r + (s + t)$, commutativity $r + s \equiv s + r$, and idempotence $r + r \equiv r$. Rules for the empty language and the empty string can also be applied, such as the one mentioned earlier; e.g. $\emptyset \cdot r \equiv r \cdot \emptyset \equiv \emptyset$, $r + \emptyset \equiv r$, and $r \cdot \varepsilon \equiv \varepsilon \cdot r \equiv r$. These simplifications preserve the language accepted by the regular expression while reducing the number of intermediate expressions. They help to mitigate—though not eliminate—the size explosion in derivatives noted by Sulzmann and Lu [9]; see Section 1.2.

2.1.1 Derivative Extension

Sulzmann and Lu [9] extend Brzozowski’s derivatives to produce lexing values in addition to deciding whether a match exists. These values record how the match occurred: which part of the regular expression corresponded to which

part of the string, which alternative branch was taken, and how sequences and Kleene stars were matched.

They provide two variants: a bitcode-based construction and an injection-based construction [9]. In the bitcode variant, bit sequences encoding lexing choices are embedded during derivative construction and, after acceptance, decoded to the value. In the injection variant, an *inj* function "injects back" the consumed characters into the value; it reverts the derivative steps to obtain the POSIX value. We focus on the bitcode variant, which more directly inspired our marked approach.

Bitcodes are sequences over $\{0, 1\}$ that encode the choices made in a match. These bits record branch choices in alternatives and repetitions made during matching.

The following example illustrates how bitcoding works with regular expression $(a + ab)(b + \varepsilon)$ and string ab . We proceed by constructing bitcoded derivatives step by step and tracking how the bitcode grows.

Initially, the algorithm internalizes the regular expression. Internalization only adds bit annotations to the alternative constructors found in the expression, where the left branch is annotated with bitcode 0 and the right branch with bitcode 1 [9]. This annotation process is implemented by the function *fuse*, whose definition is given at the end of this section. After internalization, the algorithm proceeds by taking the derivative with respect to the first character of the input string, which in this case is a .

1. Step 1: Internalizing the regular expression

$$r' = \text{intern}(r) = ({}_0a + {}_1ab) \cdot ({}_0b + {}_1\varepsilon)$$

2. Step 2: input a

$$\begin{aligned} \text{der}_a(r') &= \text{der}_a(({}_0a + {}_1ab) \cdot ({}_0b + {}_1\varepsilon)) \\ &= \text{der}_a({}_0a + {}_1ab) \cdot ({}_0b + {}_1\varepsilon) \\ &= (\text{der}_a({}_0a) + \text{der}_a({}_1ab)) \cdot ({}_0b + {}_1\varepsilon) \\ &= ({}_0\varepsilon + \text{der}_a({}_1a) \cdot b) \cdot ({}_0b + {}_1\varepsilon) \\ &\rightarrow ({}_0\varepsilon + {}_1\varepsilon \cdot b) \cdot ({}_0b + {}_1\varepsilon) \end{aligned}$$

3. Step 3: input b

$$\begin{aligned}
\text{der}_b(\text{der}_a(r')) &= \text{der}_b(({}_0\varepsilon + {}_1\varepsilon \cdot b) \cdot ({}_0b + {}_1\varepsilon)) \\
&= \text{der}_b({}_0\varepsilon + {}_1\varepsilon \cdot b) \cdot ({}_0b + {}_1\varepsilon) + \text{der}_b(\text{fuse}(\text{mkeys}(r_1), ({}_0b + {}_1\varepsilon))) \\
&= (\text{der}_b({}_0\varepsilon) + \text{der}_b({}_1\varepsilon \cdot b)) \cdot ({}_0b + {}_1\varepsilon) + {}_0(\text{der}_b({}_0b) + \text{der}_b({}_1\varepsilon)) \\
&= (\emptyset + \text{der}_b({}_1\varepsilon \cdot b)) \cdot ({}_0b + {}_1\varepsilon) + {}_0({}_0\varepsilon + \emptyset) \\
&= (\text{der}_b({}_1\varepsilon) \cdot b + \text{der}_b({}_1b)) \cdot ({}_0b + {}_1\varepsilon) + {}_0({}_0\varepsilon + \emptyset) \\
&\rightarrow (\emptyset \cdot b + {}_1\varepsilon) \cdot ({}_0b + {}_1\varepsilon) + {}_0({}_0\varepsilon + \emptyset)
\end{aligned}$$

The result of Step 2 shows that the ε symbols indicate a successful match, while the bitcode records how that match was obtained. One match is obtained by taking the left branch to match the string a , reflected in the bitcode [0]. The other match arises by taking the right branch, which matches the regular expression (ab) . The function *mkeys*, as defined by Sulzmann and Lu [9], extracts the bitcode from nullable derivatives. A subexpression is *nullable* if it can match the empty string. Its definition is given at the end of this section.

In Step 3, the resulting derivative expands into an alternative. This occurs because the concatenation has become nullable, which means the first component, r_1 , may be skipped while matching. To account for this, the derivative expands into an alternative: the left branch assumes r_1 is not skipped, while the right branch assumes it is skipped and instead takes the derivative of r_1 directly. This illustrates why derivatives tend to grow in size. Sulzmann and Lu [9] use $\text{fuse}(\text{mkeys}(r_1))$ to include the bit annotations needed when r_1 is skipped; these bits, extracted by *mkeys*, indicate how r_1 matched the empty string.

After taking the derivative with respect to the entire input string, the algorithm checks whether the result is nullable. If so, it calls *mkeys* to extract the bitcode indicating how the match was obtained. In this example, there are two possible matches, but the algorithm prefers the left one. Consequently, *mkeys* returns the bitcode [1, 1], which encodes the choice of the right branch in the first alternative of r_1 (matching the string ab), followed by the right branch in the second part of the concatenation (matching the empty string).

The final bitcode after calling *mkeys* is [1, 1]. Decoding this against the original expression yields the POSIX value:

$$\text{Seq}(\text{Right}(\text{Seq}(a, b)), \text{Right}(\text{Empty}))$$

As mentioned earlier, the formal definitions of the auxiliary functions *intern*, *fuse*, and *mkeys* are given below, as defined by Sulzmann and Lu [9].

- $\text{fuse} : bs, r \rightarrow r'$

$$\begin{aligned}
\text{fuse } bs \ (\emptyset) &\stackrel{\text{def}}{=} \emptyset \\
\text{fuse } bs \ (\varepsilon_{bs'}) &\stackrel{\text{def}}{=} \varepsilon_{(bs \cup bs')} \\
\text{fuse } bs \ (c_{bs'}) &\stackrel{\text{def}}{=} c_{(bs \cup bs')} \\
\text{fuse } bs \ ((r_1 + r_2)_{bs'}) &\stackrel{\text{def}}{=} (r_1 + r_2)_{(bs \cup bs')} \\
\text{fuse } bs \ ((r_1 \cdot r_2)_{bs'}) &\stackrel{\text{def}}{=} (r_1 \cdot r_2)_{(bs \cup bs')} \\
\text{fuse } bs \ (r_{bs'}^*) &\stackrel{\text{def}}{=} (r^*)_{(bs \cup bs')} \\
\text{fuse } bs \ (r_{bs'}^n) &\stackrel{\text{def}}{=} (r^n)_{(bs \cup bs')}
\end{aligned}$$

- $\text{intern} : r \rightarrow r'$

$$\begin{aligned}
\text{intern}(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\
\text{intern}(\varepsilon) &\stackrel{\text{def}}{=} \varepsilon \\
\text{intern}(c) &\stackrel{\text{def}}{=} c \\
\text{intern}(r_1 + r_2) &\stackrel{\text{def}}{=} \text{fuse}([0], \text{intern}(r_1)) + \text{fuse}([1], \text{intern}(r_2)) \\
\text{intern}(r_1 \cdot r_2) &\stackrel{\text{def}}{=} \text{intern}(r_1) \cdot \text{intern}(r_2) \\
\text{intern}(r^*) &\stackrel{\text{def}}{=} (\text{intern}(r))^*
\end{aligned}$$

- $\text{mkeys} : r \rightarrow bs$

$$\begin{aligned}
\text{mkeys}(\varepsilon_{bs}) &\stackrel{\text{def}}{=} bs \\
\text{mkeys}((r_1 + r_2)_{bs}) &\stackrel{\text{def}}{=} \begin{cases} bs \cup \text{mkeys}(r_1) & \text{if } \text{nullable}(r_1) \\ bs \cup \text{mkeys}(r_2) & \text{otherwise} \end{cases} \\
\text{mkeys}((r_1 \cdot r_2)_{bs}) &\stackrel{\text{def}}{=} bs \cup \text{mkeys}(r_1) \cup \text{mkeys}(r_2) \\
\text{mkeys}((r^*)_{bs}) &\stackrel{\text{def}}{=} bs \cup [1]
\end{aligned}$$

2.1.2 Size Explosion.

Even with aggressive simplifications—as shown by Sulzmann and Lu and another variant of the algorithm in *POSIX Lexing with Bitcoded Derivatives* [10]—the number of derivatives, though finitely bounded, can still grow

significantly, making practical usage difficult. The size of derivatives can become very large even for simple expressions. Consider the case $(a + aa)^*$. Here, each derivative step may introduce new structure, unfolding all possible ways the r^* expression can match the input.

Tan and Urban showed that even with simplifications such as removing redundant subterms and collapsing identical alternatives, the number of derivatives still grows to extremely large sizes. This happens because derivatives can reintroduce the same sublanguage in different syntactic forms (e.g. $a + aa$ versus $a \cdot (1 + a)$), which such rules and duplication removal do not recognise as equal [10].

Even Antimirov’s partial derivatives [1], which do not track POSIX values, may still produce cubic growth in the worst cases [1].

*** check Chensong example of size even with simplifications ***

2.2 Marked Approach

The marked approach is a method for regular expression matching that tracks progress by inserting marks into the expression. As noted by Nipkow and Traytel [6], the idea can be traced to earlier work; they cite Fischer et al. [4] and Asperti et al. [2] as reviving and developing it in a modern setting.

This approach allows for more efficient matching, particularly in complex expressions. The regular expression itself does not grow in size; instead, marks are inserted into it. With each input character, these marks move (or shift) according to a set of rules. At the end of the input, the expression is evaluated to determine whether the marks are in positions that make the expression final—that is, whether the expression accepts the string.

There is a slight difference in how marks are interpreted in the works of Fischer et al. [4] and Asperti et al. [2]. In Fischer et al.’s approach, marks are inserted after a character has been matched, thereby recording the matched character or subexpression. In contrast, Asperti et al. interpret the positions of marks as indicating the potential to match a character: as input is consumed, the marks move through the regular expression to indicate the next character that can be matched. In both approaches, acceptance is determined by evaluating the final state of the regular expression. For Fischer et al., this corresponds to having matched characters, whereas for Asperti et al. it requires that the marks end in positions where the expression can accept the empty string, ensuring that the entire input has been consumed. If the marks do not reach such positions, some characters remain unmatched

and the expression is rejected.

2.2.1 Motivation for a Marked Approach

The marked approach offers an alternative to derivatives for regular expression matching. It relies on propagating markers within the regular expression rather than constructing new subexpressions. Our main motivation is that this method could support fast and high-performance matching with POSIX value extraction, since it handles matching in a way that avoids some of the limitations of the derivative-based approach.

In the derivative method—for example, in the sequence case—the size of the expression typically increases due to the creation of new subexpressions, which contributes to the size explosion problem. By contrast, in the marked approach, matching achieves a similar result by propagating marks through the regular expression without generating larger expressions. We also hope that these marks can be used to extract POSIX values in an efficient manner.

Inspired by the works of Fischer et al. and Asperti et al. [4, 2], we aim to extend the marked approach to extract POSIX values, as well as to handle complex constructors used in modern regular expressions, such as bounded repetitions and intersections. Our work is primarily based on the algorithm described by Fischer et al. [4]. As shown by Nipkow and Traytel [6], the pre-mark algorithm of Asperti et al. is in fact a special case of the post-mark algorithm of Fischer et al., which makes Fischer et al.’s approach the most suitable foundation for extending the marked algorithm to matching with POSIX value extraction.

2.2.2 Scala Implementation of Fischer’s Marked Approach

In Fischer et al. approach, the marks are shifted through the regular expression with each input character. The process starts with an initial mark inserted at the beginning, which is then moved step by step as the input is consumed. This behaviour is implemented by the function *shift*, which performs the core logic of the algorithm. The initial specification of this function is given below, as we have developed several versions throughout our work.

The following presents the Scala implementation of the shifting behaviour as originally defined by Fischer et al. [4]. The *shift* function takes as input a regular expression to match against, a flag *m*, and a character *c*, and returns a *marked regular expression*—that is, a regular expression annotated with

marks. We write a marked regular expression as \bullet, r , where the preceding dot indicates that the expression r has been annotated with marks to record the progress of matching.

$$\text{shift} : (m, c, r) \rightarrow \bullet r$$

The flag m indicates the mode of operation: when set to `true`, a new mark is introduced; otherwise, the function shifts the existing marks. This was realised in our first implementation by adding a boolean attribute to the character constructor to represent a marked character. In later versions, we instead introduced a wrapper constructor around the character constructor to explicitly represent a marked character.

$$\text{shift}(m, c, \emptyset) \stackrel{\text{def}}{=} \emptyset$$

$$\text{shift}(m, c, \varepsilon) \stackrel{\text{def}}{=} \varepsilon$$

$$\text{shift}(m, c, d) \stackrel{\text{def}}{=} \begin{cases} \bullet d & \text{if } c = d \wedge m \\ d & \text{otherwise} \end{cases}$$

$$\text{shift}(m, c, r_1 + r_2) \stackrel{\text{def}}{=} \text{shift}(m, c, r_1) + \text{shift}(m, c, r_2)$$

$$\text{shift}(m, c, r_1 \cdot r_2) \stackrel{\text{def}}{=} \begin{cases} \text{shift}(m, c, r_1) \cdot \text{shift}(\text{true}, c, r_2) & \text{if } m \wedge \text{nullable } r_1 \\ \text{shift}(m, c, r_1) \cdot \text{shift}(\text{true}, c, r_2) & \text{if } \text{fin}(r_1) \\ \text{shift}(m, c, r_1) \cdot \text{shift}(\text{false}, c, r_2) & \text{otherwise} \end{cases}$$

$$\text{shift}(m, c, r^*) \stackrel{\text{def}}{=} \begin{cases} \text{shift}(\text{true}, c, r^*) & \text{if } \text{fin}(r) \\ \text{shift}(m, c, r^*) & \end{cases}$$

Shifting marks for the base cases \emptyset and ε is straightforward: \emptyset cannot be marked, and ε —for now—will not carry a mark, since it matches only the empty string. In the initial algorithm, ε was not marked, and this choice is carried over into later versions. The reason is to avoid complications and ensure termination of the *shift* function, some made apparent in the latter

versions as we will discuss further in subsequent sections.¹ The behaviour of the remaining cases is described next.

- **Character case** (d): In this case, if the input character c matches d and the flag m is true, a mark is added and stored in the character constructor. Otherwise, the character remains unmarked.
- **Alternative case** ($r_1 + r_2$): Marks are shifted into both subexpressions, since either branch may match the input character.
- **Sequence case** ($r_1 \cdot r_2$):
 - If r_1 is neither nullable nor in a final position, marks are shifted only into r_1 , indicating that matching proceeds with the first component.
 - If r_1 is nullable and may be skipped, marks are shifted into both r_1 and r_2 , so that either component can begin matching.
 - If $\text{fin}(r_1)$ holds, meaning r_1 has finished matching, marks are shifted into r_2 to continue matching with its component.
- **Star case** (r^*): Marks are shifted into the subexpression if m is true or if $\text{fin}(r)$ holds.

The formal definitions of the auxiliary functions fin and nullable are given below, as defined by Fischer et al. [4].

- $\text{fin}(r) \rightarrow \text{Boolean}$

$$\begin{aligned}
 \text{fin}(\emptyset) &\stackrel{\text{def}}{=} \text{false} \\
 \text{fin}(\varepsilon) &\stackrel{\text{def}}{=} \text{false} \\
 \text{fin}(c) &\stackrel{\text{def}}{=} \text{false} \\
 \text{fin}(\bullet c) &\stackrel{\text{def}}{=} \text{true} \\
 \text{fin}(r_1 + r_2) &\stackrel{\text{def}}{=} \text{fin}(r_1) \vee \text{fin}(r_2) \\
 \text{fin}(r_1 \cdot r_2) &\stackrel{\text{def}}{=} (\text{fin}(r_1) \wedge \text{nullable}(r_2)) \vee \text{fin}(r_2) \\
 \text{fin}(r^*) &\stackrel{\text{def}}{=} \text{fin}(r)
 \end{aligned}$$

¹Our choice follows Fischer et al. [4], where ε is left unmarked (‘shift EPS = EPS’). Asperti et al. [2], on the other hand, use pointed regular expressions (pREs) where acceptance of the empty string is represented by the trailing point being set to true.

- $nullable(r) \rightarrow \text{Boolean}$

$$\begin{aligned}
nullable(\emptyset) &\stackrel{\text{def}}{=} \text{false} \\
nullable(\varepsilon) &\stackrel{\text{def}}{=} \text{true} \\
nullable(c) &\stackrel{\text{def}}{=} \text{false} \\
nullable(r_1 + r_2) &\stackrel{\text{def}}{=} nullable(r_1) \vee nullable(r_2) \\
nullable(r_1 \cdot r_2) &\stackrel{\text{def}}{=} nullable(r_1) \wedge nullable(r_2) \\
nullable(r^*) &\stackrel{\text{def}}{=} \text{true}
\end{aligned}$$

3 Our Approach

We began our work by implementing the marked approach described by Fischer et al. [4] in Scala. This initial implementation of the algorithm provided only acceptance checking without any value construction.

Over the course of this work, we developed several versions of the algorithm, each addressing specific challenges. The first version extended the marked approach with bit annotations, producing values but not always the POSIX-preferred ones. This happens in cases where the marks gets overwritten when a character is matched more than once.

The second version was developed to address two main issues we faced: the overwriting of marks and the absence of a mechanism to order them so as to preserve the POSIX-preferred value. Initially, we modified the previous version to accumulate all possible paths to a match, retaining every bit sequence that could lead to acceptance. Through reasoning and testing, we kept refining it to the point where we are fairly confident that it can produce all possible values for a given string and regular expression, including the POSIX-preferred value. One resulting difficulty was the proliferation of matches in the *Star* (r^*) case, where every possible way of matching was generated. Another difficulty, which also existed in the first version, was the absence of an ordering for marks during the shifting process. We implemented an ordering after shifting to evaluate the results of the algorithm, following the work of Okui and Suzuki [7], though not inherently in the shifting process itself. This line of thought eventually led us to the final version of the algorithm, which uses string-annotated marks, where marks carry the matched string along as they are shifted through the regular expression and then added bit annotation to the marks to record the choices made during matching. The following subsections describe these different versions.

3.1 Bit-Annotated, version 1

In this version, we extended the marked approach to include bitcodes that annotate the marks being shifted through the regular expression. Inspired by Sulzmann and Lu [9], we introduced bitcodes in the form of lists attached to each mark, which are incrementally built as the marks are shifted. This version produces a value, though not necessarily the POSIX-preferred one, because when a character is matched more than once at the same point, the associated bit list may be overwritten. This can cause value erasure and, in some cases, the loss of the POSIX-preferred value, as illustrated later in Example 2. We use the bit annotations 0 and 1, similar to the bitcoded derivatives described by Sulzmann and Lu [9].

The function *shift* takes an additional argument, *Bits*, which is a list of bit elements (0 or 1). As *shift* is applied, bits are appended to the list. For example, when shifting through an alternative, 0 is added to the list passed to the left subexpression and 1 to the list passed to the right subexpression. If the input character matches a leaf character node, it is wrapped by the newly defined *POINT* constructor, which represents the mark. The associated bit list is stored inside this constructor together with the character.

We also define an additional function, *mkfn*, which extracts the bit sequence of a final constructor— that is, the path in bits describing how the expression matched. In addition, we adjust the definition of *mkeps* (originally given by Sulzmann and Lu [9]), which extracts the bit sequence of a nullable expression matching the empty string— that is, the path that led to the empty-string match. The definitions are given below, starting with the *shift* function.

$$\begin{aligned}
\text{shift}(m, bs, c, \emptyset) &\stackrel{\text{def}}{=} \emptyset \\
\text{shift}(m, bs, c, \varepsilon) &\stackrel{\text{def}}{=} \varepsilon \\
\text{shift}(m, bs, c, d) &\stackrel{\text{def}}{=} \begin{cases} \bullet_{bs} d & \text{if } m \wedge d = c \\ d & \text{otherwise} \end{cases} \\
\text{shift}(m, bs, c, r_1 + r_2) &\stackrel{\text{def}}{=} \text{shift}(m, bs \oplus 0, c, r_1) + \text{shift}(m, bs \oplus 1, c, r_2) \\
\text{shift}(m, bs, c, r_1 \cdot r_2) &\stackrel{\text{def}}{=} \begin{cases} \text{shift}(m, bs, c, r_1) \cdot \text{shift}(\text{true}, bs \cup m\text{keps}(r_1), c, r_2) & \text{if } m \wedge \text{nullab} \\ \text{shift}(m, bs, c, r_1) \cdot \text{shift}(\text{true}, mk\text{fin}(r_1), c, r_2) & \text{if } \text{fin}(r_1) \\ \text{shift}(m, bs, c, r_1) \cdot \text{shift}(\text{false}, [], c, r_2) & \text{otherwise} \end{cases} \\
\text{shift}(m, bs, c, (r)^*) &\stackrel{\text{def}}{=} \begin{cases} (\text{shift}(m, bs \oplus 0, c, r))^* & \text{if } m \\ (\text{shift}(\text{true}, bs \cup (mk\text{fin}(r) \oplus 1), c, r))^* & \text{if } m \wedge \text{fin}(r) \\ (\text{shift}(\text{true}, mk\text{fin}(r) \oplus 0, c, r))^* & \text{if } \text{fin}(r) \\ (\text{shift}(\text{false}, [], c, r))^* & \text{otherwise} \end{cases}
\end{aligned}$$

Here, \oplus stands for appending a bit to each element of a list, while \cup stands for concatenating two lists. The symbols 0 and 1 are used to represent left and right choices in alternatives. In the *Star* (r^*) case, 0 represents the beginning of an iteration, while 1 represents the end of an iteration.

Alternative case ($r_1 + r_2$):

- Marks are shifted as before, and the direction of the match is annotated: 0 is added to the bit list passed to the left subexpression, and 1 is added to the bit list passed to the right subexpression.

Sequence case ($r_1 \cdot r_2$):

- If r_1 is nullable, a mark is shifted to both r_1 and r_2 : bs is passed to r_1 (representing the path to this expression), and $bs \cup m\text{keps}(r_1)$ is passed to r_2 , where $m\text{keps}$ returns the bits for an empty-string match. This corresponds to the case where the first part of the sequence is skipped.

- If r_1 is in a final position (that is, it has finished matching), a mark is shifted to r_2 with the bit list describing how r_1 was matched, extracted using the *mkfin* function.
- Otherwise, marks are shifted only into r_1 , with *bs* representing the current path to r_1 .

Star case (r^*):

- If a new mark is introduced, *bs* is passed with 0 appended, representing the beginning of a new iteration of the star.
- If a new mark is introduced and r is in a final position, $bs \cup mkfin(r)$ is passed with 1 appended, combining the bits describing the path to r^* with the bits showing how r reached a final position.
- If r is in a final position, *mkfin*(r) is passed with 0 appended, representing the start of a new iteration.
- If no new mark is introduced, the existing marks are shifted with an empty bit list.

Next, we present two examples of matching a string and extracting a value. The first example shows how the bit list is constructed during matching, while the second demonstrates a case where the algorithm fails to produce the POSIX-preferred value. When shifting to a point (an already marked character) and the character matches again, the associated bit list is overwritten which can lead to value erasure and, in particular, to the loss of the POSIX-preferred value, as the second example illustrates.

1. String *ba*, regular expression: $a \cdot b + b \cdot a$

$$shift\ b \rightarrow (a \cdot b) + ([1] \bullet b \cdot a)$$

$$shift\ a \rightarrow (a \cdot b) + (b \cdot [1] \bullet a)$$

With no further calls to *shift*, *mkfin* is applied because the regular expression has reached a final position, indicated by a mark at the end of the right-hand subexpression in the alternative. *mkfin* then retrieves the bit list [1], corresponding to taking the right branch in the alternative.

2. String *aaa*, regular expression: $(a + a \cdot a)^*$

$$\text{shift } a \rightarrow ([0,0] \bullet a + [0,1] \bullet a \cdot a)^*$$

$$\text{shift } a \rightarrow ([0,0,0,0] \bullet a + [0,0,0,1] \bullet a \cdot [0,1] \bullet a)^*$$

$$\text{shift } a \rightarrow ([0,0,0,0,0,0] \bullet a + [0,0,0,0,0,1] \bullet a \cdot [0,0,0,1] \bullet a)$$

After the first shift on *a*, a mark is placed on the left branch with bits $[0, 0]$, indicating the start of a *Star* (r^*) iteration followed by a left choice. In the right subexpression, the mark on r_1 of the $a \cdot a$ sequence carries bits $[0, 1]$, representing the start of the *Star* (r^*) iteration followed by a right choice.

After the second shift, however, the right subexpression r_2 with bits $[0, 1]$ is overwritten during the third shift, when those bits should instead be preserved. These bits correspond to the POSIX-preferred match, which starts by matching the right-hand side first, then performing another iteration to match the left-hand side. The correct bit list in that case would be $[0, 1, 0, 0]$, with the final 1 marking the end of the *Star* iteration. This behaviour arises because *POINT* stores only a single bit list at a time, with no mechanism for preserving multiple marks.

*** below are more of a self note ***

3.2 Bit-Annotated, version 2

we are fairly sure/strongly think that this version produces all possible values including the posix value.

3.3 String-Carrying Marks

In this version, we modified the marks to have them carry the input string. initially, the full string is added to the initial mark which will be shifted through the regex, each time a character match, the character will be removed from the string of the mark. a match happens when there is a mark with an empty string/matching the empty string. marks are organized in order of posix value, we are fairly sure/think of that. basically, the only reordering happens at SEQ case, after shifting through the first part, this is to reorder the marks based on remaining strings meaning that the marks with shorter remaining strings will be at the front of the list.

4 Future Work

*** from previous report ***

This project focuses on implementing and validating a correct and efficient marked regular expression matcher under POSIX disambiguation. Several directions remain open and are planned for the next stages of the PhD:

- **POSIX Disambiguation for STAR.** While the current matcher correctly computes POSIX values for many expressions, disambiguation for nested or ambiguous STAR patterns is not yet complete. Ensuring that the correct POSIX-preferred value is selected in all cases involving repetition remains a primary target. The current implementation explores candidate paths, but the disambiguation logic for selecting among them requires refinement and formal confirmation.
- **Support for Additional Operators.** Beyond the basic constructs (ALT, SEQ, STAR, NTIMES), future work includes extending the matcher to handle additional regex operators such as intersection, negation, and lookahead. These additions require careful definition of how

marks behave and how disambiguation should be handled, but could significantly increase the expressiveness of the engine.

- **Formal Proof of POSIX Value Correctness.** A formal verification is planned to prove that the marked matcher always produces the correct POSIX-disambiguated value. This would involve defining the decoding function rigorously and proving its output corresponds to the POSIX-preferred parse. This direction is part of the original PhD proposal, where value extraction and correctness proofs were identified as key goals.

References

- [1] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
- [2] A. Asperti, C. S. Coen, and E. Tassi. Regular Expressions, au point. *arXiv*, <http://arxiv.org/abs/1010.2604>, 2010.
- [3] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, Oct. 1964.
- [4] S. Fischer, F. Huch, and T. Wilke. A Play on Regular Expressions: Functional Pearl. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 357–368, 2010.
- [5] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 189–195. ACM, 2011.
- [6] T. Nipkow and D. Traytel. Unified decision procedures for regular expression equivalence. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving*, pages 450–466, Cham, 2014. Springer International Publishing.
- [7] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. Technical Report

Technical Report 2013-002, The University of Aizu, Language Processing Systems Laboratory, June 2013.

- [8] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009.
- [9] M. Sulzmann and K. F. Lu. Posix regular expression parsing with derivatives. *Science of Computer Programming*, 89:179–193, 2014.
- [10] C. Tan and C. Urban. *POSIX Lexing with Bitcoded Derivatives*, pages 26:1–26:18. Apr. 2023.